User Mode Interrupts

A Must for Securing Embedded Systems

Sandro Pinto
Universidade do Minho
Guimarães, Portugal
sandro.pinto@dei.uminho.pt

Cesare Garlati
prpl Foundation
Santa Clara, CA, USA
cesare@prplFoundation.com

***Abstract*— With the advent of the Internet of Things (IoT), devices are becoming smaller, smarter and increasingly connected. This explosion in connectivity creates a larger attack surface and new security threats. Recent cybersecurity attacks clearly demonstrated that the success of this new Internet era depends heavily on the security of those embedded devices that make up the IoT. In this paper, we argue in favor of a paradigm shift in the way computing systems are conceived and designed. We explain why the free and open RISC-V ISA promises to be a game-changer for embedded security, and we share our experience developing the industry-first RISC-V secure implementation of FreeRTOS based on MultiZone Security, the first Trusted Execution Environment for RISC-V. In the context of this research, we explain how to implement user-mode interrupts to secure modern embedded systems.**

Keywords—Security; Containers, Trusted Execution Environment; TEE; MultiZone; User Mode Interrupt; RISC-V; firmware; embedded systems

I. INTRODUCTION

The world is undergoing an unprecedented technological transformation, evolving from isolated systems to ubiquitous Internet-enabled 'things' capable of generating and handling vast amounts of security-critical and privacy-sensitive data [1]. This novel paradigm, commonly referred to as the Internet of Things (IoT), is a new reality that is enriching our everyday life but simultaneously creating several risks. Recent cybersecurity incidents, such as the Mirai Botnet, have clearly demonstrated that the success of this new Internet era is heavily dependent upon the trust and security built in these IoT devices.

The ongoing cat-and-mouse game of hacks and patches is largely due by the intrinsic lack of security of the traditional computing model, which is not safe nor secure. Mainstream operating systems (OSes) are designed for functionality and speed. These systems follow a monolithic architecture, with most of the services enjoying privileged execution rights. Typically, programs share the same access to code and data and functional blocks communicate via shared memory structures such as buffers, stacks and hypes – a single failure in one component can bring the entire system down [2]. Even more evolved systems that implements virtual memory protection schemas have shown several vulnerabilities, mainly due to the complexity of the software necessary to operate the underlying MMU [3].

Over the last decades, several security-oriented technologies, such as Arm TrustZone [4] and Intel SGX [5], have been developed with the aim of providing stronger security primitives anchored in hardware. However, the sad reality is that these security technologies often fail to deliver on the promise. Firstly, they depend on specific hardware which is not typically available on all platforms. Secondly, the overwhelming complexity of properly implementing these security technologies often results in them not being used at all. Finally, over the last few years, confidence in these systems have been regularly questioned, due to the systematic discovery of critical vulnerabilities mostly due to their closed-source proprietary nature [4, 6, 7].

As the number of IoT devices grows into the trillions, the road to a trustworthy Internet of Things requires an urgent paradigm shift in the way modern computing systems are being built. Bloated 'vertical' monolithic operating systems should give rise to a multitude of light weight 'horizontal' microkernels. Microkernel-based OSes implements a small TCB as the core of the system, with OS services separated into mutually-protected userland servers. Apart from the reduced TCB, microkernels promote a set of design principals, such as the least privilege and fault containment, which favors for security.

Recent advances in computer architectures have brought to light an innovative computer architecture named RISC-V. RISC- V distinguishes from traditional platforms by offering a free and open instruction set architecture (ISA). RISC-V promises to be a game-changer for security by favoring simplicity over complexity, and by defining a comprehensive set of security building blocks in the ISA itself – so that the hardware "hooks" are already built into any RISC-V core. The job of orchestrating these security features and encapsulating their inherent complexity in proper implementations is left to the software layer. And in particular to a new class of light-weight microkernels providing silicon-level containerization and ultimately policy-driven hardware-enforced security through separation. The free and open standard MultiZone Security is the first Trusted Execution Environment specifically developed from the ground up for RISC-V [8]. MultiZone Security differs from traditional TEEs because it does not depend on custom hardware and simplifies the creation of secure end-to-end system stacks.

In this paper, we start by discussing why the security of traditional computing model is fundamentally flawed and why we urgently need to change the way computing systems are designed at the core. We explain why RISC-V promises to be the most secure platform and we share our experience developing the industry-first secure implementation of FreeRTOS for RISC-V. In the context of the capabilities offered by the MultiZone TEE, we explain the implementation of unprivileged user-mode interrupts as a mechanism to enhance the security of modern embedded systems.

II. THE FLAWED TRADITIONAL COMPUTING MODEL

We argue that the security of traditional computing is flawed with regard to the monolithic architecture of mainstream operating systems and the lack of underlying separation provided at the hardware level. A typical example is the Linux implementation of the TCP/IP stack – by definition exposed to remote attack - as part of the kernel itself. Since general purpose OSes are designed for functionality and broad platform support, their size and complexity has grown well beyond the limit that security experts deem acceptable. Systems featuring a bloated TCB are intrinsically more vulnerable. The likelihood of undetected code vulnerabilities increases as a result of a larger number of lines of source code [3]. However, given the lack of pervasive hardware separation mechanisms, even in low-end embedded applications, where the TCB is typically several orders of magnitude smaller, we see general purpose operating systems that combine all functional code blocks in the same privileged space – if any available at all, which leads programs to share the same access to code and data (Fig. 1).

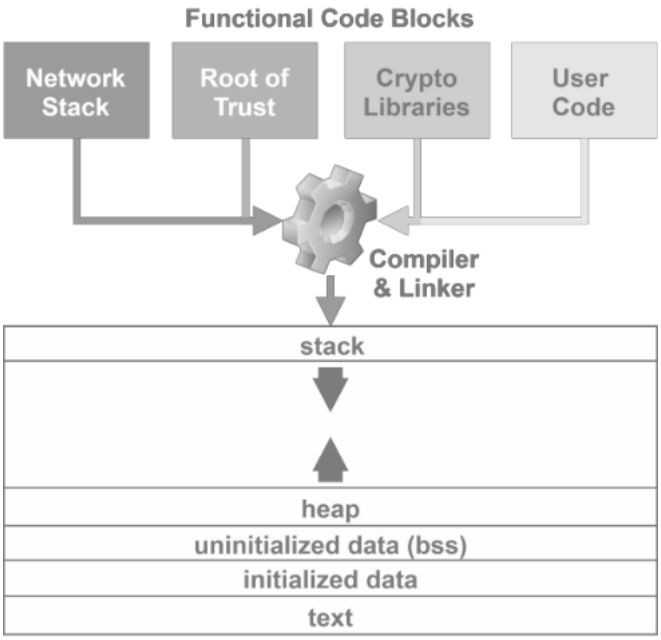

Fig. 1. Traditional computing model.

As the complexity of these systems grows, they tend to become an assemble of different code bases in the form of libraries developed and maintained by different commercial entities and open source communities. The quality and security posture of which is simply impossible to formally verify for non-trivial functionality. Due to the monolithic nature of the kernel, a single vulnerability in one component is usually enough to lead to privilege escalation, to exploit the entire system and likely to pivot into additional network-connected high value targets. For example, FreeRTOS, which has a TCB orders of magnitude smaller than Linux, was recently compromised due to several vulnerabilities in its integrated TCP/IP stack [2].

Even though rich operating systems usually implements some mechanisms intended to restrict unconditional access to memory-mapped resources (e.g., virtual memory via MMU), the complexity of these mechanisms makes them far from unbreachable. For example, the implementation of virtual memory management and hardware memory management units (MMU) has several drawbacks. Firstly, they require relative expensive hardware (e.g., silicon gates and TLBs) and software (e.g., 2-stage or 3-stage translation tables). Secondly, these systems require complex software layers to operate the MMU itself, which tends to increase the TCB of the system. As software is a product of the human intellect, it is guarantee to have defect – known as bugs. The resulting increase in TCB inevitably leads to an increased number of vulnerabilities. Nevertheless, even in resource constrained devices, which are expected to power the IoT, "the design complexity associated with correctly implementing technologies like memory protection units (MPUs) often results in them not being used at all" [9].

The very idea that simply adding complex hardware security primitives automatically results in more resilient systems is naïve at best – and often driven by aggressive marketing strategies. On the contrary, complexity is the enemy of security. Leading TEE solutions, such as Arm TrustZone and Intel SGX, enjoy vast mind share among developers, but have significant limitations. Firstly, they rely on specific hardware which is typically not available on all platforms. Secondly, they are admittedly complex and difficult to understand and properly implement. Finally, over the last few years, they have been witnessing a massive number of hacks and attacks, which have abruptly reduced the confidence in these systems. For example, a recent survey on Arm's TrustZone technology [4] revealed that, according to the National Vulnerability Database (NVD) and several security bulletins (e.g., Qualcomm, Huawei, and Samsung), there are more than 130 known vulnerabilities regarding TrustZone and TrustZone-based TEE – and some can't be patched as they are rooted in hardware.

III. THE NEED FOR A SECURITY PARADIGM SHIFT

While traditional computing systems implement security by adding complexity in the form of new layers of hardware and software, a more modern concept is to start from a minimalist, formally verifiable microkernel to enforces separation. The resulting system has a horizontal structure, with applications and services running side-by-side in isolated domains - see Fig. 2. These containers should be loosely-coupled and should not share any memory-mapped resources. Communications should be implemented through a secure message-based system. Time

and space isolation should be enforced by a preemptive kernel according to policies statically defined by the system designer.

This multi-domain design ensures that faults are contained within the sandbox and cannot affect the other parts of the system. These are analogous to docker containers implemented in servers [11], but do not require the underlying Kubernetes infrastructure which is not practical in resource-constrained devices.

To be resilient, this new security paradigm has to be strictly coupled with innovative processor architectures such as the free and open RISC-V ISA. Introduced in 2011, the RISC-V ISA has rapidly grown in popularity and has now reached a level of maturity suitable for commercial applications. RISC-V promises to be a game-changer for security due to the openness and simplicity of its ISA. The ISA itself defines some security building blocks which include four well-defined privileged levels (rings), a set of physical memory protection mechanisms, and user-level interrupts extensions. The four privilege levels include: Machine mode (M-mode), Hypervisor mode (H-mode), Supervisor mode (S-mode), and User mode (U-mode). The combination of M and U modes is particularly suitable for resource-constrained embedded systems. To control access to physical memory-mapped resources, RISC-V specifies a state-of-the-art physical memory protection (PMP) unit. In addition, the "N" extension allows interrupt delegation to user mode from higher privilege levels. "N" extensions define a framework that can also be implemented in software – via trap-and-emulate – for processors that lack this extension.

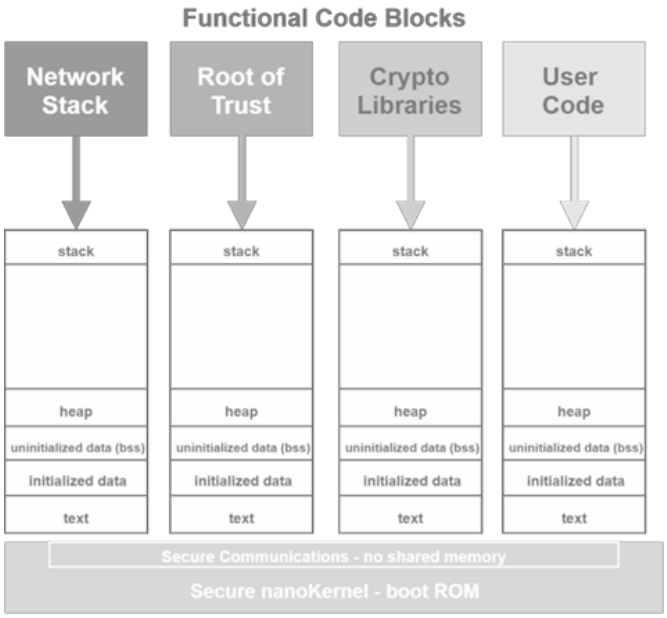

Fig. 2. Microkernel-based system: all functional blocks are isolated.

IV. THE MULTIZONE TEE APPROACH

MultiZone Security is the first Trusted Execution Environment designed from the ground up to leverage the hardware "hooks" built into the standard RISC-V ISA [8]. MultiZone Security allows system designers to properly implement secure RISC-V applications without requiring specialized security skills or changes to the existing development processes. MultiZone software is available on GitHub under Apache license 2.0 - free for non-commercial use.

MultiZone Security segregates the various functional blocks into an unlimited number of physically separated "Zones". With the Multizone Configurator, the system designer defines read / write / execute policies and maps various physical resources to each zone - RAM, ROM, I/O, interrupts. Resource assignment if fine-grained down to 4-byte. Zones can overlap resources although this is not considered a best practice and will be flagged by the configurator. Inter-zone communications are secured via the InterZone messenger, which uses no shared memory. MultiZone differs from legacy TEE technology in several ways: it doesn't require custom hardware primitives - MultiZone works with any standard RISC-V core, it supports an unlimited number of equally secure zones - contrary to the antiquated TrustZone model of the two secure / non-secure worlds, and it doesn't require changes in existing code – as it traps & emulates privileged instructions.

Fig. 3. MultiZone system configured to run four independent zones.

Fig. 3 depicts the MultiZone-based system that we developed to show a real-world example of our proposed security model. The system is configured to run four independent zones. These zones implement the basic functional blocks typically present in embedded connected devices, such as smart sensor and IoT endpoints in general. Zone 1 runs the industry-first RISC-V secure implementation of the popular FreeRTOS, zone 2 runs a secure TCP/IP stack, zone 3 provides Root of Trust, and zone 4 runs a bare-metal command line interface (CLI) for verification and benchmarking of the TEE. All zones are completely isolated and communicate through well-defined message-based interfaces provided by the secure InterZone Messenger. The MultiZone preemptive scheduler multiplexes zones execution according to a round-robin schema. An equally secure and more responsive cooperative behavior is possible by linking the yield () API part of the MultiZone C Library. Our research focuses on secure IoT devices that implement RISC-V M and U modes. To secure FreeRTOS, we adapted its source code to run in unprivileged user-mode. These modifications include FreeRTOS startup code, task management, context switching, exception handling and time management. In addition, we secured exception handling (Section IV.A) and time management (Section IV.B).

A. *Exception Handling*

The RISC-V ISA divides exceptions into two categories: synchronous exceptions and asynchronous interrupts - see Table 1. Synchronous exceptions arise as a result of instruction execution, such as accessing an invalid memory address or executing an instruction with an invalid opcode.

Interrupts, in turn, are external events that are asynchronous to the instruction stream.

Interrupt/ Exception	Exception Code	Description
1	3	Machine software interrupt
1	7	Machine timer interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	11	Environment call from M-mode

Table 1 – RISC-V ISA exception and interrupt cause.

MultiZone User Mode Exceptions. The MultiZone TEE executes exceptions handlers in secure unprivileged user mode, in the context of the zone that triggers the synchronous exception or that is mapped to the specific interrupt. RISC-V exceptions are raised at the highest privilege level. The MultiZone TEE traps into the nanokernel and then forwards execution to the appropriate zone – if not already in scope. MultiZone provides two distinct C Library APIs to register synchronous exceptions and interrupt handlers: ECAL_TRP_VECT () and ECALL_IRQ_VECT () (see Listing 1). The registration of a handler automatically enables the relative interrupt. Each zone can register a separate handler for each RISC-V synchronous exception while external asynchronous interrupt handlers are individually assigned to a single zone according to the policies defined in the MultiZone configuration file.

```
/*  MultiZone API - libhexfive.h */
…

/* Registers a handler against a trap*/
void ECALL_TRP_VECT (int, void *);

/* Registers a handler for an interrupt*/
void ECALL_IRQ_VECT (int, void *);
…
```

Listing 1 - MultiZone C API for registering exceptions and interrupt handlers.

User Mode Synchronous Exceptions. In our secure implementation of FreeRTOS, the xPortStartScheduler method registers all synchronous exceptions with the TEE via the ECAL_TRP_VECT API. All traps point to a common handler _syncexception_entry - implemented in the FreeRTOS assembly file portasm.S. Note that by design, the secure execution of exception handlers in unprivileged mode means that the rich operating system itself isn't trusted with trapping mechanisms other than the ones allowed by the TEE – i.e. yield. The message is placed in the mailbox by the nanokernel when the synchronous exception is triggered, before forwarding it to the zone. It contains the values of the mcause, mtval and mepc which are passed as arguments to the application exception handler described in the previous section. After the application handler's execution, its return value is stored in the stack, so that execution returns to a handler defined address and not necessarily to the original preempted instruction. The MultiZone TEE supports low-latency vectored interrupts that map each interrupt source to its handler. However, we opted for a simplified implementation. All exceptions are served by a single handler responsible for consistent exception entry and exit behavior - e.g., save and restore task context. The _syncexception_entry handler redirects execution to an application-defined method which must be named handle_syncexception. As implemented in our demo application, this method takes in input the register values mcause, mtval, mepc and returns the appropriate value of the register mepc pointing to the instruction to be executed upon exit. The synchronous exceptions entry point is implemented in the _syncexception_entry. After disabling interrupts and saving the preempted context, the ECALL_RECV system call is used to poll messages from the zone's inbox.

```
BaseType_t xPortStartScheduler ( void )
{
     ...
     /* 0x0 Instruction address misaligned */
     ECALL_TRP_VECT(0x0,syncexception_entry);
     /* 0x1 Instruction access fault */ ECALL_TRP_VECT(0x1,
     _syncexception_entry);
     ...
     /* 0x7 Store access fault */ ECALL_TRP_VECT(0x7,
     _syncexception_entry);
     ...
}
```

Listing 2 – Registering of synchronous exceptions in xPortStartScheduler.

User Mode Interrupts. Most interrupts are tied to the application logic and as such handled at that level. Application code registers the relevant interrupt handlers dynamically through the ECALL_IRQ_VECT API - common entry point _interrupt_entry. The final handler handle_interrupt receives in input the value of the mcause register to properly forward the execution – see as an example the interrupt setup shown in Listing 3.

```
/*Entry Point for Machine Timer Interrupt Handler*/ void
vPortSysTickHandler()
{
/* Calculate next compare value */
const uint64_t now = ECALL_CSRR_MTIME();
const uint64_t then = now + (configRTC_CLOCK_HZ /
configTICK_RATE_HZ);
/* Increment the RTOS tick. */
if ( xTaskIncrementTick() != pdFALSE ){ ulPortYieldRequired =
pdTRUE;
}
/* Request next timer interrupt */ ECALL_CSRW_MTIMECMP(then);
```

Listing 3 – Interrupt initialization example at the application level

Our implementation defines two groups of interrupt handlers in the form of arrays: one for core-local interrupts, localISR, and one for external platform-level interrupts, _interrupt_handlers. Therefore, interrupt setup differs depending on whether it is a local or global interrupt. local_irq_en simply adds the handler to local handler array and registers the exact interrupt ID with the TEE. For global interrupts, g_ext_interrupt_handlers first sets up the PLIC, then adds the handler to the global interrupt array and finally registers the machine external interrupt ID (11) with the TEE. The main interrupt handler written in assembly is the _interrupt_entry. This low-level handler acts as prologue and epilogue to the application handler and implements the interrupt entry and exit logic.

```
_interrupt_entry:
    ...
    /* Save RegFile context
    */ pushREGFILE
    /* Save mcause */
    LOAD s0, MCAUSE_OFFSET(sp)
    STORE zero, MCAUSE_OFFSET(sp)
    ...
    /* Save Task context*/
    portSAVE_CONTEXT
    /* Call IRQ handler (a0 = s0 =
    mcause) */ mv a0, s0
    jal handle_interrupt
    ...
```

Listing 4 – Low-level interrupt handler (interrupt entry logic).

Listing 4, shows the interrupt entry logic, till the moment the application level handler is called. The interrupt entry logic starts by saving the preempted context's register file to the stack. Next, it places the value of the mcause register in s0 register, a callee- saved register, which guarantees this value is unchanged until the handler finishes execution. Then the current task TCB pointer is saved via the portSAVE_CONTEXT and execution continues to the application handler. The exit logic is the reverse process and implements the restore operation.

B. Timer Management

RISC-V platforms provide a real-time counter (machine timer) exposed as a memory-mapped machine mode register (*mtime*). In the context of MultiZone, timers are provided to zones through emulation of the machine timer. At the FreeRTOS level, time management reduces to the management of the tick timer. So, modifications were performed on the timer initialization and the interrupt handler itself.

Timer initialization. MultiZone provides a software implementation of the machine timer unique to each Zone. Soft timer initialization is done via the *vPortSetupTimerInterrupt* – see listing 5. It reads the current time via the *ECALL_CSRR_MTIME API* and then calculates the timestamp of the next tick. It then installs the timer handler for the timer exception and sets the Zone's timer compare register via the *ECALL_CSRW_MTIMECMP* - which also enables the exception. The reason for registering the timer handler as software trap rather than asynchronous interrupt is that this specific hardware implementation of RISC-V has only one physical timer available for the whole system. Therefore, its secure implementation is emulated in software by the TEE. Note that this doesn't affect in any way the application flow as soft timer interrupts are at all the effects asynchronous in the context of the Zone execution. The software implementation does however affect resolution and jittering of the zone as soft timer interrupts remains pending until the Zone is in context. The actual impact on the system is however negligible thanks to the cooperative behavior of the MultiZone scheduler.

```
void vPortSetupTimerInterrupt() {

/* Calculate first tick timer compare */
const uint64_t ullCurrentTime = ECALL_CSRR_MTIME();
const uint64_t ullNextTime = ullCurrentTime +
(configRTC_CLOCK_HZ / configTICK_RATE_HZ);
/* Setup mtimer handler */ ECALL_TRP_VECT(0x3, _timer_handler);
/* Request first tick interrupt */
ECALL_CSRW_MTIMECMP(ullNextTime);
}
```

Listing 5 – Implementation of Timer initialization.

Timer interrupt. The timer interrupt handler (Listing 6) is similar to the common interrupt handler. The only difference is that instead of calling the generic interrupt handler, it calls directly the system tick handler vPortSysTickHandler. The vPortSysTickHandler function calculates the value of the timer for the next tick and calls the xTaskIncrementTick, which returns true if a new task is active and sets ulPortYieldRequired value accordingly. This value will later be checked on the handler epilogue to eventually trigger a context-switch. Finally, the soft timer compare register is set to the new tick value - which also re-enables the timer exception.

```
/* Application - Button 0 interrupt
initialization */ void b0_irq_init()
{
    ...
    /* Enable the interrupt */
    ECALL_IRQ_VECT(16+LOCAL_INT_BTN_0, _interrupt_entry);
    localISR[IRQ_M_LOCAL + LOCAL_INT_BTN_0] =
    button 0 handler;
```

Listing 6 – Timer interrupt handler.

V. CONCLUSION

In this paper we have discussed why RISC-V promises to change the way we build our systems for the better. We reported our experience in porting FreeRTOS for MultiZone Security, and we have highlighted the benefits of user mode interrupts for security.

VI. ACKNOWLEDGEMENTS

The authors would like to thank Boran Car and Don Barnetson of Hex Five Security, Inc. for their help with MultiZone and José Martins of the University of Minho for the porting of FreeRTOS.

REFERENCES

[1] Geert-Jan Schrijen and Cesare Garlati. “Physically Unclonable Functions – A new way to establish trust in silicon”. In Proceedings of the Embedded World Conference, ISBN 978-3-645-50173-6, 2018.

[2] Ori Karliner, “FreeRTOS TCP/IP Stack Vulnerabilities Put A Wide Range of Devices at Risk of Compromise: From Smart Homes to Critical Infrastructure Systems.” Zimperium, 18 October, 2018.

[3] Simon Biggs, Damon Lee, and Gernot Heiser. "The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security." In Proceedings of the 9th Asia-Pacific Workshop on Systems, p. 16. ACM, 2018.

[4] Sandro Pinto and Nuno Santos, “Demystifying Arm TrustZone: A Comprehensive Survey.” ACM Computing Surveys, vol. 51, no. 6, article 130, December 2018.

[5] Victor Costan and Srinivas Devadas. "Intel SGX Explained." IACR Cryptology ePrint Archive, no. 086, 1-118, 2016.

[6] Di Shen. "Exploiting TrustZone on Android." Black Hat USA, 2015.

[7] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." 27th USENIX Security Symposium, pp. 991-1008, 2018.

[8] Hex Five Security Inc. “MultiZone Security”. White Paper, 2018. https://hex-five.com

[9] [Brandon Lewis. “Secure IoT devices from the microcontroller, up.” Embedded Computing, 18 July, 2018.

[10] Babak Bashari Rad et al. “An Introduction to Docker and Analysis of its Performance.” International Journal of Computer Science and Network Security, vol.17, no.3, March 2017.