(MASTER) The hyplet - Joining a Program and a Microvisor for real time and security

xxx

Abstract

This paper presents the concept of sharing a microvisor address space with a standard Linux program. We add hypervisor awareness to the Linux kernel and execute code in the HYP exception level. We do this through the use of the hyplet. The hyplet is an innovative way to code interrupt service routines under ARM. The hyplet provides many benefits including: high performance, security, run time predictability and an RPC mechanism. The hyplet uses special features of the ARM hypervisor memory architecture. We demonstrate the hyplet implementation using the C programming language on an ARMv8 platform and under the Linux kernel. We provide performance measurements, use cases and security scenarios.

1 Introduction

There are various techniques to achieve real time. One is to use a single operating system that provides real time, as seen in Figure 1.

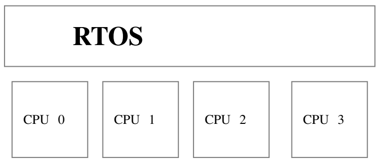

Figure 1: RTOS

Another technique is the microkernel, where the general purpose operating system (GPOS) is preempted by a microkernel. A subclass of the microkernel is the microvisor. A microvisor is an operating system that employs some characteristics of a hypervisor and some characteristics of a microkernel. A typical architecture of a microvisor is depicted in Figure 2.

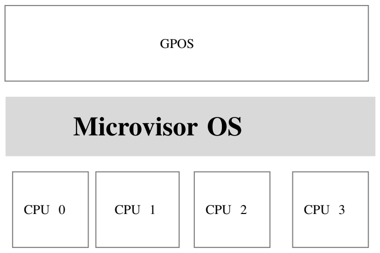

Figure 2: Microvisor

The hyplet, depicted in Figure 3 is a single real-time processing unit shared between a process and a microvisor.

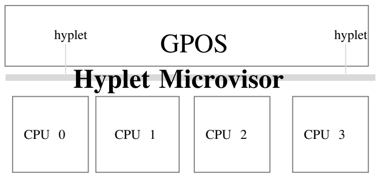

Figure 3: Hyplet Microvisor

The hyplet is a hybrid of a normal user program and a microvisor that offers real time processing and security benefits. We introduce the hyplet for the purpose of interrupt handling in real time and for the purpose of efficient interprocess communication using a powerful mechanism that is similar in nature to remote procedure calls.

There are already multiple microvisor solutions today.

These microvisors often provide security and/or real time features to applications. In a microvisor environment, each VM is encapsulated in its own logical hardware partition. Furthermore, each VM has its own minimal operating system Iqbal *et al.* [2009]. Many microvisors separate among the hypervisor exception level, the kernel and user space exception levels. However, Heiser expressed a difficulty in this approach. According to

Heiser Heiser [2011] pure virtualization is not suitable due to the lack of efficient resources sharing and rough scheduling. Heiser Heiser and Leslie [2010] Varanasi and Heiser [2011] proposed the OKL4 microvisor as a solution to the real time requirements. Kanda et al Kanda *et al.* [2008a] doubted the concept of running two distinct operating systems sharing the same application processor, as this might increase the engineering cost. For instance, supporting a low energy RTOS Kumar and Srivastava [2000] or task scheduling under limited power Jejurikar and Gupta [2002]. Performing such tasks by multiple operating systems raises the project complexity and costs.
Therefore, we present the hyplet ISR as a para-virtualization technique to reduce hardware to user space latency. We will use the term hyp-ISR to distinguish a normal ISR from an ISR in hyplet mode. In addition to the hyplet real time features, we will also demonstrate a new hyplet based RPC system. In this area, Liedtke Liedtke *et al.* [1991] showed that L4 microkernel Inter-process communications can be 20 times faster than other microkernels.

Para-virtualization is a technique in virtualization. Para-virtualization exposes to a virtual machine a software interface the resembles the hardware interface. Para-virtualization goal is to simplify and improve performance in the guest VM. Hartig Härtig *et al.* [1997] demonstrated that para-virtualization reduces the communication overhead between user to kernel services to a few percent overhead. Furthermore, in addition to hyp-ISR we present hypRPC. HypRPC is a reduced RPC mechanism which has a latency of a sub-microsecond on average, and 4 microseconds worst case. Our RPC is a type of hypervisor trap where the user process sends a procedure id to the hypervisor to be executed with high privilege without interrupts in another process address space. We use the term hypRPC for our RPC as a mixture between hypercall and RPC.
The hyplet is based on the concept of a delicate address space separation within a running process. Instead of running multiple operating systems kernels, the hyplet divides the Linux process into two execution modes. Part of the process would execute in an isolated, non-interrupted privileged safe execution environment. The other part of the process would execute in a regular user mode.

To summarize, the hyplet is meant to reduce the latency of:

- hardware interrupt to a user space program
- program to program local communication in user space programs

to sub microsecond order of magnitude. In the taxonomy of virtualization, hypelts are classified as bare metal type 2 hypervisors. A type 2 hypervisor is a hypervisor which is loaded by the host operating system. A type 1 hypervisor is a hypervisor which is loaded by the boot loader, prior to the general operating system. The hyplet is not virtual machines at all, and may execute in hardware that does not have support for interrupts virtualization. The hyplet is meant to be simple to use and adapt to an existing code. The hyplet does not require any modifications to the the boot loader, only to the Linux kernel. As such, we consider the hyplet as an extension to the Linux kernel.
This paper is organized as follows: Section 2 describes ARM architecture features in the hyplet context. Section 3 explain the hyplet in detail. Section 4 is an evaluation. Section 5 demonstrates some use cases. Section 6 is an example code. Section 7 provides an overview of related work. Section 8 is a summary of the expected future work and conclusions.

2 Background

2.1 ARM Permission model usage in the hyplet

ARM has a unique approach to security and privilege levels, that is crucial to the implementation of the hyplet. In ARMv7, ARM introduced the concept of secured and non secured world, through the implementation of TrustZone, and starting from ARMv7a, ARM presents 4 exception (permission) levels.

Exception Level 0 (EL0) refers to user space code. This is analogous to ”ring 3” in x86 platform.

Exception Level 1 (EL1) refers to operating system code. This is analogous to ”ring 0” in x86 platform.

Exception Level 2 (EL2) refers to HYP mode. This is analogous to ”ring -1” or ”real mode” on the x86 platform.

Exception Level 3 (EL3) refers to TrustZone as a special security mode that can monitor the ARM processor and may run a security real time OS. There is no direct analogous modes but related concepts in x86 are Intel's ME or SMM.

Each of the exception levels provide its own state of special purpose registers, and can access these registers of the lower levels but not higher levels. The general purpose registers are shared. Thus, moving to a different exception level on the ARM architecture, does not require the expensive context switch that is associated with the x86 architecture.
ARMv8 architecture dictates that the translation tables of

the different exception levels are to be distinct. Exception level 2 refers to HYP mode. This is analogous to ”ring -1” or ”real mode” on the x86 platform.

3 The hyplet

When there is need to improve the latency of an interrupt event in user space, the trivial approach would be to migrate code to the kernel, or inject a program as the eBPF framework suggests Corbet [2018]. However, kernel programming requires a high level of programming skills, and eBPF, which we describe in the related work section, is restrictive. A different approach would be to trigger a user space event from the interrupt, but this would require an additional context switch. A context switch in most cases is time consuming and not suitable for real-time applications. Therefore, to make sure that the program code and data are always accessible, it is essential to disable evacuation of the program's translation table from the processor. So, we chose to constantly accommodate the code and data in the hypervisor translation registers Penneman *et al.* [2013] as depicted in figure 4. In order to map a user space program, we modified the Linux ARM-KVM, Dall and Nieh [2014] mappings infrastructure to map a user space code with kernel space data.

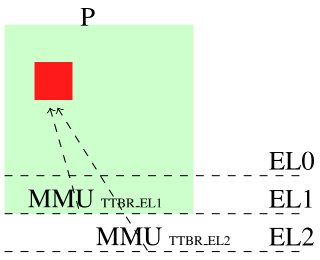

Figure 4: Asymmetric dual view

Figure 4 demonstrates how identical address may be mapped differently in two separate exception levels. TTBR0_EL1 is the register that points to the translation table for the user space program, while TTBR0_EL2 points to the translation table of the hypervisor. The shared section, colored red, is part of of the translation table of EL2 and therefore accessible from EL2. However, when executing in EL2, EL1 data is accessible without premature mapping to EL2. Therefore, the hyplet is a method to install code in a hypervisor that causes privilege elevation, and execution of such code.

3.1 The hyplet security & Privilege escalation in RTOS

Real time systems may eliminate user and kernel mode separation, or move a certain process to higher privileges levels.

Escalating privileges does provide minor performance gains, but exact a price in security. A security bug at higher privilege levels may cause greater damages compared to a bug at the user process level. For example in the XBox Steil [2005] case running games in kernel mode, provided an entry point for breaking the DRM system. The hyplet also escalate privilege levels, from exception level 0 (user mode) or 1 (OS mode)to exception level 2 (hypervisor mode).

Since the hyplet executes in EL2, it has access to EL2 and EL1 special registers. For example the EL2 hyplet has access to level 1 exception vector.

Therefore, it can be argued that the hyplet comes with a security costs. We argue that this is not the case. In most embedded systems and mobile phones no hypervisor is installed. In the case where no hypervisor is installed, exception level 1 (OS) does not have lesser access than the hypervisor mode when only one OS running, as in the hyplet use case. So leveraging a code from EL1 without a hypervisor to EL2 is just as dangerous as doing the same with a hypervisor.

Additionally, it is expected that the hyplet would be a signed code; otherwise, the hypervisor would not execute it.
The hypervisor can maintain a key to verify the signature and ensure that lower privilege level code cannot access the key. This was shown by Resh and Zaidenberg [2013] on Intel platform. In addition, on ARM the Trustzone may be configured to trap illegal access attempts to special registers, and prevent any malicious tampering of these registers.
In order to do reduce security risks associated with the hyplet, we suggest a static binary code analyzer. The code analyzer scans the ELF binary and verifies that there are no references to special purpose registers. We borrowed this idea from eBPF. The code analyzer scans the hyplet opcode, and checks that are no references to any black-listed register. With the exception of the clock register and general purpose registers, any other registers are not allowed. If the hyplets uses libc's APIs, we assume libc does not violate security.
However, since most of the memory is not mapped to the hypervisor, only a non sensitive part of the calling process memory is mapped. The hyplet does not map (and thus have no easy access to) kernel space code or data. Thus the hyplet does not pose a threat of unintentional corrupting kernel's data, or any other user process. Therefore, The risk of a rogue pointer manipulating sen-

sitive EL1 memory is minimized. (Unintentional memory corruption is almost impossible. Intentional memory corruption requires remapping by EL2 and can be trapped by TrustZone).

Last, future architecture of the ARMv8 processor, ARMv8.1 comes with a new extension called VHE Penneman *et al.* [2013] , Virtual Host Extension. With VHE, EL2 has a additional translation table, TTBR1_EL2, that would map the kernel address space, while TTBR0_EL2 will continue to accommodate user space code, similar to TrustZone. This way, it would be possible to execute the hyplet without endangering the hypervisor. VHE hardware is not available at the time of this writing, and as such we are forced to use software measures to protect the hypervisor.

3.2 The hyplet - User Space Interrupt

In Linux and other operating systems, when an interrupt reaches the processor, it triggers a path of code that serves the interrupt. Furthermore, in some cases, the interrupt ends up waking a pending process Bovet and Cesati [2005] as depicted in figure 5.

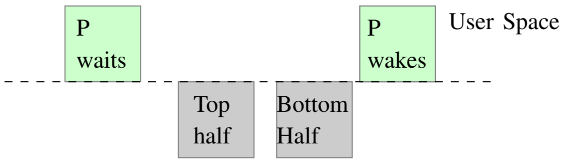

Figure 5: Common Interrupt Flow

Interrupt latency is the time that elapses from the moment an interrupt is generated to the moment it is serviced. In the hyplet, the dual view of the program, eliminates the penalty of a context switch to run user space code, and the top half and the bottom half of an interrupt.
The hyplet reduces the amount of time from the interrupt event to the program. To achieve this, as the interrupt reaches the processor, instead of executing the user program code in EL0 Flur *et al.* [2016] after the ISR (and sometimes after the bottom half), a special procedure of the program is executed in HYP mode, before the kernel's ISR. The hyplet does not changes the processor state when it is in interrupt, thus, once the hyplet is served, the kernel interrupt can be processed as well. This is depicted in figure 6.

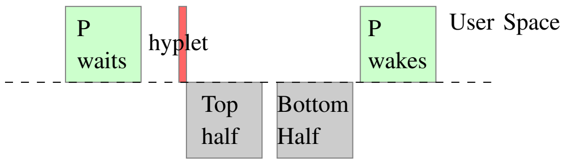

Figure 6: Latency to the hyplet

The hyplet does not require any new threads. Since the hyplet is actually an ISR, it can be triggered in high frequencies. This way we can have a high frequency user space timers in small embedded devices.
Some ARMv8 platforms do not support a complete virtualization of the interrupt controller. Raspberry PI3, for example, does not support VGIC fully. As a consequence, ARM-KVM does not run on a Raspberry PI3. For this reason, the hyplet is a para-virtualization technology. Interrupts are being routed to the hypervisor by calling the HVC (hypervisor call) from the kernel main interrupt routine.

3.3 Hypervisor based RPC

RPC (Remote procedure call) is a type of interprocess communication(IPC), in which parameters are transferred in the form of function arguments. The response is returned as the function return value. The RPC mechanism handles the parsing and handling of parameters and return values. In principal, RPC can be used locally (as a form of IPC) and remotely, over TCP/IP network protocols. In this paper we will only consider the local case.

IPC in real time systems is considered a latency challenge Härtig and Roitzsch [2006]. In many cases IPC is refrained from use because of that challenge. The solution programmers use is to put most the logic in a single process. This technique decreases the complexity but increases the program size and risks.
In multicore computers, one reason for the latency penalty is because it is possible that the receiver is not running when the message is sent. Therefore, the processor needs to switch contexts. HypRPCs are intended to reduce this latency to the sub microsecond on average. HypRPCs act as a temporary address space extension to the sending program.
HypRPCs are easy to use, because there is no need for a synchronization between the receiver and the sender. If the receiving programs exits, then the API immediately returns an error. If the function need to be replaced in real time, there is no need to notify the sending program, but simply replace the function in the hypervisor.

Figure 7 demonstrates the hypRPC state machine. The green circles represent EL0 exception level, while the red circles represent EL2 exception level. Program *P* is a

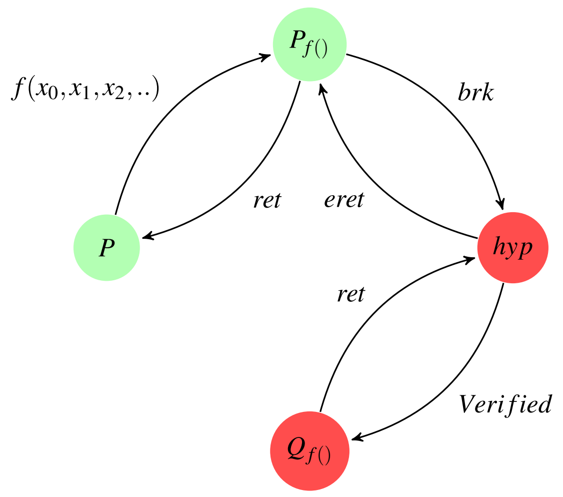

Figure 7: HypRPC State Machine

request program, and Q is a serving program. As program P loads, it defines itself as a hypRPC requesting program. HypRPC program, unlike hypISRs, is a program that when it executes the *brk* instruction, it traps into HYP mode. The reason for that is that user space programs are not permitted to perform the HVC instruction. Function $f()$ in P is a two lines function:

```
foo:
  brk
  ret
```

When P calls $f()$, the first argument is the RPC id, i.e; $x_0 = rpc_{id}$. As the processor execute *brk* , it shifts to HYP mode. Then the hypervisor checks the correctness of the caller P and the availability of Process Q, and if all is ok, it executes $f()$ in EL2.
Thus, program Q can be loaded on any processor, as a hypervised background program, and act as an auxiliary program for any program P. Figure 8 shows a possible scenario in which whenever there is any P that needs a fast RPC, the hypRPC is triggered.

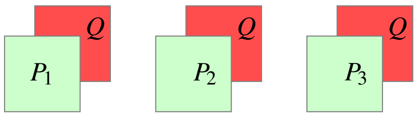

Figure 8: Background Program

As a side benefit, a hyplet RPCs can also be considered as a trusted execution environment. This is because the hyplet can be protected from modifications by keeping it encrypted. This technique is explained in this paper, in protection against reverse engineering section.

3.4 Additional benefits of the hyplet

- **Safety**
  The hypISR provides a safe execution environment for the interrupt. In Linux, if there is a violation while the processor is in kernel mode, the operating system may stop. The kernel stops because it is considered unsafe. So, the kernel prints a log that describes the failure position and reason and the system freezes.

  In the hyplet case, if there is a fault in the hypISR, the microvisor would trigger a violation (for instance, a SEGFAULT). The microvisor would send, through the kernel, a signal to the process containing the hyplet. This is possible because the fault entry of the microvisor handles the error as if it is a user space error. For example; if a divide-by-zero failure happens, the operating system does not crash, but the hyplet'ed program exits with a SIGFPE.
  Another facet of hypISR is sensitive data protection, even from the operating system kernel. We can use the hyplet to securely access data. I/O data may be hidden from EL1 and accessible only in EL2.

- **Temporality**
  We often consider software as dynamic and changeable. For example, interpreters or bash scripts, python and so on. The same cannot be said about device drivers. Interrupts service routines rarely change , i.e. it is not easy to modify a behavior of an interrupt routine in real time (while the device is running) . Consider the case of a robotic system; due to the dynamicity of the robot, the ISR may be needed to change and some services may not be needed at all. In the hyplet case, however, instead of modifying the kernel drivers, we can stop the user space hyplet program, and run a new hyplet with new heuristics.

- **Scalability**
  Though the hyplet is part of a user space program, we cannot easily have more than one hyplet-capable process on one processor. However, we can have several hyplets over the same interrupt in the same program. There is no restriction on the amount of hyplets to use on any interrupt, other than the time it takes the hyplet to complete execution. There is also no restriction on the amount of memory the hyplet uses nor the stack size. This is a benefit, since the Linux kernel stack is restricted to 16 kilobytes, and virtual memory is not abundant in kernel mode.

- **Programming Languages**
  The only requirement is an ELF binary regardless of the programming language.

- **Unification of interrupts**
  The hyplet can be assigned to any interrupt. All types of interrupts may be joined together to the hyplet. For example, the same hyplet may serve an input event from the USB subsystem and the video subsystem.

- **SMP**
  It is possible to register different hyplets over different processors. This means that if an interrupt is assigned to more than one processor, then in order to trap it, we need to bind to each processor a hyplet capable program.

- **Scope of Code Change**
  The hyplet patch does not interfere with the hardcore of the kernel code, neither does it require any modification to any hardware driver. The modifications are in the generic ISR routine, in a program exit, and we introduce a new system call: sys_hyplet. For this reason it is easy to apply it as it does not change the operating system heuristics. Microvisors, such as seL4 and Xvisor are not so easy apply on arbitrary hardware, as they require a modify the boot loader, a fact that makes them impossible to apply in some cases, for example, when the boot loader code is closed. Jailhouse Baryshnikov [2016] and KVM won't even run because virtualization hardware does not suffice (the GIC virtualization is incompatible) in some cases, one of which is raspberry PI3. RT PREEMPT is still not so stable enough. For instance, while preparing this paper we learned how difficult it is to apply the RT PREEMPT patch on raspberry PI3. In Android OS it is undesirable to apply RT PREEMPT because it changes the entire operating system behavior. In short, it is best localize the changes as little as possible.

- **Dynamicity**
  The hyplet is a part of a running program. Like other Unix-like APIs, such as signals or sockets, it may be removed and assigned dynamically while the hyplet'ed process runs. If the process exits the hyplet will remove itself automatically from the hypervisor. Any data section or code section that is unmapped from the process, while the process runs, is automatically unmapped from the hyplet as well, if it is mapped to the hypervisor.

4 Evaluation

We demonstrate that the hyplet is suitable for hard real time systems. We will provide synthetic microbenchmarks, and compare our solution to Normal Debian Linux, RT PREEMPT Linux, seL4 microkernel Klein *et al.* [2008], and Xvisor, all on a Raspberry PI3. Raspberry PI3 main specifications are shown in 1:

Soc	Broadcom BCM2837
CPU	4 cores, ARM Cortex A53, 1.2GHz, (clocked to 700MHz)
RAM	1GB LPDDR2 (900 MHz)
Clock	19.2 Mhz

Table 1: PI3 specifications

We selected Xvisor because Xvisor is a thin microvisor. We chose RT_PREEMPT because it is considered a free open source non-commercial RTOS Linux OS according to Fayyad-Kazan *et al.* [2013] and others. We chose seL4 because it is a hard real time mathematically proven microkernel.

4.1 Latency

In order to evaluate PI3's interrupt latency, we measured the delay from an attached hardware to the start of the hyplet. For this purpose, we connected an Invensense mpu6050 Fitriani *et al.* [2017] to the PI, and configured this IMU to work in i2c protocol. In i2c, for each 8 bits of data, there is an acknowledgement signal, that generates an interrupt to the PI. We wanted to measure the time interval between the moment of the i2c ACK, to the moment the processor runs the main interrupt routine. So, we connected a logic analyzer probe to the SDA of the IMU, and programmed one of the PI's GPIO to trigger a signal in the kernel's the main interrupt routine. This way we could take the time of the IMU ACK signal, and the kernel ISR time. The results were an average of 3.9 μs , maximum 9μs, and the minimum was 1.7μs.
This means that a user space program that expects to be woken for each interrupt will fail. A minimal kernel to user latency, even in seL4, as we show in the paper, is between 2 to 5 microseconds. In other operating systems on this hardware, it is even longer. For this reason, if we want to propagate information in real time to user space, the interrupt must execute in user space, hence the hyplet.

4.2 Timer

We continue the evaluation and construct a timer. A common flow of a timer driven program is as depicted in algorithm 1.

In the hyplet case we modify the flow of the program, by using the new system call sys_hyplet as in algorithm 2.

Algorithm 1: typical timer use

```
Connect to an oscillator;
Configure the device frequency;
Start the device;
while Device is running do
    Wait on Event;
    do something;
end
```

Algorithm 2: typical hyplet use

```
if need connect to an oscillator then
    Configure the device frequency;
    Start the device;
    hyplet_start(the irq , hyplet procedure)
end
... Hyplet function awakes periodically, there no
  need for a special thread ...
```

In table 2 we measured delay latencies of programs. We conducted a delay of 1 ms for 5 minutes, while Linux/seL4 runs. In RT_PREEMPT and Normal Raspbian Linux we used cyclictest Gleixsner, a real time test suit for Linux.

In the hyplet we wrote a simple test that wakes up in each interrupt. In Xvisor, to make the test equal to the hyplet, it terms of which privilege level the code was executed, we wrote a simple test that waits for 1 ms, in HYP mode (not in the VM/guest OS). Nevertheless, it is notable to say that Xvisor was not intended to run real time programs.

ranges in μs	RT_ PRPT	Hyplet	Nrmal	Xvsr	seL4
0	0	99.9477	0	0	0
1	0	0.0523	0	0	0
2-5	0	0.0020	0	0	100
6-10	0	0	47.7	99.9	0
11-15	69	0	49.7	0	0
16-20	28	0	1.6	0	0
21-25	2	0	0.25	0	0
26-30	0.085	0	0.26	0	0
31-35	0.01	0	0.0874	0	0
36-40	0.05	0	0.034	0	0
41-45	0.001	0	0.034	0	0
46-50	0.0003	0	0.05	0	0
51-55	0	0	0.0321	0	0
56-100	0	0	0.18	0	0
101+	0	0	0.0014	.1	0

Table 2: : Latencies Distribution in percentage

In the hyplet case, 99.96% of the samples were bellow 1μs latency , and 100% were bellow 5μs. In RT PREEMPT case, the upper boundary was 47μs, and 14μs on average. In normal Linux the maximum value was 144μs, and the values distribution was higher. Xvisor presents an impressive benchmark where 99.9% samples jitter in less than 8 μs, the rest unfortunately, were nearly 500 us. seL4 is an RTOS.
It is evident that ISR-hyplet can provide hard real time in a regular Linux kernel.

4.3 Fast RPC

We evaluated the round trip of calling a function that returns the time. For Xvisor, Native Linux and RT_PREEMPT we used ptsemtest, which is part of cyclictest. Ptssemtest measures the interprocess latency communication with POSIX mutexes. In seL4 we used ptssetest-like test (sync.c), because ptssemtest is not available in seL4. The hyplet test was a C program that made an RPC to a hyplet'ed process. The reference test is to evaluate to the cost of the calling the function of the hyplet when not in HYP mode.

Name	Avg	Max
Ref	156ns	520ns
Hyplet	520ns	4.2μs
Normal	13μs	56μs
RT PRMT	15 μs	59μs
Xvisor	203μs	7067μs
seL4	8μs	17μs

Table 3: Round Trip RPC

It is evident that the hyplet is the fastest RPC, even in the worst case.
We note that it is essential not to overuse the hypRPC (over 10000/second calls). HypRPCs bypasses the operating system, and hence might delay the processor from moving through a quiescient state Bovet and Cesati [2005]. An overuse of the processor usually trigger kernel watchdogs and might even hog the operating system.

5 Use cases

This section details real use cases for the hyplet and a possible use case.

5.1 Trusted Interrupts

The hyplet can be used to mask the handling of an interrupt so that it will not be visible by the OS driver. Interrupts handled by the hyplet can be verified by TPM or

Trustzone, and pose an extra layer of protection in order to reverse or modify, unlike OS based interrupt handler. In order to modify a normal OS interrupt it is sufficient to elevate privileges to the OS level. In order to modify a hyplet one must first elevate permissions to the OS level, and then attack and subvert the hypervisor itself. To prevent a malicious hyplet from being injected, we offer a static analysis tool that scans the binary ELF and checks that there are no inferences to special purpose registers, such as TTBR0_EL1 or HCR_EL2.

5.2 Protection against reverse engineering

On x86 platforms, TrulyProtect provides anti-reverse engineering Resh *et al.* [2017a], end-point security Resh *et al.* [2017b], video decoding David and Zaidenberg [2014], forensics etc. TrulyProtect relies on Dynamic Root of Trust Measurement (DRTM) attestation to create a trusted environment in the hypervisor to receive encryption keys Rosenblatt *et al.* [2001].

We have used the hyplet to implement a TrulyProtect-like system on the ARM platform. We have encrypted parts of the software and used the hyplet in order to switch context and elevate privileges. Our systems then decodes the code in the hypervisor context (a hyplet), so that the code or decryption keys will not be available to the OS.

Our system for protection against reverse engineering has a cost affiliated with first execution and decryption of the code, but very low per iteration overhead as demonstrated in the table below.

Iterations	Encrypted	Clear
1	1185	1127
10	2737	2597
100	18022	18018
1000	173925	171251
10000	1758997	1670811

Table 4: Duration of stack access in ticks

5.3 Robotics

A kinetic robotic system is composed of a motion controller and axes. The motion controller is referred as the master and the axes are the slaves. The motion controller communicates with the slaves over a fieldbus protocol. A fieldbus Thomesse [2005] protocol is a generic term for real time communication protocols like Sercos XIE *et al.* [2001], EtherCAT Shan *et al.* [2007], CAN LIU and SU [2006], Modbus Rotvold *et al.* [2007] and many others. We can describe a motion controller as receiving the coordinates of each axis (slave), calculating the future coordinates of each axis and transmitting them. This is a cyclic operation; and the closest time it is performed to the time it is sent, the more smooth the kinematics would be. In figure 9 the coordinates are received at cycle *t*, the new position is calculated and sent at cycle *t+1*.

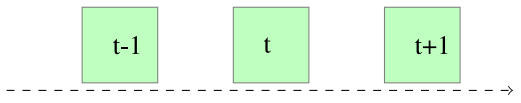

Figure 9: Field Bus

However, some programmers write this code in the kernel, and avoid the kernel-to-user space latency altogether. Since it is not easy to access user space data from an interrupt context, prior to the transmit, we miss the opportunity of last minute calculations. These calculations are important because the hardware timer jitters, so the next position calculations have some small errors. Therefore, this paper offers the hyplet would execute and modify the axes' data prior to the regular kernel timer interrupt, right before the data is sent. Figure 10 depicts this design, the hyplets are in red.

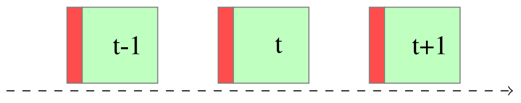

Figure 10: FieldBus with the hyplet

In addition, usually Linux main timers (the tick) are programmed to run at 1Khz, but with the hyplet it is possible to execute the hyp-ISR in much higher frequencies, for instance to run the hyplet in 10Khz, and pass only tenth of the interrupts to the operating system.

6 Usability

The hyplet is very easy to use. The following code is an example of a hyplet. The program maps data and code to the hypervisor through a set of APIs, and when told it routes the IRQ to user space. The program is compiled without any special requirements, and executed like any other ELF binary.

```
int some_global = 0;
/*
 * user_hyplet is executed in a
 * hyplet context.
 * some_global is accessible from
 * EL2 and EL0. the variable odd
 * on the stack may be protected
 * or not.
 * Opaque may be any value
 * passed from the kernel.
 * The function returns indication whether
 * or not to filter the interrupt
  * from the kernel.
 */
long user_hyplet(void *opaque)
{
    int odd = 0;

    some_global++;
    if (some_global % 2)
     odd = 1;
    if (odd)
     return MASK_IRQ;
    return NO_MASK;
}

int hyplet_start(int irq)
{
   int stack_size = 1048576;
   void *stack_addr;

/*
 * Map the entire binary to the
 * hypervisor. This eases the .bss
 * mapping of the program.
*/
  hyplet_map_all();
/*
 * Create a stack and fault it.
*/
   stack_addr = malloc(stack_size);
   memset(stack_addr, 0x00, stack_size);
/*
 * You may choose not to map the stack
 * and use the hypervisor
 * stack for the sake of security.
 * Here we map the stack.
*/
  hyplet_set_stack((long)stack_addr,
    stack_size);
/*
 * Here we mark which function we wish
 * to invoke in EL2.
 * user_hyplet is the hyplet.
*/
   hyplet_set_callback(user_hyplet);
/*
 * begin trapping the interrupt
 * and route it to user space
*/
  hyplet_trap_irq(irq);
  return 0;
}
```

In addition, we added some special APIs.

6.1 Application interface

- Synchronization
  API: *hyp_lock(spinlock),hyp_unlock(spinlock)*
  In cases where the programmer wishes to protect a resource from concurrent access from EL0 and EL2, or EL2 and EL2 from two processors. The locks are implemented as spin locks.

- Get Time
  API: *hyp_gettime()*
  Returns the current time in nanoseconds. It is the uses cntvct_el0 register that holds the the current clock value.

- Printing
  EL2 API: *hyp_print(const char* format,...)*
  EL0 API: *print_hyp()*
  The format print string and the values passed are recorded to a temporary buffer. This buffer is allocated in EL0 and then mapped to EL2. When the program is in EL0, it should call print_hyp, to print the data to the program's terminal, as if it is regular C's printf.

- Event
  API: *hyp_wait()*
  When an user space program needs to be notified of the completion of the hyplet, it can call this API and get notified. There is no restriction on the number of the callers.

6.2 Additional

A hyplet can be removed in two ways:

- Termination
  The minute the process terminates, gracefully or not.

- Unregisteration
  The program explicitly unregisters the hyplet.

It is a good practice to lock the hyplet memory to the RAM to avoid relocation, invalidation or swapping.

7 Related work

EBPF Corbet [2018] Borkmann [2016] is described as in-kernel virtual machine, and it provides the ability to attach a program to a certain trace point in the kernel. Whenever the kernel reaches the trace point, the program is executed without a context switch. eBPF is undergoing a massive development, and is mainly used for packet inspection, tracing and probing. EBPF supports x86 architectures, and ARM. It runs in kernel mode which is considered unsafe, but it uses a verifier to check that there are no illegal accesses to kernel areas, or tampering some registers. Access to user space is done through memory maps.

EBPF uses LLVM and requires clang to generate a JIT code, and has a quite small instruction set. As a consequence, eBPF has serious limitations. Only a subset of the C language can be compiled into eBPF, it has no loops, no native assembly, no static variables, has no atomics, may not take long time, and is restricted to 4096 instructions. Many vulnerabilities in an eBPF program might jeopardize the operating system.

This is not the case with the hyplet. The hyplet is not a program that executes in the kernel's address space, but in the user's address space. So, there is no need for maps to share data between the user and the kernel. The hyplet does not require any special compiler extensions, much less restricted (what mapped prematurely can be accessed) and less complicated to use compared to eBPF. The hyplet is meant to propagate hardware real time events to a user space program, eBPF collects data.

Hyperupcalls Chan *et al.* [2018], which are ePBF extension for a hypervisor, are a mean to run hypervisor code in the guest's kernel context. Hyperupcalls are intended mainly for monitoring the health of the guest VM, and are available only for the x86 architecture. The hyplet on the other hand, only uses the hypervisor and is not intended for control and management of virtual machines. Nevertheless, it is possible to combine eBPF and the hyplet technologies, so that an eBPF program will invoke a hyplet directly.

There has been a significant amount of research on a secure microkernels and microvisors. A prominent microvisor is the OKL4 by Open kernel labs. The OKL4 microvisor Heiser and Leslie [2010] is a secure hypervisor that is supported by Cog Systems and General Dynamics. The OKL4 microvisor supports both paravirtualization and pure virtualization. It is designed for the IoT and mobile industries, and supports ARMv5, ARMv6, ARMv7 and ARMv8.Unlike the hyplet the OKL4 microvisor is a full kernel executing in HYP mode. OKL4 microvisor has an open source sister project microkernel called seL4. Installing seL4 and running it is a challenging task, which requires expertise, as well as to adopt the hardcore of the code. Other microvisors for the ARM platform are Xvisor Xavier *et al.* [2016] and JailhouseBaryshnikov [2016].

Dune Belay *et al.* [2012] is a system that provides a process rather than a machine abstraction through virtualization. Dune offers a sandbox for untrusted code, a privilege separation facility, and a garbage collector. Dune is implemented on Intel Architecture.

Xen-Arm Kanda *et al.* [2008b] is a para-virtualized hypervisor based on Xen framework. Xen ARM hypervisor may host many different operating systems, but requires modifications to the guest OS.

MirageOS Madhavapeddy *et al.* [2015] is a unikernel for ARM, based on Xen Barham *et al.* [2003]. Though the hyplet share similarities to unikernels, the hyplet is not a kernel, but part of a process's address space. When the program is no longer in EL2, the program is a regular Linux program that may use any API available. In addition , the hyplet does not require any complex boot handling, and its memory footprint is a few kilobytes. However, both the hyplet and MirageOS Madhavapeddy *et al.* [2015] gain from the fact that there is no need to context switch, when entering the application code.

Rump kernels Kantee and others [2012] are virtual lightweight containers for drivers in NetBSD Mewburn [2001]. Rump kernels run on top of the hypervisor, and are processes running in a hypervisor mode, wrapped by containers that enables the driver operations, such as threads and synchronization primitives. Rump Kernels are designed for running drivers with little if any modifications, and still leave the kernel monolithic. It is not easy to run the hyplet as a rump kernel in ARMv8a, because there is no easy way to access the kernel's data without a premature mapping to the hypervisor. Moreover, Rump Kernels are implemented on Intel x86 processors Matz *et al.* [2013].

In the area of pure virtualization, some efforts, such as Jailhouse and Xvisor, were made to run a guest OS as a RTOS, for instance, the evaluation of VxWorks Barbalace *et al.* [2008] as a RTOS guest over a Linux KVM Kivity *et al.* [2007], Zhang *et al.* [2010] showed a sub-millisecond interrupt response, or when RT PREEMPT is the guest operating system, Zuo *et al.* [2010] showed an average of $28us$. Heiser Heiser [2011] disagrees with this approach and had the hypervisor execute the RTOS, and the guest as the general OS. However, Jailhouse demonstrates that it possible to run RTOS guest on top of a thin hypervisor.

In the Linux area, the topic of user-space drivers handling IO events, exists in the Linux kernel inside the Universal I/O (UIO) framework. The UIO Agrawal and Malhotra [2012] device driver Ganapathy *et al.* [2008] is a user space driver that blocks until an interrupt arrives. UIO offers an easy way to interact with various hardware devices. UIO device drivers are not suitable for devices with a high interrupt frequency.

8 Summary

8.1 Future work

We intend to implement the hyplet for PowerPC. PowerPC shares ARM capabilities and the hyplet will be efficient. Intel and AMD hyplet will be implemented mainly for completion. We expect that ARM virtualization host extension Brash [2015] becomes available for commercial use, and we will port the hyplet to this architecture. VHE will resolve the security matters altogether.

8.2 Conclusions

We have introduced a new way ARM hypervisor instructions can enhance Linux performance in real time systems. These features allows for security and performance benefits. The hyplet allows coding interrupts with a predictable μs latency and highly efficient RPC. We've implemented hyplets variant as security solution for ARM.

References

Hemant Agrawal and Ravi Malhotra. Device drivers in user space: A case for network device driver. *International Journal of Information and Education Technology*, 2(5):461, 2012.

Antonio Barbalace, A Luchetta, G Manduchi, M Moro, A Soppelsa, and C Taliercio. Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application. *IEEE Transactions on Nuclear Science*, 55(1):435–439, 2008.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

Maxim Baryshnikov. Jailhouse hypervisor. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2016.

Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Osdi*, volume 12, pages 335–348, 2012.

Daniel Borkmann. On getting tc classifier fully programmable with cls bpf. *tc*, (1/23), 2016.

Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel*. Oreilly, 3rd. edition, 2005.

David Brash. The armv8-a architecture and its ongoing development specification, 2015.

Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. Hashkv: Enabling efficient updates in {KV} storage via hashing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 1007–1019, 2018.

Jonathan Corbet. Bpf comes to firewalls, 2018.

Christoffer Dall and Jason Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

Asaf David and Nezer Zaidenberg. Maintaining streaming video drm. In *Proceedings of The International Conference on Cloud Security Management ICCSM-2014*, page 36, 2014.

Hasan Fayyad-Kazan, Luc Perneel, and Martin Timmerman. Linuxpreempt-rt vs. commercial rtoss: how big is the performance gap? *GSTF Journal on Computing (JoC)*, 3(1), 2013.

Diah Ayu Fitriani, Wahyu Andhyka, and Diah Risqiwati. Design of monitoring system step walking with mpu6050 sensor based android. *JOINCS (Journal of Informatics, Network, and Computer Science)*, 1(1):1–8, 2017.

Shaked Flur, Kathryn E Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the armv8 architecture, operationally: concurrency and isa. In *ACM SIGPLAN Notices*, volume 51, pages 608–621. ACM, 2016.

Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 168–178, New York, NY, USA, 2008. ACM.

Thomas Gleixsner. rt-tests.

Hermann Härtig and Michael Roitzsch. Ten years of research on l4-based real-time systems. In *Proceedings of the 8th Real-Time Linux Workshop*, 2006.

Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 66–77, New York, NY, USA, 1997. ACM.

Gernot Heiser and Ben Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM.

Gernot Heiser. Virtualizing embedded systems: Why bother? In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 901–905, New York, NY, USA, 2011. ACM.

Asif Iqbal, Nayeema Sadeque, and Rafika Ida Mutia. An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2110:15, 2009.

Ravindra Jejurikar and Rajesh Gupta. Energy aware task scheduling with task synchronization for embedded real time systems. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 164–169. ACM, 2002.

Wataru Kanda, Yu Yumura, Yuki Kinebuchi, Kazuo Makijima, and Tatsuo Nakajima. Spumone: Lightweight cpu virtualization layer for embedded systems. In *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, volume 1, pages 144–151. IEEE, 2008.

Wataru Kanda, Yu Yumura, Yuki Kinebuchi, Kazuo Makijima, and Tatsuo Nakajima. Spumone: Lightweight cpu virtualization layer for embedded systems. In *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, volume 1, pages 144–151. IEEE, 2008.

Antti Kantee et al. Flexible operating system internals: the design and implementation of the anykernel and rump kernels. 2012.

Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

"Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood". sel4: formal verification of an os kernel. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 168–178, New York, NY, USA, 2008. ACM.

Pavan Kumar and Mani Srivastava. Predictive strategies for low-power rtos scheduling. In *Computer Design, 2000. Proceedings. 2000 International Conference on*, pages 343–348. IEEE, 2000.

Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a m-kernel based os. *ACM Operating Systems Review*, 25(2):51–62, April 1991.

Xiao-qiang LIU and Mei SU. Design of data acquisition system based on can fieldbus [j]. *Instrument Technique and Sensor*, 9:009, 2006.

Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu: Just-in-time summoning of unikernels. In *NSDI*, pages 559–573, 2015.

Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2013.

Luke Mewburn. The design and implementation of the netbsd rc. d system. In *USENIX Annual Technical Conference, FREENIX Track*, pages 69–79, 2001.

Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the arm architecture. *J. Syst. Archit.*, 59(3):144–154, March 2013.

Amit Resh and Nezer Zaidenberg. Can keys be hidden inside the cpu on modern windows host. In *Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013*, page 231. Academic Conferences Limited, 2013.

Amit Resh, Michael Kiperberg, Roee Leon, and Nezer Zaidenberg. System for executing encrypted native programs. *International Journal of Digital Content Technology and its Applications*, 11, 2017.

Amit Resh, Michael Kiperberg, Roee Leon, and Nezer J Zaidenberg. Preventing execution of unauthorized native-code software. *International Journal of Digital Content Technology and its Applications*, 11, 2017.

William Rosenblatt, Stephen Mooney, and William Trippe. *Digital rights management: business and technology*. John Wiley & Sons, Inc., 2001.

Eric D Rotvold, Donald R Lattimer, Michael J Green, Robert J Karschnia, and Marcos AV Peluso. Interface module for use with a modbus device network and a fieldbus device network, July 17 2007. US Patent 7,246,193.

Chun-rong Shan, Yan-qiang LIU, and Ji HUAN. Ethercat-industrial ethernet fieldbus and its driver design [j]. *Manufacturing automation*, 11:025, 2007.

Michael Steil. 17 mistakes microsoft made in the xbox security system. In *22nd Chaos Communication Congr.*, 2005.

J-P Thomesse. Fieldbus technology in industrial automation. *Proceedings of the IEEE*, 93(6):1073–1101, 2005.

Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 11. ACM, 2011.

Bruno Xavier, Tiago Ferreto, and Luis Jersak. Time provisioning evaluation of kvm, docker and unikernels in a cloud platform. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 277–280. IEEE, 2016.

Jingming XIE, Youping CHEN, Zude ZHOU, and Bing CHEN. Sercos protocol and its application in numerical control systems [j]. *Machinery & Electronics*, 5:000, 2001.

Jun Zhang, Kai Chen, Baojing Zuo, Ruhui Ma, Yaozu Dong, and Haibing Guan. Performance analysis towards a kvm-based embedded real-time virtualization architecture. In *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on*, pages 421–426. IEEE, 2010.

Baojing Zuo, Kai Chen, Alei Liang, Haibing Guan, Jun Zhang, Ruhui Ma, and Hongbo Yang. Performance tuning towards a kvm-based low latency virtualization system. In *Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on*, pages 1–4. IEEE, 2010.