FreeTEE: when real-time and security meet

S. Pinto, D. Oliveira, J. Pereira, J. Cabral, A. Tavares
Centro Algoritmi - University of Minho
{sandro.pinto, daniel.oliveira, jorge.pereira, jorge.cabral, adriano.tavares}@algoritmi.uminho.pt

***Abstract*—The pervasive use of embedded computing systems in modern societies altogether with the industry trend towards consolidating workloads, openness and interconnectedness, have raised security, safety, and real-time concerns. Virtualization has been used as an enabler for safety and security, but research works have proven that it must be extended and improved with hardware-based security foundations. ARM Trustzone has been used for the realization of Trusted Environments, however in this case real-time requirements are completely disregarded.**

This work in progress paper presents FreeTEE, an embedded architecture that emphasizes and preserves the real-time properties of the system but still guarantees security from the outset. TrustZone technology is exploited to implement the basic building blocks of a Trusted Execution Environment (TEE) as a lower-priority thread of a RTOS. Preliminary results demonstrated that the real-time properties of the RTOS remain practically intact.

***Index Terms*—TrustZone, Real-time, Security, TEE, Monitor, FreeRTOS, ARM.**

I. Introduction

Embedded systems are widespread in modern societies and are present in a huge part of our key infrastructures. Driven by cost reduction, time to market pressure and design flexibility, the trend nowadays is towards the consolidation of a wide range of functions into a single intelligent unit [1]. Such consolidation of workloads altogether with the explosion on connectivity leveraged an increasingly connected world where internet-based services penetrated traditional application domains, creating a much larger attack surface area [2]. Obviously, this trend raised security, safety, and real-time (RT) concerns, since embedded systems were not designed to deal with this level of interconnectedness [3] and coexistence.

Virtualization technology has been used as an enabler for safety and security in embedded systems [4], [5], leveraging workloads separation in secure partitions to increase system reliability and stability. However, virtualization, *per-si*, is not enough to provide the desired security level (i.e., it provides to some extent system integrity but no confidentiality) [6] and it must be extended with new security-oriented technologies which promote hardware as the initial root of trust. ARM Trustzone [7] and Intel TXT (Intel Trusted Execution Technology) are examples of existing hardware-based security foundations which have been exploited to design trustable and safe embedded devices from the outset [8], [9].

ARM TrustZone, in turn, has been gaining particular attention not only due to the massive presence of ARM processors in the market (nearly 95% of the world's mobile handsets), but also because GlobalPlatform standardized the concept of a Trusted Execution Environment. The principle behind a TEE is to provide an execution environment which is isolated from the rich execution environment (REE), separating all the security operations, which need to be protected, in a special OS which is isolated from the rich OS. The applicability of this technology ranges from mobile wallets and NFC payments, to premium contact protection, Bring your Own Device (BYOD) and Digital Rights Management (DRM). Being completely devoted to security requirements, TEE specification does not contemplate RT properties on the secure OS. Many approaches have been proposed [8], [9], [10], [11], [12], [13], but no one provides a one-size-fits-all solution. Some of them only exploit TrustZone for virtualization [9], [10], while others or implement a TEE [8] or only extend the TEE architecture with more security features [12], [13].

This work in progress paper goes beyond state-of-the-art presenting a TrustZone-based architecture that implements the basic building blocks of a TEE as a lower-priority thread of a RTOS. FreeRTOS was slightly modified to support secure services as lower priority tasks, and also to schedule the REE only in the idle periods (asymmetrically). A specific monitor layer was developed to mediate the execution of both OSes. The TrustZone API client library as well as the TrustZone kernel module were also implemented. Preliminary results corroborate our predictions, demonstrating that the RT properties of the system remain practically intact.

II. ARM TrustZone

TrustZone technology [7] refers to security extensions implemented by ARM since the ARMv6 architecture. This hardware security extensions provide a secure and separate execution environment that protects the integrity and confidentiality of secure-sensitive processing, by splitting the hardware and software resources into two worlds - the *secure world* and the *normal world*.

A. *TrustZone Hardware*

The TrustZone hardware architecture can be seen as a dual-virtual system, which splits all the system's physical resources into two possible virtual environments. The major changes introduced in the hardware architecture include the ability to tag system resources as belonging to the secure or normal world. The new 33rd processor bit - the NS (Non-Secure) bit -, indicates in which world the processor is currently executing, and is propagated over the memory and peripherals buses. To preserve the processor state during the world switch, TrustZone adds an extra processor mode: the *monitor mode*. When running in monitor mode, the processor state is considered always secure. Since the processor only runs in one world at a time, software stacks in both worlds can be bridged via a new privileged instruction - SMC (*Secure Monitor Call*). The monitor mode can also be entered by configuring it to handle interrupts and exceptions in the secure side.

The memory infrastructure outside the core can be also partitioned into the two worlds through the TrustZone Address Space Controller (TZASC). DRAM can be partitioned into distinct memory regions, each of which can be configured to be used in either world or both. The processor also provides two virtual Memory Management Units (MMUs), and isolation is still available at the cache-level. System peripherals can be also configured as secure or non-secure through the TrustZone Protection Controller (TZPC).

B. TrustZone API

The TrustZone API (TZAPI) [14] is an application API which specifies how normal applications running on the rich OS interact with the isolated execution environment. Basically, following a client-server model, the API defines a set of abstract software interfaces by which non-secure client applications (NSCApps) can interact with the secure services. The API allows clients to send commands and requests to a secure service, and exchange data between both worlds. Secondary features of the API allow, for example, to query the properties of installed services as well as download new security services at run-time. The (publicly available) TrustZone API does not include any specification about how to develop applications running inside the isolated execution environment. Hence, while it could be useful for application developers, by itself it does not fully specify the APIs needed for developing secure services.

III. FreeTEE

Fig.1 depicts the FreeTEE architecture. As can be seen, the software components are distributed between both worlds. Adopting a bottom-up description, the software running in the secure world is composed by the Monitor layer, the T-RTOS and its corresponding RT and secure service tasks. The monitor component, running in monitor mode, works as a gatekeeper and is responsible for managing the World Control Block (WCB) on each world switch. The T-RTOS, running in kernel mode, is the RT environment with extended secure capabilities which allows not only the development of RT tasks but also secure services. The software running in the normal world, in turn, consists of a GPOS with the respective TZAPI-dependent software (i.e., the privileged TZ kernel module and the unprivileged TZAPI library) and the NSCApps.

A. Secure World Software

The system starts booting on the secure world side by performing a set of operations which includes registers and stacks initialization, memory, peripherals and interrupt controller configuration, as well as GPOS image loading. Memory and peripherals are configured as secure or non-secure, and

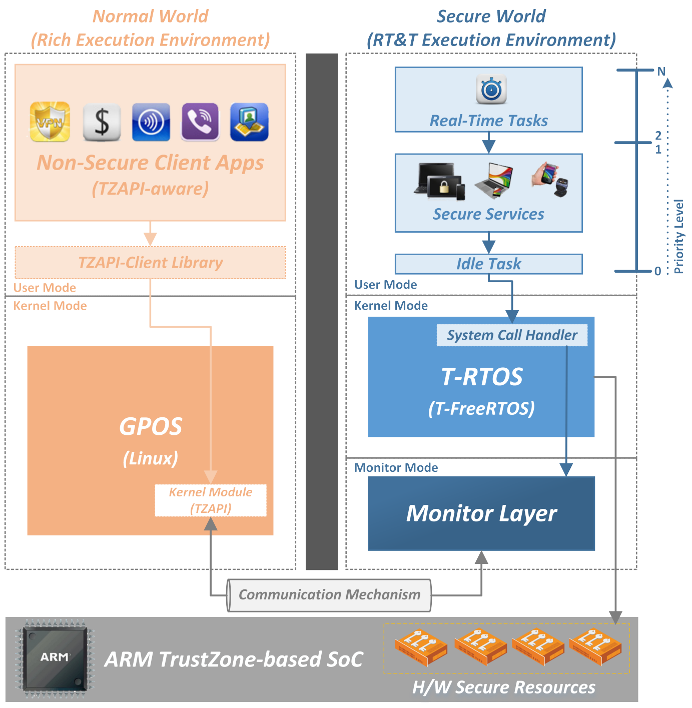

Fig. 1: FreeTEE Architecture

the latter are configured to trigger interrupts (i.e., FIQs and IRQs for secure and non-secure peripherals, respectively). The GIC (Generic Interrupt Controller) is set to route fast interrupt requests (FIQ) to secure world, and interrupt requests (IRQ) for normal world. IRQs are masked during the secure world execution, and the ability of the normal world to manage the configuration of fast interrupts is disabled. Once the system boot finishes, the execution control is transferred to the T-FreeRTOS which will only invoke the Monitor on its idle periods.

1) T-FreeRTOS: T-FreeRTOS is the modified version of FreeRTOS with extended secure capabilities. The main modifications on the kernel side encompass: (i) the implementation of the ”idle scheduling”; (ii) the increasing number of system calls; and (iii) the support for the TZAPI communication. First, in order to preserve the RT properties of the system, the idle task was modified to implement what is called an ”idle scheduling”: the RTOS has a greater scheduling priority than the GPOS, and consequently the GPOS is only scheduled during the idle periods of the RTOS. This is achieved by disabling the FIQ and IRQ bits on the *SCR* register, ensuring that FIQ/IRQ exceptions do not trigger a switch to monitor mode, and consequently the world switch is only performed through a specific system call (SMC). Obviously, the number of system calls was also extended: since the idle task as well as secure services run in user mode, requesting new kernel services dictates new specific system calls. Last but not least, a new small kernel module for managing the TZAPI communication was also implemented: it is responsible for interpreting the commands/data received from the NSCApps and acting accordingly to the desired operation. If the requested action has to be handle by the secure services, the request is transferred to the user space.

2) Monitor: The Monitor component, although running in a higher level of privilege (monitor mode) than T-FreeRTOS, is configured to behave in a passive way so that T-FreeRTOS has the processor as long as RT tasks are ready-to-run. Hence, from the secure side, the Monitor will be dispatched only when T-FreeRTOS is idle or returning from a secure service, by invoking specific system calls that will trigger a SMC instruction. On the other hand, once the GPOS starts executing, the Monitor will be invoked through the specific SMC instruction (i.e., through TZ kernel module) as well as later when a FIQ is triggered (e.g., T-RTOS systick). In all situations, the monitor will perform a world switch operation that saves the state of the current world in its own WCB, and restores the WCB of the ready-to-run world. Due to the intrinsic TrustZone hardware capabilities in banking an extensive list of processor and coprocessor registers, the WCB of each world is minimal and composed only by 28 registers.

B. Normal World Software

The normal world software provides the foundation for application developers to design and implement standard NSCApps that interact with secure services. The GPOS provides a rich and flexible environment by which NSCApps, following the TZAPI specification (TZAPI library), interact with the secure services through the TZAPI kernel module.

1) TZAPI-Client Library: The TZAPI-client library exposes the standardized API defined by TZ specification, abstracting the application developer from the specificities of the TZ message formats and the TZ kernel module Input/Output Control (IOCTL) calls. We implemented all the specification, which includes the descriptors (e.g., `tz_device_t`, `tz_session_t`), control functions (e.g., `TZDeviceOpen`, `TZDeviceClose`, `TZOperationPerform`), encoder and decoder functions (e.g., `TZEncodeUin32`, `TZDecodeArraySpace`), service manager functions (e.g., `TZManagerOpen`) and asynchronous operations, with the exception of functions related to the run-time download and removal of services.

2) TZAPI Kernel Module: The TZ kernel module implemented for the GPOS (Linux) provides a pseudo-character device that implements a logical communication channel (between the normal world and the secure world) on top of the real communication channel, and provides the functional foundation to implement the normal world TZAPI library. It provides a set of specific IOCTLs that semantically understands parameters, allocates memory buffers, encodes and decodes data, prepares the requests and establishes the communication (through SMC instruction). Among implemented IOCTLs, `TZ_IOCTL_SES_OPEN_REQ` and `TZ_IOCTL_ENC_UINT32`, for example, are invoked when the API `TZOperationPerform` for opening a session and the API `TZEncodeUint32` for encoding message is called, respectively.

Thread Metrics	**FreeRTOS**	**FreeTEE**	
	μ	μ	**ov. (%)**
Cooperative Context Switch	141629101	141629105	**+0.0000028**
Preemptive Context Switch	45284714	45284718	**+0.0000077**
Interrupt Processing	73019547	73019549	**+0.0000021**
Interrupt Preemption	36449742	36449745	**+0.0000069**
Message Passing	75779528	75779528	**+0.0000000**
Semaphore Processing	102329325	102329329	**+0.0000039**
Memory Alloc/Dealloc	83009493	83009493	**+0.0000000**

TABLE I: Thread Metrics Benchmark Results

IV. Preliminary Results

The implemented solution was evaluated on the Fast Models emulator, using a model of the Versatile Express (VE) board with a single-core ARM Cortex-A9. In order to measure the impact in the RT properties of the system and assess the overhead introduced by our approach we performed two different kind of experiments: first we compared the native single-core version of the FreeRTOS (v. 7.0.2) against FreeTEE, running Thread Metrics Benchmark Suite, and then we performed specific microbenchmarks to investigate the latency of the system. MMU, caches, branch predictor and others dynamic architectural features were disabled in the secure world side. Performance Monitoring Unit (PMU) were used to assess the world switch and latency overhead.

Table I presents the achieved results from running Thread Metrics Suite. As it can be seen, the overhead introduced by our approach in the RT behaviour is null. This is perfectly understandable because once T-FreeRTOS starts running RT tasks, it will never be interrupted by any secure-related feature. Furthermore, all introduced kernel modifications were carefully implemented to privilege the execution of RT features first. For example, in conditional statements (if, switch), secure features were introduced after RT features to not compromise the execution flow.

Interrupt latency is the measurement of system's response-time to an interrupt, which corresponds to the elapsed time between interrupt assertion and the instant that a response happens. Equation 1 expresses the system latency: τ_H is the hardware dependent time which depends on the interrupt controller on the board as well as the type of the interrupt; τ_{OS} is the OS-specific induced overhead; and τ_{WS} is the monitor-specific induced overhead (world switch).

$$\tau_{IL} = \tau_H + \tau_{OS} + (\tau_{WS}) \tag{1}$$

Our experiments showed that latency in the native system (FreeRTOS) is 172 clock cycles. In equation 1, the last parcel is the extra overhead induced by our approach, which only happens in a specific case: when the RTOS has no RT ready-to-run task and consequently Monitor is invoked to perform a world switch. Since Monitor runs with all interrupt sources disabled, the worst case scenario happens when a FIQ request (e.g., RTOS tick) arrives while a secure to non-secure context switch is starting. In this case, the request is handled only after two complete world switches, which corresponds to a worst case interrupt latency of 335 clock cycles. Since the overhead introduced on latency has a deterministic upper bound, it can be taken into account when designing the RT system.

V. Research Roadmap

Work in the near future will proceed through the implementation of the GlobalPlatform TEE Client and GlobalPlatform TEE Internal specifications. The TEE client API, like TrustZone API, defines a set of interfaces for connecting to and invoking a secure service. The TEE internal API, on the other hand, defines the runtime support for the development of trusted applications running inside the TEE. Since GlobalPlatform consortium not only is leading in providing specifications and standards for the development of security solutions but also providing a more extensive specification than TrustZone API, we will guarantee a higher level of interoperability and standardization in our system.

As FreeTEE is currently implemented (single-core), the GPOS only runs when there is no RT ready-to-run task in the system. Migration to multicore will help to overcome this drawback, and so research will focus on migration to ARM multicore architectures. Several multicore configurations targeting asymmetric (AMP) and symmetric (SMP) multiprocessing should be exploited and experimented, to conclude which one better fits the FreeTEE requirements and use-cases. For example, an AMP approach will be adequate to run each OS simultaneously, however if the GPOS request a secure service to the T-RTOS while it is running a RT task, the response will be delayed until the OS finishes executing such task, and the advantage will be neglected.

From a different perspective, research will continue towards the investigation and development of a secure hardware-based communication mechanism for TrustZone-based architectures. Since no message-protection mechanism exists in TrustZone, man-in-the-middle attacks can be performed to manipulate the messages transferred through the channel (i.e., shared memory). Security analysts have proven the vulnerabilities on the TrustZone insecure channel, and to ameliorate this problem Jang et al. proposed a framework called SeCRet [15]. Our idea will go beyond state-of-art and it will be developed in the form of proof-of-concept, a secure hardware-based communication mechanism.

VI. Conclusion

As the emergence of next-generation embedded devices continues to stretch the market's imagination with unique combinations of applications, not only RT and safety but now also security is emerging as a new dimension in embedded system design. ARM TrustZone technology offers an innovative approach to address security from the outset and is being used as a foundation for a TEE realization, but the problem is that TEE specification does not address RT requirements. This work in progress paper presented FreeTEE, a TrustZone-based architecture that implements the basic building blocks of a TEE as a lower-priority thread of FreeRTOS. This approach preserves the RT properties of the system and still guarantees security from the outset. As demonstrated by preliminary results, the RT properties of the RTOS remain practically unaffected, and only a small overhead on latency is induced.

The research roadmap section described that research in the near future will focus on the implementation of GlobalPlatform specification, and on the migration and exploration of different multicore configurations. Research will then proceed towards the development of a secure hardware-based communication mechanism which reinforces and protects the secure channel.

VII. Acknowledgements

Sandro Pinto is supported by FCT - Fundação para a Ciência e Tecnologia (grant SFRH/BD/91530/2012). This work has been also supported by FCT - Fundação para a Ciência e Tecnologia within the Project Scope: PEst-UID/CEC/00319/2013.

References

[1] G. Heiser, “The role of virtualization in embedded systems,” in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, ser. IIES '08. ACM, 2008, pp. 11–16.

[2] S. Karnouskos, “Stuxnet worm impact on industrial cyber-physical system security,” in *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, Nov 2011, pp. 4490–4494.

[3] D. N. Serpanos and A. G. Voyiatzis, “Security challenges in embedded systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1s, pp. 66:1–66:10, Mar. 2013.

[4] F. Bruns, D. Kuschnerus, and A. Bilgic, “Virtualization for safety-critical, deeply-embedded devices,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. ACM, 2013, pp. 1485–1492.

[5] U. Steinberg and B. Kauer, “Nova: A microhypervisor-based secure virtualization architecture,” in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. ACM, 2010, pp. 209–222.

[6] F. Bazargan, C. Y. Yeun, and M. J. Zemerly, “State-of-the-art of virtualization, its security threats and deployment models,” *International Journal for Information Security Research (IJISR)*, vol. 2, no. 3/4, pp. 335–343, 2012.

[7] ARM, “ARM Security Technology - Building a Secure System using TrustZone Technology,” Tech. Rep., 2009.

[8] L. Jing, J. Chunhua, and Y. Xia, “Design and implementation of security os based on trustzone,” in *Electronic Measurement Instruments (ICEMI), 2013 IEEE 11th International Conference on*, vol. 2, Aug 2013, pp. 1027–1032.

[9] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares, “Towards a lightweight embedded virtualization architecture exploiting arm trustzone,” in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, Sept 2014, pp. 1–4.

[10] O. Schwarz, C. Gehrmann, and V. Do, “Affordable separation on embedded platforms,” in *Trust and Trustworthy Computing*, ser. Lecture Notes in Computer Science, T. Holz and S. Ioannidis, Eds. Springer International Publishing, 2014, vol. 8564, pp. 37–54.

[11] X. Ge, H. Vijayakumar, and T. Jaeger, “Sprobes: Enforcing kernel code integrity on the trustzone architecture,” *CoRR*, vol. abs/1410.7747, 2014.

[12] D. Liu and L. P. Cox, “Veriui: Attested login for mobile devices,” in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '14. ACM, 2014, pp. 7:1–7:6.

[13] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, “Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. ACM, 2014, pp. 90–102.

[14] ARM, “TrustZone API Specification - version 3.0,” Tech. Rep., 2009.

[15] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, “Secret: Secure channel between rich execution environment and trusted execution environment,” in *Proceedings of the 2015 Network and Distributed System Security*, ser. NDSS '15, 2015.