# How to build a Split-Trust Hardware and its OS

A Technical Reference Manual

ZHIHAO "ZEPHYR" YAO, SEYED MOHAMMADJAVAD SEYED TALEBI, MINGYI CHEN, ARDALAN AMIRI SANI, and THOMAS ANDERSON

Split-Trust hardware is a novel architecture that minimizes the number and complexity of hardware and software components that a smartphone owner needs to trust. The hardware design is composed of statically-partitioned, physically-isolated trust domains.

We create a few simple, formally-verified hardware components to enable a program to gain provably exclusive and simultaneous access to both computation and I/O on a temporary basis. To manage this hardware, we create OctopOS, an OS composed of mutually distrustful subsystems.

This tutorial introduces the implementation details of split-trust hardware and OctopOS. First, we discuss the implementation of split-trust hardware, including its hardware components and the partitioning of the hardware into trust domains. To this end, we highlight our mailbox and its delegation model, domain power management, domain-bound DMA, formal verification efforts, and other hardware technical details. We also discuss the challenges of implementing a multi-domain architecture on a commodity CPU-FPGA board. Note that split-trust hardware does not rely on the programmability of FPGA; instead, we use the CPU-FPGA board to prototype the hardware design. We expect a split-trust architecture to be implemented as ASIC in a commodity smartphone SoC.

Second, we discuss the implementation of OctopOS, which is a de-centralized operating system scattered among different domains, and how it interacts with the split-trust hardware. OctopOS mainly consists of a resource manager, I/O services, a TEE runtime, a Linux compatibility layer, and I/O services. We currently support I/O services for *keyboard*, *serial out*, *storage*, *network*. The guide provide a general guideline for the readers to implement other I/O services. OctopOS is mainly implemented in C on top of Xilinx's libraries.

## Contents

## 1 HARDWARE OVERVIEW

Split-Trust hardware splits a traditional computer into multiple trust domains, each of which is physically isolated from the others. The hardware design is based on the following principles: (1) Domains must be *physically isolated* (i.e., share no hardware components, and each domain implements its own processor and memory). (2) Mailboxes along with other hardware components, which are formally-verified, enforce *exclusively-used* communication. (3) The hardware design is *flexible* and can be easily extended to support new TEE and I/O domains. Figure 1 shows a simplified overview of the split-trust hardware.

### 1.1 Mailbox

Mailbox is a key component of the split-trust hardware, which provide a communication channel between domains. The communication channel is *exclusively-used*, meaning that only one domain can use the mailbox at a time. To use the channel, a domain must first acquire the mailbox from the Resource Manager, which is a special domain that manages the hardware resources of the split-trust hardware. The Resource Manager is not trusted by the other domains, and therefore, the mailbox is implemented to ensure that the Resource Manager cannot interfere with the communication.

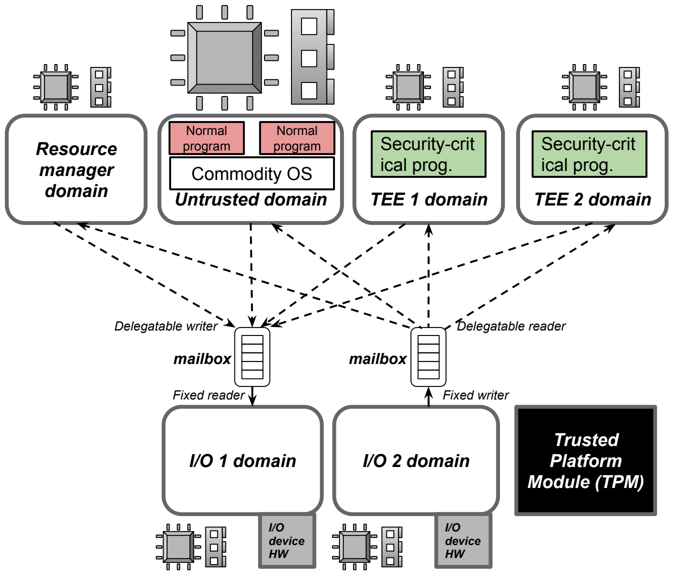


Fig. 1. *Simplified overview of the split-trust hardware. The figure does not show all mailboxes for clarity.*

## 1.2 Domain Separation

Each domain has its own processor. We use a powerful processor for the untrusted domain, which accommodates a commodity OS and its (untrusted) programs, to achieve high performance. For other domains, we use weaker Microblaze microcontrollers to keep the hardware cost small. Internal to each domain, we use an AXI bus to connect the processor, memory, and other hardware components (e.g., interrupt controller, mailboxes, reset guard).

Each domain has its own memory as well and domains do not (and cannot) share memory. In our prototype, we use the main memory for the untrusted domain. For other domains, we use a small amount of on-chip Block Memory (BRAM).

A TEE domain is responsible for executing a TEE program. TEE domains does not have direct access to the I/O devices, and therefore, they must use the I/O services to access the I/O devices.

Each I/O domain also has exclusive control of an I/O device, which is wired to and only programmable by the processor of that domain and which directly interrupts that processor. In the prototype, we implement the keyboard, serial out, storage, and network I/O devices. We use constraints to ensure that the pins of each I/O devices (e.g., network SFP, serial rx and tx) are only mapped to the respective domains.

## 1.3 Domain Memory

As we will discuss in Section 3, we implement a BRAM-based read-only memory (ROM) to store the bootloader. Once booted, the domain starts to use a dedicated BRAM for its memory (RAM), aside from the ROM. Therefore, each domain has two BRAMs: one for the ROM and one for its RAM. We use Vivado's BRAM generator to generate both the BRAMs, and we implemented a simple IP to disable or enable the ROM's memory bus Write Enable (WE) signal. A developer may adjust the RAM and ROM size of each domain in Vivado's address editor, and the BRAM generator will automatically generate the correct RAM size, aligned to the power of two.

## 1.4 Resource Manager Domain

The Resource Manager is a special domain that manages the hardware resources of the Split-Trust hardware. Although it is responsible for delegating the hardware resources to other domains, it is not trusted by the other domains.

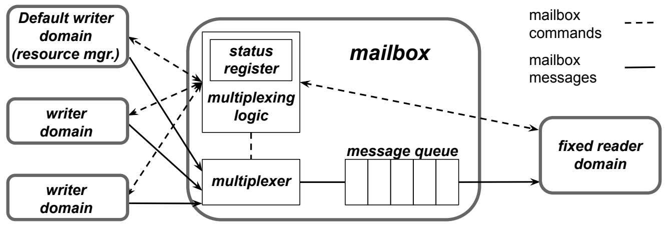


Fig. 2. *Mailbox design.*

The Resource Manager is responsible for delegating the I/O device to a TEE domain. It is also responsible to reset a domain before each delegation to ensure that the domain is in a clean state. We design and implement several hardware components to ensure the Resource Manager is able to correctly manage these resources without being trusted.

## 1.5 Development Platform

We prototype the Split-Trust hardware on the Xilinx ZCU102 development board, and use Xilinx Vivado 2020.1 to implement the design. The board is based on the Xilinx Zynq UltraScale+ MPSoC, which has a programmable logic fabric (600K system logic cells), and a quad-core ARM Cortex-A53 processor [1]. We implement all our design in the programmable logic fabric, except for the untrusted domain which runs Linux on the Cortex-A53 processor. Theoretically, the Split-Trust hardware can be implemented on any FPGA board or ASIC, but since we heavily rely on Xilinx's hardware IPs to implement the Microblaze processors, memory controllers and other hardware components, we expect the porting effort (for example, replacing IPs) to be non-trivial.

## 1.6 The Split-Trust Hardware Emulator

Because the FPGA board may not be available to all researchers, we implement a hardware emulator to facilitate the development of the Split-Trust hardware. The emulator is a software-based emulator that runs on a commodity Linux machine. It emulates the hardware components of the Split-Trust hardware, including the Resource Manager, the mailbox, the TEE domains, the I/O domains, and an untrusted domain. We introduce the implementation and features of the emulator in Section 14.

## 2 MAILBOX

## 2.1 Overview

Mailbox is a key component of the Split-Trust hardware, as it enables domains to communicate with each other. At the core of our delegatable mailbox, we utilize a simple LogiCORE IP Mailbox [2], which is a stateful hardware FIFO, allowing two domains (i.e., a writer and a reader) to communicate through message passing. Although the LogiCORE IP Mailbox is bi-directional, we only use it in one direction to implement our abstraction, i.e., from the writer to the reader. Our mailbox implements a wrapper around the LogiCORE IP Mailbox, which provides delegation capability and enables exclusive communication across domains. For simplicity, "mailbox" in the rest of the paper refers to our delegatable mailbox.

As shown in Figure 2, a mailbox has a fixed end (reader or writer) and a delegatable one. The fixed end is hard-wired to a specific domain. The delegatable one is wired to multiple domains, but only one can use it at a time, enforced by a hardware multiplexer within the mailbox. This end is by default (i.e., after a mailbox reset) under the control of the Resource Manager domain. That means, at first, only the Resource Manager can send/receive messages to the domain on the other side of the mailbox, e.g., to use an I/O device. The Resource Manager can delegate it to another domain, which is then able to *exclusively* communicate with the domain on the fixed end of the mailbox.

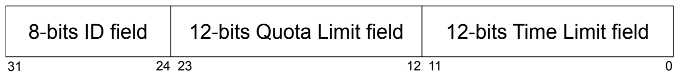


Fig. 3. *Mailbox state register.*

## 2.2 Auto Generation of Mailbox Code

Due to different configurations of our mailboxes (i.e., different directions of communication, and different numbers of delegatable ends), we have developed a Python 3 script to automatically generate the Verilog code for our mailboxes. The script can be found at octopos_hardware/scripts/generate_code_for_MailBox.py. To use the script, a developer need to specify the number of delegatable ends and the direction as parameters to the script. Specifically, if the direction is *many writers, one reader*, the value of the second line should be 1. For example, a serial out domain is the only reader of a mailbox, all other domains write (i.e., print) to this mailbox. If the direction is *one writer, many readers* (e.g., a keyboard), the value of the second line should be 0.

The script will internally instantiate a LogiCORE IP Mailbox, control logics and necessary internal AXI bus and ports. It outputs a Verilog file named *Octopos_MailBox_...v*, which contains the source code for the whole mailbox. The filename reflects the provided configurations. A developer can then include this file in her Verilog code to instantiate a mailbox, or preferably, she can start an empty Vivado project, add the Verilog file to the project, and package it into an IP.

Once the mailbox is packaged into an IP, a developer can easily reuse it by including the IP location into the project's IP Repository, and instantiate the IP in her design. A tutorial on how to create an IP project can be found in the Vivado Design Suite User Guide [3]. We also provide a step-by-step guide to update our mailbox IP at octopos_hardware/docs/Update-Mailbox-IP.rst.

If manual configurations of the LogiCORE IP Mailbox is desired, a developer can do so by opening the IP project, locating the LogiCORE IP Mailbox in the source, double-clicking it, and changing the configurations, such as the depth of FIFO, the type of underlying memory (e.g., Block RAM, Ultra RAM, Distributed RAM), etc.

## 2.3 Mailbox State

We verbosely comment on the Verilog code of the mailbox, and we encourage the reader to read the code to understand the mailbox design in detail. Specifically, our Verilog code labels the input (delegatable) ports as "In0", "In1", etc. In0 is designed differently as it has the initial control of the mailbox. In our design, the Resource Manager is the initial controller of the mailbox, and it can delegate the mailbox to other domains. The rest of the inputs are treated equally.

The state of a mailbox is controlled by *Current Exclusive Owner (CXOwner)*, *Limit*, and *Timeout*. As shown in Figure 3, the fields are stored at an offset of a memory-mapped 32-bit register, which is accessible (both read and write) by the mailbox's current CXOwner. A CXOwner with access to the mailbox can query and modify the state register through a control path, which will be introduced below in this section.

The *CXOwner* field stores the ID of the current domain that has exclusive access to the mailbox. The *Limit* field stores the maximum number of messages allowed to be sent to or received from the mailbox. The *Timeout* field stores the maximum time allowed for the CXOwner to use the mailbox. For both the *Limit* and *Timeout* fields, the value 0 means no access, and value -1 means infinite. Upon reset, *CXOwner* is set to In0, and both *Limit* and *Timeout* are set to -1, granting unlimited access to In0.

The *CXOwner* field has 8 bits, and therefore, the mailbox can support up to 256 domains. The *Limit* field has 12 bits for delegation up to 4096 messages. It is possible to adjust how many data path register reads/writes constitute a single message. The *Timeout* field has 12 bits for delegation up to 4096 units of time. The unit of time is currently set to 1 second. With 1 second granularity, 12 bits of the *Timeout* field can support up to 4096 seconds, or 68 minutes. It is possible to define a smaller time quota (e.g., 1

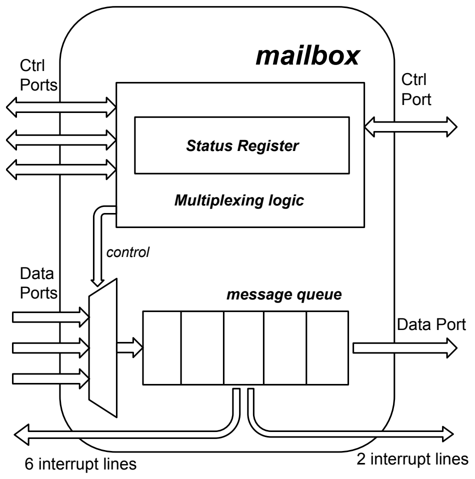


Fig. 4. *Mailbox ports, assuming the mailbox has three delegatable write ends and one fixed read end.*

millisecond) by dividing a smaller amount of clock cycle (assuming a 100MHZ clock source). With higher precision (e.g., 0.1 seconds), the *Timeout* field can support up to 409.6 seconds, or 6.8 minutes.

As shown in Figure 4, the data path of the mailbox is controlled by the multiplexer according to the state, and it is connected to the LogiCORE IP Mailbox for data access. Each possible CXOwner and the fixed owner has a data port for sending or receiving messages.

Similar to the data path, the control path is an AXI interface exposed to every possible CXOwner of the mailbox. The fixed side can always access both the data and control ports. The potential CXOwner can only access its data port and control port when it has exclusive access to the mailbox.

The data path selection is enforced by the multiplexer. The control path selection is enforced by mailbox's hardware logic that checks the current CXOwner. These properties are of vital importance to the security of the mailbox, and thus, they are formally-verified.

A violation happens when a potential CXOwner tries to access the mailbox (either data port or control port) when it is not the CXOwner. For data port violation, the accessing processor will access an invalid (unmapped) MMIO memory because the physical connection to the data path is disconnected by the multiplexer. For control port violations, the behavior of the mailbox can be defined in the hardware logic. The current implementation of the mailbox will simply ignore the control port access (nop when write, and getting 0xFFFFFFFF when read).

## 2.4 Interrupts

Similar to data and control ports, the mailbox has a set of data interrupt lines and another set of control interrupt lines. Every potential CXOwner has one data interrupt and one control interrupt. The interrupt lines are connected to the interrupt controller of each processor.

The data interrupt exposes the interrupt line of the LogiCORE IP Mailbox, after the data path is multiplexed. The control interrupt is generated by the mailbox multiplexing logic whenever the CXOwner is changed, for example, when the mailbox is delegated to another processor, or the CXOwner's delegation is expired.

The interrupt lines are named in the following format: for the fixed owner, the data interrupt line is named *Interrupt_fixed*, and the control interrupt line is named *Interrupt_ctrl_fixed*; for the potential CXOwners, the data interrupt lines are named *Interrupt_In0*, *Interrupt_In1*, etc., and the control interrupt lines are named *Interrupt_ctrl_In0*, *Interrupt_ctrl_In1*, etc.

LogiCORE IP Mailbox has a built-in interrupt threshold logic (i.e., how many pending messages before an interrupt is generated). These thresholds can be configured by the CXOwner processor through the

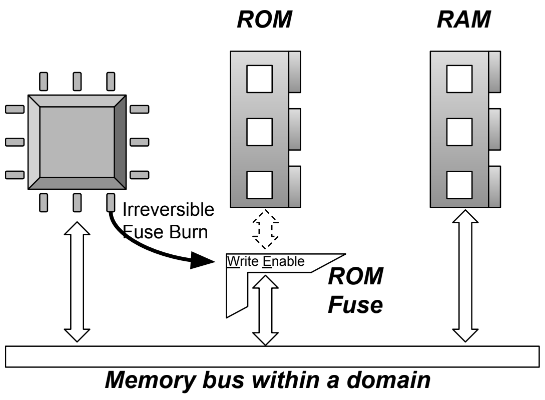


Fig. 5. *The ROM fuse. The dotted line indicates a filtered signal.*

data port, which passthrough to the LogiCORE IP Mailbox. After handling a data interrupt, the processor must clear the interrupt following LogiCORE IP Mailbox's instruction.

The control interrupt is a level interrupt, and it must be cleared by a processor after serving an interrupt. To clear the interrupt, the processor must write a 0x1 to offset 4 at the control port's base address.

We have provided a simple driver for the mailbox's control interface. The driver provides a set of functions to query and modify the mailbox's state, and to handle the mailbox's interrupts. It can be found at octopos/arch/sec_hw/octopos_mbox.c.

### 2.5 Nested Delegation

In the current implementation of Split-Trust hardware, only the Resource Manager can delegate the mailbox to other domains. However, our mailbox supports nested delegation by design. For example, if the Resource Manager delegates the mailbox to domain A, domain A can further delegate the mailbox (the rest of domain A's quota) to domain B. In our early experiments, we found that nested delegation is easy to implement and may be useful. But such a design is not necessary for the current implementation of Split-Trust hardware.

### 2.6 Mailbox Reset

Each mailbox has a reset signal, and it is wired to the reset signal group of the domain that **owns** the mailbox. In Split-Trust hardware, the domain connected to the mailbox's fixed owner interface is usually the **owner** of the mailbox.

Once a domain is reset to flush any state associated with the domain, the mailbox is reset as well. The mailbox resets itself, and vertically passes the reset signal to the LogiCORE IP Mailbox.

The reset of LogiCORE IP Mailbox flushes all the data in the FIFO, and at the same time, flush the configurations of LogiCORE IP Mailbox, such as interrupt thresholds. Therefore, the future owner needs to re-initialize the LogiCORE IP Mailbox after resetting. It should be done by the domain's software, and it is not the responsibility of the mailbox. Specifically, a domain should re-initialize the data path after becoming the owner of a mailbox. The delegation interrupt can be used to notify the domain that it is now the owner. An example of such initialization and re-initialization can be found at octopos/arch/sec_hw/mailbox_interface/mailbox_storage.c.

## 3 ROM FUSE

For the integrity of the bootloader, it must not be modified by any software once loaded into the memory. To achieve this goal, we develop a BRAM-based ROM mechanism, namely ROM fuse, to prevent write access to the bootloader memory. As an alternative, we could pre-populating a ROM IP with the bootloader binary, but this solution is not flexible as it requires hardware synthesis and implementation every time we change the bootloader program.

As shown in Figure 5, ROM fuse is a simple piece of hardware, allowing a piece of BRAM act as RAM at the very beginning of the system's power-up and later can be converted to ROM irreversibly. The ROM fuse controls the BRAM memory bus's Write Enable (WE) signal to allow or disallow write access to the BRAM. At the very beginning of each domain's bootloader program, the bootloader burns the ROM fuse to permanently disable write access to its memory, effectively converting the BRAM to ROM. §7.2 describes how the bootloader burns the ROM fuse.

## 3.1 ROM fuse Implementation

BRAM Fuse has a simple 1-bit internal state register, which indicates whether the fuse has been burnt or not. If the fuse is not burnt, ROM fuse forwards the input WE signal to WE. However, if the fuse is burnt, WE always outputs zero. Hence, no write to the memory is possible after the fuse is burnt. A crucial characteristic of the fuse is that burning it is irreversible. Once the fuse is burnt, our hardware logic prevents any further change to this state. This property is formally proven.

ROM fuse exposes an AXI interface for burning the fuse. A processor can burn the fuse by writing "0xDEADDEAD" to the MMIO-ed base address of the ROM fuse.

## 3.2 Reset Resistance

The irreversible property of our ROM fuse is resistant to the reset signal, which means even when the Resource Manager resets a domain and a reset signal re-initializes all the hardware in the domain including the ROM fuse, a burnt fuse remains burnt. To achieve this, we leverage the fact that when we power on the FPGA board before the reset signal arrives, all the register values have random values. The register we refer to here is any 32-bit AXI MMIO-ed register, similar to the one that we use for the control interface of the ROM fuse, and not the 1-bit state register internal to the ROM fuse. We create another unconnected 32-bit register solely for the purpose of detecting whether the programmable logic is freshly programmed.

The first time the programmable logic is programmed, the ROM fuse is not burnt, and the 32-bit dummy register value is random. Upon detecting the random value, the ROM fuse knows that it is the first time the programmable logic is programmed. When the ROM fuse is first burnt, it will write a fixed value, "0xDEADBEEF", to the dummy register. Therefore, the next time the programmable logic is programmed, the ROM fuse will detect the fixed value in the dummy register, and will immediately burn the fuse within a few clock cycles. The chance of the dummy register value randomly colliding with "0xDEADBEEF" is very low ($2^{-32}$). Although this solution may not work with ASIC, it demonstrates the needed security property of ROM fuse.

## 3.3 ROM Size Adjustment

BRAM size can be adjusted by editing the address range assigned to each BRAM in the address editor. However, because we modify the BRAM memory bus of ROM, adjustment of the ROM size may not propagate due to possible bugs. If this happens, a developer will notice that although the ROM size in the address editor is changed, the actual ROM size is not changed in the generated bitstream. In this case, the developer need to re-initialize the Vivado project.

## 4 DOMAIN RESET AND RESET GUARD

Our physically-isolated domains and the mailbox on its own cannot guarantee session availability. This is because we need to ensure that during a session, the domains used by a security-critical program remain powered up (given that there is enough battery capacity or a stable power supply).

Domains and their mailboxes need to be reset to a clean state after use, specifically, upon delegation, yield, and session expiration. We leave the resetting of the domains to the Resource Manager under the limitations enforced by the reset guards.

Even though the Resource Manager is untrusted, this does not pose a problem since a program can verify, using local and remote attestation through TPM as well as some measures provided by the domain runtime that (1) a domain has been reset, (2) it has not been used since the last reset, (3) it will be reset after use and before use by other domains.

Specifically, a bootloader (Section 7.2), which is stored in a ROM and is part of the remotely attested root of trust, is tasked with fully cleaning all the state information in the domain upon reset. The bootloader calculates the hash of the domain program image, and extends it to its assigned TPM PCR (§6). Furthermore, a I/O domain extends its TPM PCR with zero when it is first used after reset, effectively indicating it has been used since reset. A correct PCR value indicates that the domain has been reset and has not been used since the last reset.

## 4.1 Power Management

To achieve exclusive usage of computation (the TEE domain) and I/O resources (the I/O services through delegatable mailboxes), the Split-Trust hardware must be able to prevent a domain from being reset when it is actively using an I/O device, i.e., in a delegation session. We implement a power management unit (PMU) to manage the power supply of the domains. Our PMU consists of reset lines, the Processor System Reset Module (PSRM) [4], and reset guards. These components are implemented per-domain. It is important to note that the PMU is different from the Xilinx ZCU102's built-in PMU IP. We implement our own PMU instead of utilizing the built-in PMU IP because it is not flexible enough to support individual power management for individual domains.

All hardware components of a domain (including its Microblaze processor, memory, memory controller, interrupt controller, mailboxes, etc.) are wired to a PSRM. The PSRM is a Xilinx hardware IP that generates reset signals for different types of hardware IPs. The reset guard sits between the reset signal of a domain and the PSRM of the domain, to prevent illegal reset requests from reaching the PSRM.

The PMU normally takes commands from the Resource Manager. The Resource Manager uses this capability to reset other domains when needed, e.g., reset a TEE domain before running a new program, or apply Dynamic Voltage Frequency Scaling (DVFS) to manage the system's power consumption. We do not support DVFS for the domains in our prototype. Hence,we mainly focus on the reset interface, although similar principles can be applied to DVFS.

However, the Resource Manager is not a trusted component; hence it may try to reset a domain during a session. Therefore, we add a simple hardware component, called the *reset guard*, for the reset signals, which ensures that as long as the quota on a mailbox has not expired, the domains on both sides of the mailbox cannot be reset, hence ensuring session availability. Once the quota expires (or if the access to the delegable end of the mailbox is yielded), the mailbox is returned back to the Resource Manager and the Resource Manager is allowed to reset and reuse the domains.

## 4.2 Reset Guard Implementation

As shown in Figure 6, the reset guard guarantees that the reset signal is not asserted ,

- when any one of the domain's mailboxes is in use by another domain (other than Resource Manager).
- when the domain is actively using another domain's mailbox.

The reset guard exposes a simple AXI4-Lite interface to the Resource Manager. Specifically, the Resource Manager writes "0xDEADBEEF" and then "0xDEADDEAD" to the base address of the reset guard to initiate a reset request, and reads the value at the base address of the reset guard to query whether

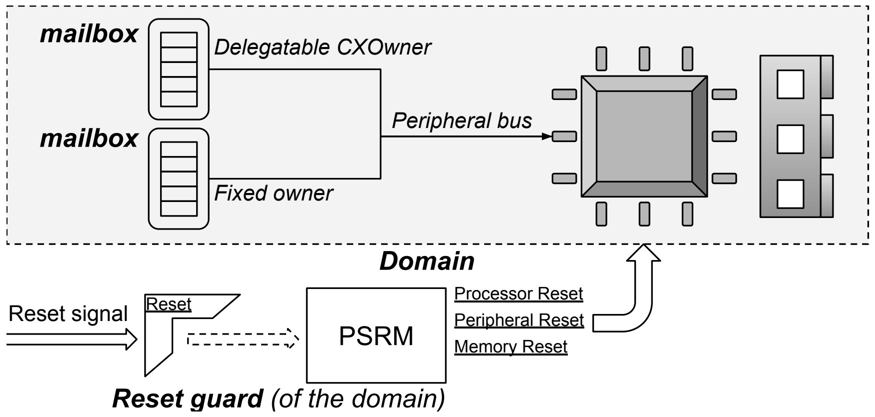


Fig. 6. *The reset guard. The dotted lines indicate a filtered reset signal. The shaded area indicates the domain that is reset.*

the last reset request is blocked or not. The return value “0x0000AAAA” means a successful reset, and “0x0000FFFF” means the reset has been blocked.

The reset guard reads a busy signal (a register) from each relevant mailbox. The mailboxes use these signals to indicate whether they are in use or not. The hardware design details can be found at octopos_hardware/docs/reset_module.rst

As mentioned in Section 3.1, registers are initialized to random value after the programmable logic is programmed. Therefore, the reset guard may block the first reset signal because the mailbox’s busy register is not initialized to “0”. To solve this problem, we use the same technique as in ROM fuse to detect whether the programmable logic is freshly programmed. The reset guard will not block the first reset signal regardless of the busy register value. The first reset signal is triggered by the programmable logic’s power-on reset when the programmable logic is fully programmed.

## 5 DOMAIN-BOUND DMA

By default, the data plane of I/O domains is implemented over mailboxes, as discussed earlier. As an example, consider the network service. When a client needs to send a packet, it sends it to the service, which then queues it on the transmit queue of the network device (e.g., by using DMA or by directly writing the packet onto the queue). This design, however, causes performance overhead due to additional copying to and from a mailbox.

While the performance overhead is acceptable for TEE domains, it is not so for the untrusted domain. An important hardware primitive that enables a legacy machine to achieve high I/O performance is Direct Memory Access (DMA). In legacy systems, DMA is often a security concern as it can access all of the physical memory address space, creating an obstacle for using it in the Split-Trust machine.

To use the performance benefits of DMA while safely integrating it into this machine, we implement *domain-bound DMA*, defined with the following two restrictions. (1) The DMA engine is hard-wired to only read/write to the memory of the untrusted domain, and cannot be programmed to do otherwise. (2) The DMA engine can stream data in and out of the I/O device only when the I/O domain is used by the untrusted domain. Therefore, when an I/O domain is used by a TEE domain, DMA is not used and data is transferred using mailboxes. But when the untrusted domain uses the I/O domain, data is transferred using domain-bound DMA for performance reasons.

### 5.1 Arbiter

We achieve this with a simple hardware component called the *arbiter*, which is a switch that decides if the data streams of the I/O device are connected to a DMA engine or to a simple FIFO queue accessible

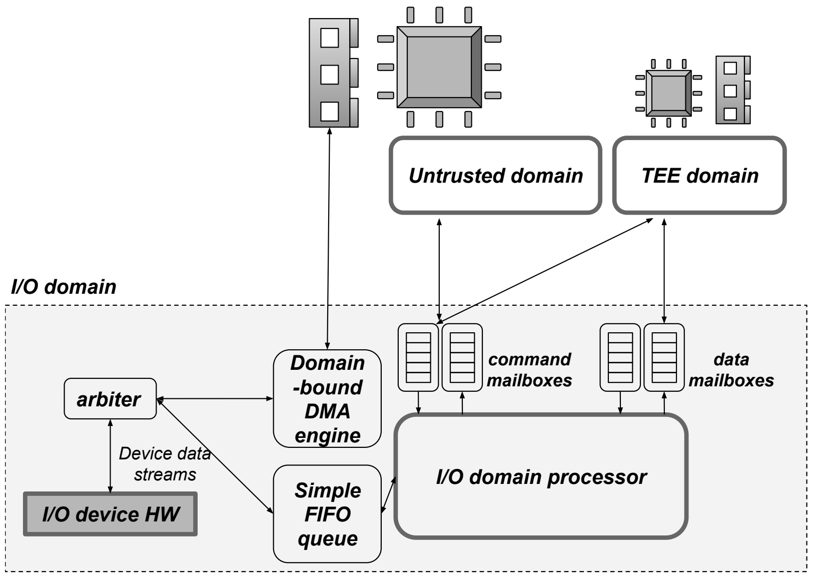


Fig. 7. *Domain-bound DMA design.*

to the I/O domain. As in a legacy machine, the untrusted domain may also use an IOMMU to further restrict DMA targets in order to isolate its own address spaces.

Figure 7 shows this design. The *arbiter* is a simple hardware switch that determines whether the data streams of the I/O device is connected to a DMA engine or a simple FIFO queue. When the untrusted domain is given access to the I/O domain, the arbiter is programmed to connect the streams to the DMA engine, which is hard-wired to the memory of the untrusted domain. Whenever the I/O domain is reset, the arbiter defaults the connection of data streams to the mailbox. In this case, the data is simply moved (read/write) to and from the simple FIFO queue and the mailboxes of the I/O domain.

The multiplexing logic of the arbiter is quite similar to that of the mailboxes. The client domain can verify the state of the arbiter by directly accessing its status register (although this detail is not shown in Figure 7).

We note that this optimization does not impact the I/O protocol. The main difference is the conduit used to transfer the actual data, a mailbox or domain-bound DMA. Therefore, the domain-bound DMA simply implements an optimized data plane for the untrusted domain.

To control the arbiter, the Resource Manager writes the value “0xF0F0F0F0” to the base address of the arbiter to select the DMA engine, and “0x0” to select the mailbox.

## 6 TPM

Trusted Platform Module (TPM), as specified by the Trusted Computing Group (TCG), is a tamper-resistant security co-processor connected to the main processor over a bus [5]. Traditionally, it provides security features for the machine as a whole, such as the measurements of the loaded software , which can then be used for remote attestation. This makes TPM unsuitable for more fine-grained security features, such as remote attestation of a specific program. As a result, in-processor TEE solutions, such as Intel SGX, integrate the root of trust in the processor itself, tightly coupling it with various features of the processor (such as virtual memory and cache), further bloating the trusted processor.

A hardware root of trust is needed during remote attestation to convince the party in charge of a security-critical program of the authenticity of the hardware and the correctness of the loaded program. In a Split-Trust hardware, we use a TPM to realize the root of trust for the Split-Trust hardware. This TPM can provide fine-grained security features for a Split-Trust machine since different components of this OS run on separate processors.

One might argue that a TPM is not as secure against physical attacks as in-processor TEE solutions since the processor needs to communicate with the TPM over the bus. However, we note that this is not fundamental to the TPM protocol. That is, if a TPM is integrated into the chip hosting the processor (e.g., in an SoC), it can enjoy protection against the aforementioned physical attacks.

### 6.1 TPM PCRs & Localities

To integrate TPM into a Split-Trust machine, we need a different set of parameters from the ones found in existing TPM chips. Existing TPMs, while theoretically ideal for a Split-Trust OS, cannot support such a machine because they assume a single trust domain in the machine. Therefore, we make some modifications to the TPM to integrate it into a Split-Trust machine.

We add more *Platform Configuration Registers (PCRs)* to the TPM as well as more *localities*, which define access permissions to the PCRs. Using these additions, we allow each domain to separately and securely keep track of all the software loaded into the domain processor and provide a certified report attesting to that. Modifying the number of PCRs and their access permissions in TPM does not change its fundamental design principles. Indeed, the TPM specification does not specify these parameters [6], leaving them to the implementers.

### 6.2 TPM Hardware Prototyping

Due to the lack of a TPM chip on our development board, we implement an off-board TPM on a Raspberry Pi 3 Model B+, connect it to the ZCU102 board's PMOD connectors, and multiplex the TPM requests from the domains to the Raspberry Pi.

We implement a TPM multiplexer in the programming logic of the ZCU102 board. The TPM multiplexer exists solely for the purpose of prototyping, and is not part of the Split-Trust hardware design. The purpose of the TPM multiplexer is to forward each domain's TPM requests to the Raspberry Pi, and forward the responses from the Raspberry Pi to the corresponding domain. The TPM multiplexer is implemented with a Microblaze microprocessor and a UART interface, which is connected to the Raspberry Pi over PMOD. Each domain has a fixed FIFO (not the delegatable mailbox) to communicate with the TPM multiplexer.

The Raspberry Pi is a single-board computer that runs Debian-based Raspberry Pi OS. We run a TPM emulator [7] on the Raspberry Pi to emulate a TPM chip. The Raspberry Pi is connected to the ZCU102 board over the PMOD connectors, with a single pair of TX/RX lines for the UART interface, and the power and ground lines.

On the Raspberry Pi, we run a TPM emulator [7] (using the TPM2-TSS libraries [5, 8] and the TPM2-ABRMD [9] Resource Manager daemon) to emulate a TPM chip. The TPM emulator allows us to modify it to support multiple localities and PCRs. We expect that the TPM emulator will be replaced by a hardware TPM chip customized for Split-Trust in the future.

## 7 OCTOPOS BOOTLOADER

### 7.1 Platform Initialization

The boot process of the Split-Trust hardware is an orchestrated process that involves several stages. Before the Split-Trust hardware (bitstream) is programmed into the FPGA, the standard Xilinx boot process is executed.

(1) The Split-Trust hardware is powered on.
(2) The jumpers on the board are configured to boot from the board's SD card reader. Specifically, it looks for a file named "BOOT.bin" on the SD card.
(3) Xilinx's First Stage Boot Loader (FSBL) [10] packaged in the binary package is executed on one of the ARM A53 cores.

(4) FSBL loads the bitstream from the binary package, and programs the programmable logic.
(5) At this time, the Split-Trust hardware is fully programmed into the FPGA.
(6) FSBL lunches U-Boot on the ARM A53 core for subsequent booting of the untrusted domain.
(7) FSBL exits.

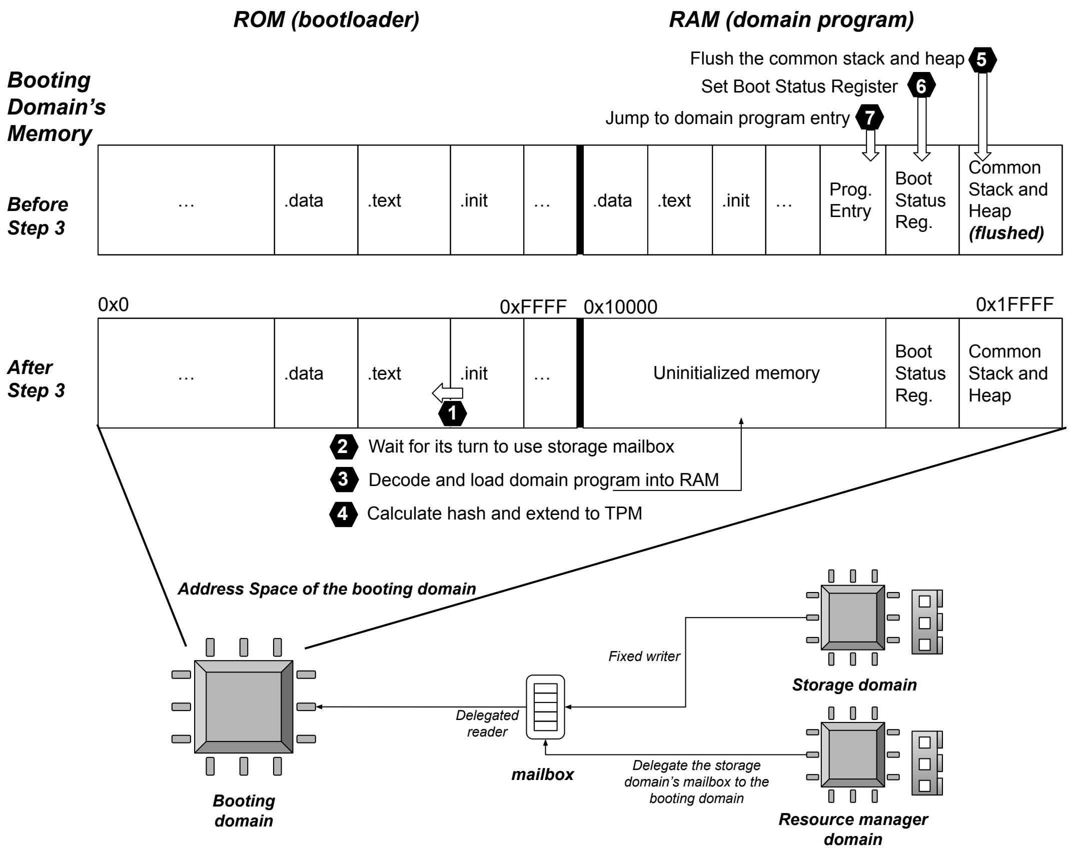
## 7.2 OctopOS Boot

In our final prototype, we use a partitioned region of the main memory as the storage media. As we discuss in §11, using the main memory as the storage media is not necessary for the Split-Trust hardware, but its data transfer speed is much faster than other storage media solutions we have tried (e.g., PMOD-based SD card, and QSPI flash chip).

Within the storage media, we store the boot image of each domain, including the storage domain itself's image, and the untrusted domain's Linux kernel. This initial content of storage media is stored as a partition file file on the SD card. Either FSBL or U-Boot can load the partition file into the partitioned main memory region that is used as the storage media. We implement the data transfer at the very beginning of U-Boot because U-Boot is easier to modify and debug than FSBL.

After the Split-Trust hardware is programmed, the boot process continues. The Split-Trust hardware is fully programmed into the FPGA, and all the ROMs (technically, RAM, because their ROM fuses are not yet burnt) are prepopulated with the boot code (domain bootloaders). The bootloaders are programmed into the ROMs using Xilinx's "updatemem" tool. The process is automated in octopos/Makefile.

(1) All domains' bootloaders start to run in their respective processors. The bootloaders burn their respective ROM fuses.
(2) At this time, only the storage domain bootloader has access to storage. Storage domain bootloader loads the storage domain image from its storage media into the memory of the storage domain and exits.
(3) The storage domain fully boots and initializes.
(4) The Resource Manager domain bootloader finishes waiting for the storage domain to boot. It loads the Resource Manager domain image from the storage domain (using its API called cover mailbox, which the Resource Manager domain has default access to). The Resource Manager bootloader uses a simple file system to read the right blocks of data containing the Resource Manager domain image. The Resource Manager bootloader exits.
(5) The Resource Manager domain fully boots and initializes.
(6) The bootloaders of the I/O domains and TEE domains finish waiting for the Resource Manager domain to boot. Each of them waits on their turn to use the storage mailbox.
(7) The Resource Manager domain grants each domain access to the storage mailbox in a round-robin fashion. To do so, it invokes the storage service API to transfer the required images, and delegates the data plane to the corresponding domain.
(8) Each I/O domain and TEE domain bootloader receives the images, and loads its images from the storage domain (using mailbox). Once the domain image is loaded, the bootloaders exit, allowing for the next domain to use storage to load its image.
(9) Each I/O domain and TEE domain fully boots and initializes. This happens in parallel for I/O and TEE domains.
(10) The untrusted domain bootloader loads the untrusted domain's Linux kernel from the storage domain (using mailbox).
(11) The untrusted domain Linux kernel boots, and mount the storage domain as the root file system. The untrusted domain opportunistically requests delegation of the storage mailbox from the Resource Manager domain.
(12) The untrusted domain Linux kernel launches a customized rootfs from the storage domain.



ROM (bootloader)
RAM (domain program)
Booting Domain's Memory
Flush the common stack and heap 5
Set Boot Status Register 6
Jump to domain program entry 7
Before Step 3
…
.data
.text
.init
…
.data
.text
.init
…
Prog. Entry
Boot Status Reg.
Common Stack and Heap (flushed)
0x0
0xFFFF
0x10000
0x1FFFF
After Step 3
…
.data
.text
.init
…
Uninitialized memory
Boot Status Reg.
Common Stack and Heap
1
2 Wait for its turn to use storage mailbox
3 Decode and load domain program into RAM
4 Calculate hash and extend to TPM
Address Space of the booting domain
Fixed writer
Storage domain
Delegated reader
mailbox
Delegate the storage domain's mailbox to the booting domain
Booting domain
Resource manager domain

Fig. 8. *Boot process of an individual domain. The figure does not show all binary sections for simplicity. The figure assumes the size of the ROM and the RAM are both 0xFFFF. Prog stands for program, intr stands for interrupt, tram stands for trampoline, reg stands for register.*

Each individual domain's booting process includes extending the measurement of their respective domain image to the corresponding TPM PCR.

Our measurements show that it takes approximately 4 seconds to boot all domains excluding the untrusted domain, which takes an additional 8.65 seconds to boot the Linux kernel and the rootfs (all from our storage domain).

## 7.3 Bootloader of an individual domain

Each domain's bootloader is a simple program that is responsible for loading the domain's image from the storage domain into the domain's memory. The storage domain is special because its bootloader has direct access to the storage media. The storage domain's bootloader is responsible for loading the storage domain image from the storage media into the storage domain's memory. The Resource Manager domain is the default owner of the storage mailbox. Its bootloader waits for the storage domain to boot, and then loads the Resource Manager domain image from the storage domain (using the storage mailbox and a simple file system).

Besides these two special cases, the bootloaders of the I/O domains and TEE domains are very similar (share the same code). Figure 8 shows the boot process of an individual domain.

First, the bootloader's main function is called. The bootloader burns the ROM fuse of its ROM, making it read-only. Second, the bootloader waits for the Resource Manager domain to boot, and then waits for its turn to use the storage mailbox. Third, once the Resource Manager domain delegates the storage mailbox to the domain, the bootloader loads the domain image from the storage domain. While it reads each block of data (512 bytes per mailbox message), it decodes the data, writes the data into the domain's

RAM, and updates a rolling SHA256 hash of the data. The domain image data is in SREC format. The details of our SREC loader are discussed in §7.4. Fourth, once the bootloader finishes loading the domain image, it extends the finalized hash to the corresponding TPM PCR. Fifth, the bootloader flushes the Common Stack and Heap. We describe the details of the need for Common Stack and Heap in §7.5. Sixth, the bootloader sets the Boot Status Register, which is needed for implementing a trampoline mechanism for interrupt handling (Section 8.1). Seventh, the bootloader jumps to the entry point of the domain image.

## 7.4 Loading Domain Images in SREC Format

Each domain binary is compiled and linked with the Vivado Vitis IDE into elf binary format. To be dynamically loaded, elf binaries must be converted into SREC files. SREC is a simple format for storing binary data in ASCII text. We use "mb-objcopy" tool for the conversion from elf to SREC format. This process has been automated in octopos/Makefile.

Bootloaders load the domain images in SREC format from the storage domain, and write the SREC file line by line into the domain memory. Specifically, it decodes each line in the SREC file, and "memcpy" the decoded data into the decoded address. Upon seeing a certain type of SREC line ("SREC_TYPE_9"), the bootloader finalizes the hash, extends the hash to the corresponding TPM PCRs, and jumps to the entry point of the domain. The bootloader does not need to know the size of the domain image, because the Resource Manager delegates exactly the amount of quota equal to the size of the domain image divided by the size of the storage data mailbox message. The SREC loading logic is implemented in octopos/arch/sec_hw/bootloader_interface/bootloader_fs.c.

## 7.5 Bootloader Memory Sanity

As we discuss in §3.1, each domain's Microblaze processor is wired to at least two BRAM blocks, where one of them is used as the ROM for bootloader memory, and the others are used as the RAMs for domain memory.

BRAM block are limited to certain sizes (powers of 2 KB). Multiple RAMs are useful to construct a total memory space that is not limited by the size of a single BRAM block. For simplicity, we assume that only one BRAM block is used for the RAM.

The ROM of a domain is prepopulated with the domain's bootloader, and the content is embedded into the programmable logic bitstream. At the beginning, they are not read-only, but the bootloaders immediately burn their respective ROM fuses right after they launch, irreversibly making the memory read-only ( §3.1, §15.4.1)

Because the bootloader cannot write any data to the ROM memory region, it cannot even modify its own variables. Therefore, the bootloader must be free of any memory write, except for stack and heap, which are treated differently. One disallowed memory region is the BSS section. The block starting symbol (BSS) stores uninitialized global variables. Because this region falls into the ROM memory region, the bootloader cannot write to it. Therefore, BSS is not allowed. All global variables must be initialized (so that they are stored in the data region) and must not be modified. OctopOS bootloader developers must be aware of this limitation and review all the code, including any linked libraries, to ensure that the bootloader does not write to the ROM memory region, except for stack and heap.

The stack and heap are treated differently. The bootloader uses a small region from the RAM (another BRAM block that is not locked, which will be used as the RAM for domain memory) as the stack and heap. We call this the "Common Stack and Heap region". At the end of the boot process, the bootloader must erase the entire Common Stack and Heap region to ensure that the domain memory is free of any data residue from the bootloader.

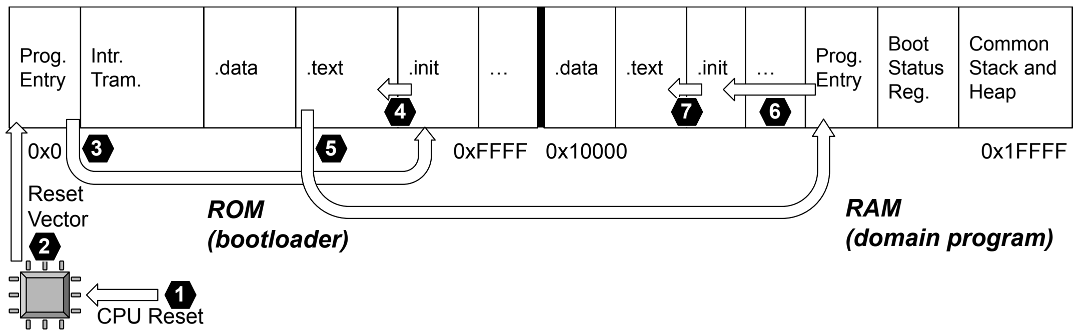


Fig. 9. *CPU reset and binary execution flow within a domain. This figure shares the same notations as Figure 8*

## 8 RESET VECTOR

As we discuss in §3.1 and §7.5, each domain has a ROM and at least one RAM. All memories, although physically separated, are logically mapped to the same address space.

Microblaze processors are hard-wired to load the program entry at a fixed address (a.k.a. the reset vector, in our case, 0x0). Therefore, the bootloader must be placed at address 0x0, which is located in the ROM region. The processor automatically loads the bootloader from the ROM upon reset.

However, the domain program is loaded to an offset far from the beginning of the address space (because the base address of RAM has to at least skip the *size* of ROM), which is not compliant with the Microblaze address convention. We modify the linker script of the domain program to place the program entry point of the domain program at an address known to the bootloader. Therefore, toward the end of the bootloader's execution, it jumps to the correct entry point of the domain program.

We illustrate this process in Figure 9. First, an external reset signal resets the Microblaze processor. Second, the processor jumps to the reset vector (0x0). Third, the bootloader's program entry jumps to the *.init* section. Fourth, it subsequently jumps to the *.text* section and executes the bootloader's main logic, which includes loading the domain program from the storage domain into the RAM region, hashing, and extending the hash to the TPM PCRs. Fifth, the bootloader jumps to the program entry point of the domain program, which is located at the address known to the bootloader. Sixth and finally, the domain program executes its main logic.

### 8.1 Interrupt Trampoline

Similar to the reset vector, the Microblaze processor jumps to a fixed address (a.k.a. the interrupt vector, in our case, 0x10) when an interrupt occurs. Sw_exception and hw_exception are handled similarly at another fixed address, in this section, we only discuss the interrupt handling because exceptions are not enabled in our system.

The hardware expects that the interrupt handlers (or instructions that will eventually jump to the handlers) are placed at fixed addresses. The bootloader works out of the box because its linker places the bootloader at the ROM, the beginning of the address space starting at 0x0. Therefore, the bootloader's interrupt handlers are placed at the correct addresses.

However, similar to the reset vector, the domain program's interrupt handlers are placed at an offset far from the beginning of the address space. Therefore, the Microblaze processor always jumps to the bootloader's interrupt handlers when an interrupt occurs.

To solve this problem, we implement a trampoline. We place *Trampoline Handlers* at the memory addresses where the Microblaze processor would expect a interrupt handler. We designate a status register (Boot_Status_Reg) at a known address in the RAM region, which can be modified by the bootloader. When the bootloader starts, it sets the status to 0x0, and when it finishes, it sets the status to 0x1.

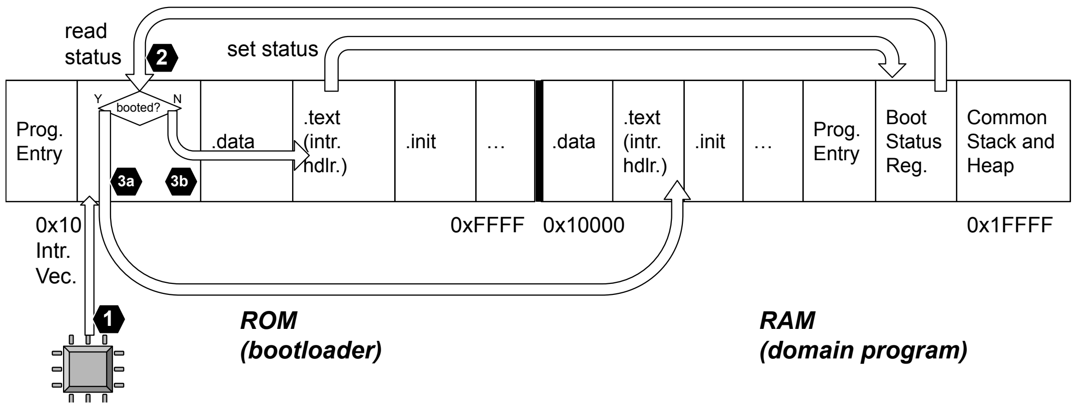


Fig. 10. *Interrupt trampoline delivers an interrupt to the domain program. This figure shares the same notations as Figure 8*

.

As shown in Figure 10, when an interrupt occurs , the Microblaze processor jumps to the *Trampoline Handlers*(Step 1). In Step 2, the *Trampoline Handlers* reads the status of the Boot_Status_Reg. If the status is 0x1, it means the bootloader has finished, and the *Trampoline Handlers* jumps to the domain's interrupt handler (Step 3a). If the status is 0x0, it means the bootloader has not finished. The execution flow switches to the bootloader's interrupt handler (Step 3b).

The code snippets below describe the *Trampoline Handlers*. This code is compiled into the bootloader, and the bootloader's linker script places it at the default address (i.e., the interrupt vector). The domain program's handler code remains unchanged, but the domain's linker script must place it at the address known to the *Trampoline Handlers*.

```
void __interrupt_handler(void)
{
    /* the compiler saves the volatiles and MSR */
    unsigned int *boot_status_reg = (unsigned int *)BOOT_RESET_REG;
    if (*boot_status_reg) {
        /* the bootloader has finished */
        /* jump to the domain's interrupt handler */
        (*KNOWN_INTERRUPT_HANDLER_ADDRESS)();
    } else {
        /* the bootloader has not finished */
        /* jump to the bootloader's interrupt handler */
        InterruptVectorTable[0].Handler(...);
    }
    /* the compiler restores the volatiles and MSR */
}
```

It is true that both the bootloader and the domain program can modify the BOOT_RESET_REG because it is in the RAM region. We do not consider this as a security issue since the worst thing the domain can do is to cause the bootloader interrupt handler to be called instead of the domain program. If such a thing happens, the bootloader memory state is clean (as the stack and heap are erased before loading the domain program) and therefore cannot leak any information.

## 8.2 Image Size Optimization

To optimize the boot time, we need to minimize the size of the bootloaders and the domain images. All Microblaze binaries are built with the "-Os" flag, which optimizes for size.

We use Petalinux to build the untrusted domain's Linux kernel and root file system. To minimize the size of the Linux kernel, we disable the following unnecessary drivers in the kernel configuration ("petalinux-config -c kernel"): *PCI bus; MTD; Serial ATA; SPI; GPIO; Multimedia; Sound; USB; LED; Virtio; Staging driver; extcon; Industrial IO; Reliability; Android; FPGA*.

To minimize the size of the untrusted domain's root file system, we omit the following unnecessary packages ("petalinux-config -c rootfs"): *fpga management; havged; mtd-utils; pciutils; run-postinsts; udev-extraconf;packagegroup-core-ssh-dropbear; tcf-agent; watchdog-init; hellopm; hwcodecs; debug-tweaks*.

## 9 OCTOPOS TEE RUNTIME

The OctopOS TEE runtime is the software that runs in the TEE domains. It is responsible for the following tasks:

- Initializing the TEE domain's hardware, such as the mailboxes and the interrupt controller.
- Communicating with the Resource Manager to inform its status, so that the Resource Manager can schedule the domains.
- Launching applications in the TEE domain.
- Providing APIs to request and yield the I/O services provided by the I/O domains.
- Providing APIs to communicate with other TEE domains.

The TEE runtime is provided to facilitate the use of the TEE domain, and it is not mandatory. An application can provide its own code to perform the tasks listed above. If the application chooses to use the TEE runtime, it must trust the TEE runtime to perform the tasks correctly because the TEE runtime runs in the same domain as the application.

The TEE runtime provides APIs to use the I/O services. The requests are sent to the Resource Manager through a *fixed* FIFO, and the Resource Manager will delegate the I/O domain mailbox to the TEE domain. We borrow the term "syscall" from the traditional operating system to describe the I/O requests sent to the Resource Manager.

The Resource Manager will return the response to the TEE domain through the TEE domain's mailbox. Each TEE domain has one delegatable mailbox, and the TEE runtime will use the mailbox to receive the response from the Resource Manager. If needed, this mailbox can be delegated to another TEE domain to allow the two TEE domains to communicate with each other, without the need for the Resource Manager.

The TEE runtime performs due diligence to ensure that the Resource Manager correctly handles the requests. For example, the TEE runtime will read the status register of the mailbox to ensure that the requested number of quota and time limits are correctly set. If the TEE runtime detects that the Resource Manager is not behaving correctly, it will terminate the application.

The full list of TEE runtime APIs can be found in octopos/runtime/runtime.c.

## 9.1 Storage APIs

The TEE runtime provides block I/O APIs to access the storage domain. Before using the I/O APIs, an application must call "request_secure_storage_access()" to request access to the storage domain. This API call will request the Resource Manager to create a secure storage partition for the application. The Resource Manager will then delegate the storage mailbox to the TEE domain. If there is another active delegation, the Resource Manager will wait for the current delegation to finish. This is common for the storage domain to be busy because the untrusted domain's Linux rootfs is hosted on the storage domain, and the Linux kernel will access the storage domain frequently.

The read and write storage block APIs mimic the POSIX block I/O API, providing the following arguments for the read and write access: “start_block” specify the block number to be read/written (local to the domain’s partition); “num_blocks” specify the number of blocks to be read/written. The block size of OctopOS storage is 512 bytes, the same as our mailbox message size.

Another set of fine-grained APIs is provided to access an offset within a block, which is useful for applications that need to read/write data that is not aligned to the block size. The storage APIs can be found in octopos/runtime/storage_client.c.

We implement the storage driver of our Linux Compatibility Layer on top of these storage APIs. Our block device driver accesses the storage domain mailbox using the same code as in the TEE runtime. §13 describes the details of our Linux Compatibility Layer.

## 9.2 Network Communication APIs

The TEE runtime provides APIs to use the network domain. An application must call “request_network_access()” to request access to the network domain. This API call will request the Resource Manager to switch the network domain’s arbiter to the TEE domain. The API arguments specify how many quota and time limits the application needs. The TEE runtime implements a simple IP stack, and writes the IP packets to the network domain’s mailbox. The network APIs can be found in octopos/runtime/network_client.c.

## 10 OCTOPOS I/O PROTOCOL

The fundamental concept in the protocol is that each I/O service views its underlying I/O device as one or more resources. For example, in the case of the storage service, each partition is a resource, and in the case of the network, port is the resource. The protocol is a set of operations that the service implements in order to allow the OS to allocate these resources for the apps to use in a secure fashion.

We implement our keyboard, serial out, storage, and network services using the I/O protocol. We expect other services to be implemented in a similar fashion. Table 1 below shows the operations that the I/O service needs to implement.

| **Operation** | **Description** | **Rules** |
|---|---|---|
| *IO_OP_QUERY_ALL_RESOURCES* | Query all available resources | Nop if a resource is bound |
| *IO_OP_CREATE_RESOURCE* | Create a new resource | Nop if not authenticated |
| *IO_OP_BIND_RESOURCE* | Bind one or more resources | Nop if any resources is already bound; No unbound until a service reset |
| *IO_OP_QUERY_STATE* | Query the state of the I/O service | Nop if a resource is bound; Nop if not authenticated |
| *IO_OP_AUTHENTICATE* | Authenticate with the service | Nop if a resource is bound; Nop if not authenticated |
| *IO_OP_DEAUTHENTICATE* | Deauthenticate with the service | Nop if a resource is bound; Nop if not authenticated |
| *IO_OP_DESTROY_RESOURCE* | Destroy a resource | Nop if no resource is bound; Nop if not authenticated |
| *IO_OP_SEND_DATA* | Send data to the service | Nop if no resource is bound; Nop if not authenticated |
| *IO_OP_RECEIVE_DATA* | Receive data from the service. | Nop if no resource is bound; Nop if not authenticated |

Table 1. I/O Service Operations

We omit the details of the protocol implementation. The interested reader can find the full OctopOS I/O protocol in octopos/docs/io_protocol.rst.

## 11 OCTOPOS STORAGE DOMAIN

### 11.1 Storage Media

We use a statically partitioned portion of the main memory as the storage media of the storage domain. We implement a customized RAM-based file system on top of the partitioned main memory.

This is only for the prototype and in the future, we plan to use a dedicated storage device (e.g., an SSD) for the storage domain. Alternatively, the development board provides a QSPI flash that can be directly mapped to the programmable logic. To use the QSPI flash memory, we need to implement a driver to read and write blocks, and to manage the erase cycle of the flash (e.g., using a wear-leveling algorithm). The QSPI code is no longer maintained, but can be found in octopos/arch/sec_hw/qspi/.

As yet another alternative, we can use a PMOD device (e.g., a microSD card) as the storage media. Indeed, we have implemented drivers for these two alternative designs and proved that they work. However, we have not included them in our final prototype because their performance is not as good as the partitioned main memory.

Regardless of the storage media, the storage domain creates a simple partitioning on top of the storage media's file system (e.g., a RAM-based file system), or raw blocks (e.g., a QSPI flash memory). Each partition is identified by a unique partition ID. The block numbers referred to by a domain are local to the partition owned by the domain.

### 11.2 Storage Protocol

We follow the OctopOS I/O protocol (§11) in implementing the storage domain . In the case of the storage domain, the resource is the partition, and the data is the block data. The storage domain provides the following APIs to a domain that is using the storage:

- *IO_OP_QUERY_ALL_RESOURCES*: query all the partitions that are available.
- *IO_OP_CREATE_RESOURCE*: create a new partition.
- *IO_OP_BIND_RESOURCE*: bind to a partition.
- *IO_OP_QUERY_STATE*: query the state of a partition.
- *IO_OP_AUTHENTICATE*: authenticate to a partition.
- *IO_OP_SEND_DATA*: send data to a partition.
- *IO_OP_RECEIVE_DATA*: receive data from a partition.
- *IO_OP_DEAUTHENTICATE*: deauthenticate from a partition.
- *IO_OP_DESTROY_RESOURCE*: destroy a partition.

Readers are encouraged to read octopos/storage/storage.c for the details of the storage domain's implementation.

## 12 OCTOPOS NETWORK DOMAIN

We implement the network domain using the OctopOS I/O protocol. The network domain directly communicates with the network interface card (NIC) IP core in the programmable logic, which is connected to the Ethernet port on the SFP connector on the development board. The software stack for the network domain is built on top of axieithernet version 5.10 [11].

The network domain implements a send IP packet interface, which delivers any IP packet submitted by the TEE domain or the untrusted domain. The network domain also implements a receive IP packet interface, which delivers any IP packet received from the NIC to the TEE domain or the untrusted domain. The arbiter programmed by the Resource Manager decides which domain receives the IP packet.

The network domain exposes IO_OP_BIND_RESOURCE, and IO_OP_QUERY_STATE. Unless a port is bound, the network will not send out or deliver any IP packets. Unlike the storage domain, the network data are not in the form of blocks, but in the form of IP packets. Although we can enforce quota limit on the number of IP packets, the uncertain nature of the network traffic makes it difficult to predict the quota needed. Therefore, in practical use, we always set the quota to the maximum value and focus on the time limit. Once a port is bound, the network domain will send out any IP packets received from the connected domain, until the time limit expires.

## 13 LINUX COMPATIBILITY LAYER

The OctopOS Linux Compatibility Layer is a set of Linux kernel modules that provide a Linux-compatible interface to use the I/O services implemented in the Split-Trust's hardware. The compatibility layer effectively allows Linux to be the runtime of any TEE domain, although currently, the TEE domain's processor (no MMU) is not configured to support Linux .

We implement all mailboxes in the Linux device tree. We implement the driver for our mailbox's control interface, and port the LogiCORE IP Mailbox driver to the Linux kernel. At the Linux kernel boot time, all mailboxes that are connected to the untrusted domain are probed and initialized.

We implement the storage service as a block device, and it is mounted as the root file system. On Block I/O (bio) request, the driver checks if the delegation is valid. If not, the driver sends a request to the Resource Manager to request storage mailbox delegation. If the delegation is valid, the driver sends the block read or write request to the storage mailbox.

We implement the network service support using Xilinx's Linux kernel driver for the AXI Ethernet IP core. The driver understands how to communicate with the IP core through the DMA interface. The arbiter is transparent to the driver.

## 14 EMULATOR

The emulator is a software emulation of our Split-Trust hardware. It is implemented in C and is used to test the OctopOS software stack. The emulator consists of the following components:

- *All the OctopOS hardware domains*: The emulator implements all the domains of the Split-Trust hardware, including the I/O domain, the TEE domains the storage domain, and the network domain.
- *Mailbox emulation*: The emulator implements a mailbox interface to emulate the communication between the domains.
- *Untrusted domain*: The emulator implements an untrusted domain that runs Linux.
- *TPM emulation*: The emulator uses the same TPM emulator as the OctopOS hardware.
- *Experimental I/O domains*: The emulator implements experimental I/O domains that are not implemented in the OctopOS hardware, such as the Bluetooth domain.

To install the emulator, fetch all its submodules using *git submodule update –init –recursive*. Then, download a rootfs and copy to arch/umode/untrusted_linux/. We recommend using *CentOS6.x-AMD64-root_fs*. Then, follow the "Setting up the TPM dependencies" instructions under docs/tpm.rst to download and build all TPM modules. Then, run *make umode* to build the emulator. Finally, launch the emulator using *arch/umode/emulator/emulator.sh path_to_OctopOS*.

## 15 FORMAL VERIFICATION

To ensure *limited yet irrevocable* delegation and other security features of the Split-Trust hardware, we distill a set of properties that the mailbox, reset guard, arbiter and ROM fuse should satisfy, and use *SymbiYosys* to prove the properties.

We present all formal theorems, and snippets of key verification logic. Readers are encouraged to read and run the full proof files in octopos_hardware/formal_verification/. We use Hoare logic solely to present the theorems that involve state transitions in a concise manner, and the actual proof is done by SymbiYosys's Satisfiability Modulo Theories Bounded Model-Checking (smtbmc) engine. The theorem or lemma discussed in this section are proved in their corresponding subfolders. To run a proof, simply *cd* into the subfolder and run *sby -f filename.sby*.

## 15.1 Formal Properties of Mailbox

We use Domain and a number to denote the domain that is connected to a specific input of the mailbox. For example, domain0 is the domain that is connected to the input ID0 of the mailbox.

The theorems that we prove for mailbox are listed in Table 2, and descibed in detail in the following subsections.

| Property | Proved theorems |
|---|---|
| Exclusive access | 5.a Domains w/o exclusive access to mailbox cannot change which domain has exclusive access, nor the remaining quota. |
| | 5.f If a domain does not yield its exclusive access, its exclusive access is guaranteed as long as the quota has not expired. |
| | 5.c The domain with exclusive access to the mailbox can correctly read or write from/to the queue. |
| | 5.d The domains w/o exclusive access to the mailbox cannot read/write to the queue. |
| Limited delegation | 3.c When given exclusive access, a domain cannot use the mailbox more than its delegated quota. |
| | 3.b When the quota delegated to a domain expires, the domain loses exclusive access. |
| Excl. access verif. | 4.b The domain with exclusive access can correctly verify its exclusive access and remaining quota. |
| | 4.a The domain on fixed end of mailbox can correctly verify domain with exclusive access on the other end and remaining quota. |
| Default exclusive access | 1 After reset, the resource manager domain has exclusive access by default. |
| | 2 The resource manager domain does not lose its exclusive access unless it delegates it. |
| | 3.a When a domain loses exclusive access (yield/expiration), the exclusive access will be given to the resource manager domain. |
| Confidentiality | 5.b Domains w/o excl. access cannot use mailbox's verif. interface to find out which domain has excl. access and remain. quota. |
| | 5.e Upon delegation/yield/expiration, the data in the queue is wiped. |

Table 2. *Theorems we prove for our mailbox. Proving some of these require proving lemmas not listed here.*

#### *15.1.1 Mailbox Theorem 1.* After a reset, domain0 always becomes the owner of the Mailbox.

$$\{owner\ id = X\} reset\ signal \{owner\ id = ID0\} \tag{1}$$

In Formula 1, X represents an arbitrary input ID. As shown in the verification logic below, we prove that after an active low reset signal becomes active, the 8-bit Verilog wire array "owner_id" becomes the ID0 (which represents the Resource Manager). We save the value of the reset signal in the previous clock cycle in reset_latched, so that we can assert that the owner_id is ID0 when the reset signal is active in the previous clock cycle.

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (!aresetn);
    reset_latched <= !aresetn;
    if (aresetn) begin
        prev_owner_id <= owner_id;
        if( (reset_latched) ) begin
                assert(owner_id == `ID0);
        end
    end
    init <= 0;
end
```

#### 15.1.2 *Mailbox Theorem 2.* Domain0 never loses the ownership of the mailbox (Formula 2) unless it delegates the mailbox to another domain (Formula 3).

$$\{owner\ id = ID0\}skip\{owner\ id = ID0\} \tag{2}$$

$$\{owner\ id = ID0\}no\ delegation\{owner\ id = ID0\} \tag{3}$$

Formula 2 is a trivial property because a register or a wire remains stable in a circuit without external changes. As shown in the logic below, we prove Formula 3, that if domain0 does not delegate the ownership of the Mailbox to another domain, (i.e., write_req_0 remains 0), the owner_id remains ID0.

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (!aresetn);
    assume(write_req_0 == 0);
    if (aresetn) assert (owner_id == `ID0);
    init <= 0;
end
```

#### 15.1.3 *Mailbox Theorem 3.a.* When a domain (other than domain0) loses ownership of the mailbox, the next owner (owner_id) is always domain0.

$$\{owner\ id = IDX, IDX \neq ID0\}ownership\ change\{owner\ id = ID0\} \tag{4}$$

This theorem does not care what causes the ownership change, making the theorem more general to cover all possible cases, such as voluntary yield, quota or time exhaustion, and unlikely but possible hardware bugs. As shown in the logic below, we prove Formula 4,that ownership is changed to ID0 in the clock cycle after a domain loses ownership of the Mailbox. Readers may want to relax this theorem to allow the ownership to be delegated to another domain (nested delegation, see Section 2.5).

```
reg init = 1;
always @(posedge clk) begin
  if (init) assume (!aresetn);
    if (aresetn) begin
        prev_owner_id <= owner_id;
        if ((owner_id != prev_owner_id) && (prev_owner_id != `ID0)) begin
            assert(owner_id == `ID0);
    ...
  init <= 0;
end
```

#### 15.1.4 *Mailbox Theorem 3.b.* When the remaining time limit or quota limit becomes zero, the owner domain loses the ownership of the mailbox, and according to Theorem 3.a, the next owner is always domain0.

$$\{owner\ id = IDX, IDX \neq ID0\}time\ limit \leq 1,\ clock\ tick\{owner\ id = ID0\} \tag{5}$$

$$\{owner\ id = IDX, IDX \neq ID0\}quota\ limit \leq 1,\ data\ access\{owner\ id = ID0\} \tag{6}$$

As shown in the logic below, we prove that when either the time limit or quota limit becomes zero, the owner_id becomes ID0. Limit_expired and time_expired are registers to be updated (in Verilog, the operator is <=) on the edge of a clock cycle, and therefore, the if condition, "time_expired || limit_expired", means that the quota was zero in the previous clock cycle and the ownership must have already expired.

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (!aresetn);
    if (aresetn) begin
        limit_expired <= (limit == 12'b0) && (owner_id != `ID0);
        time_expired <= (time_out == 12'b0) && (owner_id != `ID0);
        if (time_expired || limit_expired) assert (owner_id == `ID0);
    ...
    init <= 0;
end
```

### 15.1.5 *Mailbox Theorem 3.c.*

Domains cannot overuse their time limit and quota limit more than the initial limits that domain0 delegates to them. Since we already proved Theorem 3.b, we only need to prove that the data and time quota are correctly updated. We prove this theorem by proving two lemmas: Lemma 1 and Lemma 2.

### 15.1.6 *Mailbox Theorem 3.c Lemma 1.*

Every mailbox data access correctly updates the remaining data quota.

$$\{quota\ limit\} if\ quota\ limit\ > 1, data\ access \{quota\ limit = quota\ limit - 1\} \quad (7)$$

In the following logic, the condition "owner_id == prev_owner_id" and "prev_limit>0" means the mailbox usage did not result in the ownership change, and the condition "latched_limit_req == 1" means a read has happened in the previous clock cycle. The condition "prev_owner_id != ID0" excludes domain0, which has an infinite quota. Then the correct behavior in this situation is to decrease the value of the quota by one.

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (!aresetn);
    if (aresetn) begin
        ...
        assume((limit != `INFINITY));
        if ((owner_id == prev_owner_id) && (prev_owner_id != `ID0) &&
        (latched_limit_req == 1'b1 ) && (prev_limit >0)) begin
            assert(limit == prev_limit - 1);
        end
        ...
    init <= 0;
end
```

### 15.1.7 *Mailbox Theorem 3.c Lemma 2.*

Upon every clock tick, the mailbox correctly updates the remaining time limit.

$$\{time\ limit\} if\ time\ limit\ > 1, clock\ tick \{time\ limit = time\ limit - 1\} \quad (8)$$

The following logic is similar to the previous one, except we no longer need to check the condition "latched_limit_req == 1", and we assert that the time limit is decreased by one.

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (!aresetn);
    if (aresetn) begin
        ...
        if ((owner_id == prev_owner_id) && (prev_owner_id != `ID0)
       && (latched_time_req == 1'b1 ) && (prev_timer >0)) begin
            assert(timer == prev_timer - 1);
       end
    ...
    init <= 0;
end
```

*15.1.8 Mailbox Theorem 4.a.* The domain connected to the fixed input can always read the correct status register of the mailbox. This is to facilitate the domain that owns the mailbox to read the current owner and limits of the mailbox.

This is a trivial property because the state register value propagates to the fixed domain's read register independent of other signals. The solver proves it using the simple logic below.

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (!aresetn);
    if (aresetn) assert(read_state_fixed == state_reg);
    init <= 0;
end
```

*15.1.9 Mailbox Theorem 4.b.* The domain that currently owns the mailbox can always read the correct status register of the mailbox. This is to facilitate the domain that owns the mailbox to verify its ownership and limits of the mailbox.

Similar to Theorem 4.a, we prove this theorem by proving the following trivial property.

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (!aresetn);
    if (aresetn) begin
        case(owner_id)
            `ID0: assert(read_state_0 == state_reg);
            `ID1: assert(read_state_1 == state_reg);
            ...
    end
    init <= 0;
end
```

*15.1.10 Mailbox Theorem 5.a (Lemma 3).* When a domain owns the mailbox, other domains (including domain0) cannot change the ownership of the mailbox or the remaining quota and time limits. The only exception is that the owner can voluntarily give up the ownership of the mailbox.

As shown in Formula 9, we capture the following assumptions: (1) The mailbox is owned by a domain. (2) The owner domain does not yield ownership of the mailbox. (3) The owner domain does not transfer data using the mailbox. (4) The clock tick does not happen. (5) All other signals are subject to change.

We capture the following properties: (1) The mailbox is still owned by the same domain. (2) The remaining quota and time limits are unchanged.

$$\begin{aligned}\{owner\ id = IDX, IDX \neq ID0,\ time\ limit,\ quota\ limit\} \\ skip\ clock\ tick,\ skip\ data\ access,\ skip\ yield \\ \{owner\ id = IDX,\ time\ limit,\ quota\ limit\}\end{aligned} \tag{9}$$

The following logic captures the above assumptions and properties. In the beginning, we delegate the ownership of the mailbox to domain 2. The counter is used to orchestrate the initialization and verification phases.

A specific domain number is required in the verification code, but it does not matter which domain number we choose. A Split-Trust developer may choose to enumerate all possible domain numbers, and verify this theorem for each possible domain owner.

```
reg init = 1;
reg [3:0] counter;
always @(posedge clk) begin
    ...
    assume(write_2_owner_id != 8'hFF);
    if (init) assume (!aresetn);
    prev_state_reg <= state_reg;
    if (counter == 2) begin
        write_req_0 <= 1;
        write_state_0 <= 32'h020FF0FF;
    end else begin
        write_req_0 <= 0;
    end
    if (aresetn) begin
        counter <= counter + 1;
        if(counter > 4) begin
            assert (owner_id == `ID2);
            assert (prev_state_reg == state_reg);
        end
    end else begin
        counter <= 0;
    end
    init <= 0;
end
```

*15.1.11 Mailbox Theorem 5.b.* When a domain owns the mailbox, other domains (including domain0, but not including the fixed domain) cannot read the current owner and limit information.

We prove this theorem using the following logic.

```
always @(posedge clk) begin
```

```
    if (init) assume (!aresetn);
    if (aresetn) begin
    case(owner_id)
        `ID0: begin
                assert(read_state_1 == `NO_ACCESS);
                ...
        end
        ...
    endcase
    end
    init <= 0;
end
```

*15.1.12 Mailbox Theorem 5.c.* When a domain owns the mailbox, it can correctly transfer data to/from the FIFO (the wrapped LogiCORE IP Mailbox). The following logic asserts the write and read signals of the internal FIFO carry the exactly same data as the signals of the mailbox exposed to the owner domain. “S0_data0_AXI_WDATA” is the write signal and “S0_data0_AXI_RDATA” is the read signal exposed to the outside world, and “S0_data_AXI_WDATA” and “S0_data_AXI_RDATA” are the internal signals.

```
always @(posedge clk) begin
    if (init) assume (!aresetn);
    if (aresetn) begin
    case(owner_id)
        `ID0: begin
            assert( S0_data_AXI_WDATA == S0_data0_AXI_WDATA);
            assert( S0_data_AXI_RDATA == S0_data0_AXI_RDATA);
        end
        ...
    endcase
    end
    init <= 0;
end
```

*15.1.13 Mailbox Theorem 5.d.* When a domain owns the mailbox, other domains cannot read or write to/from the FIFO. As shown in the logic below, we assert that the external signals are always 0 for other domains.

```
    always @(posedge clk) begin
        if (init) assume (!aresetn);
        if (aresetn) begin
        case(owner_id)
            `ID0: begin
                assert( 32'b0 == S0_data0_AXI_WDATA);
                assert( 32'b0 == S0_data0_AXI_RDATA);
            end
            ...
        endcase
        end
        init <= 0;
    end
```

*15.1.14 Mailbox Theorem 5.e.* When the owner of the Mailbox changes, we reset the FIFO. This property ensures that when the ownership changes, no stale data is left in the FIFO. The following logic asserts that the internal reset signal is asserted when the ownership changes.

$$\{internal\ mailbox\ resetn = 0\}ownership\ change\{internal\ mailbox\ resetn = 1\} \tag{10}$$

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (!aresetn);
    if (aresetn) begin
        prev_owner_id <= owner_id;
        if( owner_id != prev_owner_id ) assert (!internal_mailbox_resetn);
        end
    end else begin
    prev_owner_id <= 0;
    end
    init <= 0;
end
```

*15.1.15 Mailbox Theorem 5.f.* Lemma 1 and Lemma 2 already prove that quota and time limit are correctly updated at the time of data accesses and clock ticks. This theorem further proves without any data access and clock tick, the quota and time limit will not change by any other signals.

*15.1.16 Mailbox Theorem 5.f Lemma 4.* Lemma 4 will prove that for any domain other than domain0, the quota limit will remain unchanged without data access.

$$\{quota\ limit\}skip\ data\ access\{quota\ limit\} \tag{11}$$

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (!aresetn);
    reset_latched <= !aresetn;
    ...
    prev_limit <= limit;
    prev_owner_id <= owner_id;
    if( (!reset_latched) && (owner_id == prev_owner_id) ) assert(limit == prev
    ...
    init <= 0;
end
```

*15.1.17 Mailbox Theorem 5.f Lemma 5.* Lemma 5 will prove that for any domain other than domain0, the time limit will remain unchanged without a clock tick.

$$\{quota\ limit\}skip\ clock\ tick\{quota\ limit\} \tag{12}$$

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (!aresetn);
    reset_latched <= !aresetn;
    ...
    prev_timer <= timer;
    prev_owner_id <= owner_id;
    if( (!reset_latched) && (owner_id == prev_owner_id) ) begin
        assert(timer == prev_timer);
    end
    ...
    init <= 0;
end
```

## 15.2 Formal Properties of Reset Guard

Reset Guard is a hardware module that prevents a domain from being reset by the Resource Manager, if any one of the mailboxes connected to the domain is in the middle of a delegation, either providing service to other domains, or using service from another domain. Each domain has its own Reset Guard. Reset Guard exposes one port to the Resource Manager allowing it to initiate a reset, multiple ports to the domain's mailboxes, and one output port to the domain's PSRM. The reset signal originating from the Resource Manager is forwarded to the domain's PSRM only when all the conditions satisfy.

| Property | Proved theorems |
|---|---|
| Filtered Reset | 1. The reset signal does not get forwarded if any other domain is using one of this domain's mailboxes. |
| | 2. The reset signal does not get forwarded if this domain is using any of the other domain's mailboxes. |

Table 3. *Theorems we prove for our Reset Guard.*

*15.2.1 Reset Guard Theorem 1.* The reset signal does not get forwarded if any other domain is using one of this domain's mailboxes. In the following logic, we ignore the first reset by assuming that fr (first reset) signal is low. The fr signal is to workaround the random register value issue in the first reset, as mentioned in Section 4.2. The verification logic checks if any of the mailbox busy signals are 1 (mailboxes_busy != 4'b0), and if so, it asserts that the output reset signal is not active (assert(out_reset_n == 1)). Note that the output reset signal is active low.

```
reg  init = 1;
always @(posedge clk) begin
    ...
    assume(fr == 1'b0);
    if (mailboxes_busy != 4'b0)  assert(out_reset_n == 1'b1);
    ...
    init <= 0;
end
```

*15.2.2 Reset Guard Theorem 2.* The reset signal does not get forwarded if this domain is using any of the other domain's mailboxes. Similar to Theorem 1, we ignore the first reset and check if any of the connected mailbox busy signals is 1 (domains_busy != 4'b0), and if so, we assert that the output reset signal is not active.

```
reg init = 1;
always @(posedge clk) begin
    ...
    assume(fr == 1'b0);
    if (domains_busy != 4'b0)  assert(out_reset_n == 1'b1);
    ...
    init <= 0;
end
```

## 15.3 Formal Properties of Arbiter

The arbiter is a hardware module that arbitrates the access to the shared resource among the domains. Section 5 describes the implementation of the arbiter. When the untrusted domain is given access to the I/O domain, the arbiter connects the streams to a DMA engine hard-wired to the memory of the untrusted domain. When the trusted domain is given access to the I/O domain, the arbiter connects the streams to a mailbox.

As Table 4 shows, we prove the exclusive control property and the port mirroring property.

| Property | Proved theorems |
|---|---|
| Exclusive control | 1. The arbiter control interface can arbitrarily change its state between trusted and untrusted. |
| | 2. Nothing other than its control interface can change the arbiter's state. |
| Port mirroring | 1. When the arbiter is connected to a trusted domain, all values of I/O data interface signals are identical to FIFO interface values. |
| | 2. When the arbiter is connected to an untrusted domain, all values of I/O data interface signals are identical to DMA interface values. |

Table 4. *Theorems we prove for our arbiter.*

*15.3.1 Arbiter Theorem 1.* The arbiter control interface can arbitrarily change its state between trusted and untrusted. The following logic asserts a change of the arbiter's state upon writing to the arbiter's control interface. In Formula 13, X is a wildcard that can be either trusted or untrusted. The assertion checks that the arbiter's state is consistent with the control interface.

$$\{switch = X\}write\ STATE\_REG\_UNTRUSTED\ to\ arbiter\ control\{switch = trusted\} \tag{13}$$

```
reg init = 1;
always @(posedge clk) begin
    if (init) assume (resetn == 1'b0);
    if (resetn) begin
        if (write_state_value == `STATE_REG_UNTRUSTED)
    should_be_trusted <= 0;
        else should_be_trusted <= 1;
        assert(should_be_trusted == trusted);
    end else begin
        should_be_trusted <=1;
    end
    init <= 0;
```

```
end
```

*15.3.2 Arbiter Theorem 2.* Nothing other than its control interface can change the arbiter's state. In Formula 14, X is a wildcard that can be either trusted or untrusted. Similar to Arbiter Theorem 1, the following logic asserts that the arbiter's state does not change when the control interface is not written to.

$$\{switch = X\} skip\ write\ to\ arbiter\ control \{switch = X\} \tag{14}$$

```
reg  init = 1;
always @(posedge clk) begin
    if (init) assume (resetn == 1'b0);
    ...
    if (resetn) begin
        prev_trusted <= trusted;
    assume(s_ctrl0_axi_wvalid == 1'b0);
    assert (prev_trusted == trusted);
    end else begin
    prev_trusted <= 1;
    end
    init <= 0;
end
```

*15.3.3 Arbiter Theorem 3.* When the arbiter is connected to a trusted domain, all values of I/O data interface signals are identical to FIFO interface values. In the following logic, we assume that the arbiter is connected to a trusted domain (assume(trusted == 1)), and we check that all values of the incoming I/O data interface signals are identical to the outgoing FIFO interface values.

```
reg  init = 1;
always @(posedge clk) begin
  if (init) assume (resetn == 1'b0);
  if (resetn) begin
    assume(trusted == 1'b1);
    assert(O_axi_txd_aresetn   == T_axi_txd_aresetn  );
    assert(O_axi_txc_aresetn   == T_axi_txc_aresetn  );
    ...
  end else begin
    prev_trusted <= 1;
  end
  init <= 0;
end
```

*15.3.4 Arbiter Theorem 4.* When the arbiter is connected to an untrusted domain, all values of I/O data interface signals are identical to DMA interface values. Similar to Arbiter Theorem 3, we assume that the arbiter is connected to an untrusted domain (assume(trusted == 0)), and we check that all values of the incoming I/O data interface signals and the outgoing DMA interface values are identical.

```
reg  init = 1;
always @(posedge clk) begin
    if (init) assume (resetn == 1'b0);
    if (resetn) begin
        assume(trusted == 1'b0);
        assert(O_axi_txd_aresetn == U_axi_txd_aresetn);
        assert(O_axi_txc_aresetn == U_axi_txc_aresetn);
        ...
    end else begin
        prev_trusted <= 1;
    end
    init <= 0;
end
```

## 15.4 Formal Properties of ROM fuse

ROM fuse is a one-time writable fuse to disable the BRAM memory bus's Write Enable (WE) signal. By disabling the RAM's WE signal we practically convert it to a ROM. The main feature of this hardware is its irreversibility, which means once the fuse is burnt (WE signal disabled), there is no way to enable it back until power-cycling the device.

| Property | Proved theorems |
|---|---|
| Irreversibility | 1. Burning the ROM fuse is irreversible |

Table 5. *Theorems we prove for our ROM fuse.*

*15.4.1 ROM fuse Theorem 1.* Burning the ROM fuse is irreversible. The ROM fuse must maintain its irreversible state even if the reset signal is asserted.

$$\{rom_fuse = burned\}any\ signal\{rom_fuse = burned\} \tag{15}$$

The verification logic goes as follows: On positive edges of the clock, we increment a counter which is initialized to zero. When the counter is one, (i.e., one CLK cycle after the beginning of the test), we assume that the ROM fuse has been locked. Then, we ask the prover to prove that the ROM fuse cannot be unlocked in the following clock cycles no matter how other signals are changing.

```
reg [7:0] init_counter = 0;
always @(posedge clk) begin
  if (init_counter == 1) assume (locked == 1'b1);
  if (init_counter >2) assert(locked == 1'b1);
  if (init_counter < 100) init_counter <= init_counter + 8'b1;
end
```

## REFERENCES

[1] Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, 2016.
[2] LogiCORE IP Mailbox. https://www.xilinx.com/products/intellectual-property/mailbox.html#documentation, 2022.
[3] Vivado Design Suite User Guide. https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug973-vivado-release-notes-install-license.pdf, 2021.
[4] Processor System Reset Module v5.0. https://docs.xilinx.com/v/u/en-US/pg164-proc-sys-reset, 2015.

[5] Trusted Computing Group (TCG). TPM 2.0 Library. https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2019.

[6] Trusted Computing Group (TCG). TCG PC Client Specific TPM Interface Specification (TIS), Specification Version 1.3. https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientTPMInterfaceSpecification_TIS__1-3_27_03212013.pdf, 2013.

[7] IBM. Software TPM Introduction. http://ibmswtpm.sourceforge.net/ibmswtpm2.html, 2021.

[8] tpm2-software community. TPM2 Software Stack. https://tpm2-software.github.io, 2023.

[9] tpm2-software community. TPM2 Access Broker and Resource Management Daemon implementing the TCG spec. https://github.com/tpm2-software/tpm2-abrmd, 2023.

[10] AMD Xilinx. Zynq UltraScale+ FSBL. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842019/Zynq+UltraScale+FSBL, 2022.

[11] axiethernet Documentation. https://xilinx.github.io/embeddedsw.github.io/axiethernet/doc/html/api/index.html, 2022.