SECSSY HYPERVISOR: SECURITY-SAFETY SYNERGY FOR AEROSPACE

S. Pinto, J. Martins, J. Lopes, M. Abreu, and A. Tavares

Centro Algoritmi, Universidade do Minho, Guimaraes, Portugal

{*sandro.pinto, j.martins, j.lopes, m.abreu, atavares*}*@dei.uminho.pt*

ABSTRACT

Safety has been, for a long time, a major concern for the aerospace industry. The recent increased interconnectivity, altogether with the on-going trend for adopting commercial off-the-shelf computing systems, have raised several security concerns, and proven security is gaining attention as a vulnerability that can also affect safety. Current approaches go towards isolation provided by space and time partitioning of system virtualization. The problem is existent virtualization solutions were primarily prepared to deal with accidental hardware faults or software bugs, and are not ready to fully manage malicious or intentional faults.

This work describes the implementation of SecSSy hypervisor. SecSSy is a hardware-assisted virtualization solution, which addresses security at several stages of system development. SecSSy relies on a secure hardware architecture as the foundation to implement a secure software architecture, all steamed by a safe and secure development process. To the best of authors' knowledge, this is the first solution offering such a complete security-safety synergy for aerospace systems.

Key words: Virtualization; Security; Safety; Hypervisor; TrustZone; Real-Time, ARM.

1. INTRODUCTION

Safety has definitely been the primary concern of aerospace system architects for a long time, but, recently, the mindset started to change. The increasing connectivity of these systems to external networks, as well as the increasing adoption of commercial off-the-shelf (COTS) components for lowering costs, have been significantly increasing the attack surface, and promoting security also as a major requirement for the aerospace industry [1, 2]. The recent discussion around the possible hacking performed by Chris Roberts in a United Airlines flight [3] has raised several questions, and proven security is gaining attention as a vulnerability that can also affect safety in unanticipated ways [4]. These systems are made of a very wide range of software and hardware components, from high-criticality controllers, to low-criticality in-flight entertainment systems, and consequently, the avionics platform might be the target of security issues that could have an impact on the aircraft safety [1, 4].

Space and time partitioning (STP) provided by means of system virtualization has been used as an implementation technique to consolidate and integrate different applications into one single generic computing resource, while guaranteeing separation of concerns between functionally independent software components [5]. Over the last few years several embedded hypervisors have been proposed in the aerospace domain [6, 7, 8, 9], but these systems were primarily prepared to deal with accidental hardware faults or software bugs, not malicious or intentional faults. Recently, Steven VanderLeest et al. [2] presented his vision about how to address the safety and security challenges of next-generation of airborne computing systems. He pointed the use of hardware-assisted virtualization (ARM Virtualization Extensions), altogether with hardware security-oriented technologies (ARM TrustZone), as a promising approach to provide robust partitioning and isolation. We have also implemented RTZVisor (Real-Time TrustZone-assisted Hypervisor) [10] as a real-time hypervisor for space applications assisted by ARM TrustZone [11]. The distinct aspect of RTZVisor is the use of a security-oriented technology for guaranteeing strong hardware-enforced isolation between the multiple guest operating systems (OSes).

Both aforementioned solutions [2, 10] propose the use of robust, hardware-enforced, partitioning via virtualization to address both safety and security issues. We believe this is not enough to achieve the desired security level. Security must be addressed through best practice-based design, common guidelines, and context-awareness [12, 13], without risking other system design properties, such as safety and real-time. This work describes the implementation of SecSSy. SecSSy is an extended version of RTZVisor which targets security also from the onset, by applying a secure development process. The software architecture was reinforced by using an object-oriented implementation, applying coding standards, and using several microkernel principles. To the best of our knowledge, this is the first hypervisor offering such a complete security-safety synergy for aerospace.

2. BACKGROUND

2.1. ARM TrustZone

TrustZone technology [11, 14] refers to the security extensions introduced with ARMv6K in all ARM Cortex-A processors. The TrustZone hardware architecture can be seen as a dual-virtual system, partitioning all system's physical resources into two isolated execution environments. Recently, ARM also decided to extend TrustZone for the Cortex-M processor family, which presents slight differences from TrustZone for application processors. The remainder of this section is focused on the TrustZone specification for Cortex-A processors.

At the processor level, the most significant architectural change is its partition into two worlds: the secure and the non-secure worlds. A new 33^{rd} processor bit, the *Non-Secure* bit, indicates in which world the processor is executing. To preserve the processor state during the world switch, TrustZone adds an extra processor mode: the monitor mode. Software stacks in the two worlds can be bridged via a new privileged instruction - *Secure Monitor Call* (*SMC*). The monitor mode can also be entered by configuring it to handle IRQ, FIQ, and Aborts exceptions in the secure world. Several special registers are banked, such as a number of *System Control Coprocessor* (*CP15*) registers. The TrustZone Address Space Controller (TZASC) extends TrustZone security to the memory infrastructure. TZASC can partition the DRAM into different secure and non-secure memory regions. Secure world applications can access normal world memory but the reverse is not possible. The TrustZone-aware Memory Management Unit (MMU) provides two distinct MMU interfaces, and isolation is still available at the cache-level. System devices can be dynamically configured as secure or non-secure through the TrustZone Protection Controller (TZPC). To support the robust management of secure and non-secure interrupts, the Generic Interrupt Controller (GIC) provides both secure and non-secure prioritized interrupt sources.

2.2. RTZVisor

RTZVisor [10] is a bare-metal hypervisor carefully designed to meet the specific requirements of real-time space applications. RTZVisor exploits COTS ARM TrustZone technology to implement strong spatial and temporal isolation between guests (Figure 1). All data structures and hardware resources are predefined and configured at design time, and devices and interrupts can be directly managed by specific guest partitions. Exceptions and errors are managed through a special component called Health Monitor, which is able to recover guests from undefined states.

RTZVisor multiplexes the several guest OSes over the non-secure side. This means it requires careful handling

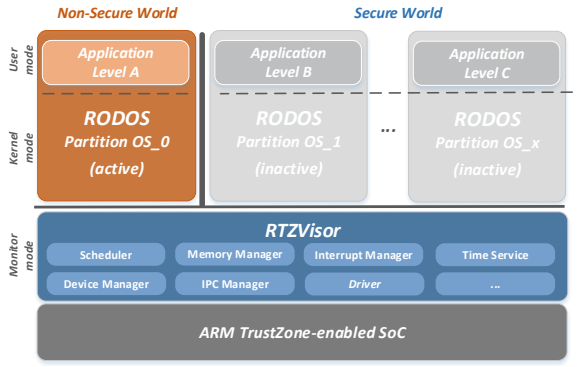

Figure 1. RTZVisor System Architecture

of shared hardware resources, such as processor registers, memory, caches, MMU, devices, and interrupts. Processor registers are preserved into a specific virtual machine control block (VMCB). This virtual processor state (vCPU) includes the core registers for all processor modes (vCore), the CP15 registers (vCP15) and some registers of the GIC (vGIC). RTZVisor offers as many vCPUs as the hardware provides, but only a one-to-one mapping between vCPU, guest and real CPU is supported.

The strong spatial isolation is ensured through the TZASC, by dynamically changing the security state of the memory segments. Only the guest partition currently running in the non-secure side has its own memory segment configured as non-secure, while the remaining memory is configured as secure. The granularity of the memory segments, which is implementation defined, limits the number of supported virtual machines (VMs). Furthermore, since TrustZone-enabled processors only provide MMU support for single-level address translation, it means that guests have to know the physical memory segment they can use in the system, requiring relocation and consequent recompilation of the guest OS. Temporal isolation is achieved through a cyclic scheduling policy, ensuring one guest partition cannot use the processor for longer than its defined CPU quantum. The time of each slot can be different for each guest, depending on its criticality classification, and is configured at design time. Time management is achieved implementing two levels of timing: there are timing units for managing the hypervisor time, and others for managing the partitions time. Whenever the active guest is executing, the timers belonging to the guest are directly managed and updated by the guest on each interrupt. For inactive guests the hypervisor implements a virtual tickless timekeeping mechanism, which ensures when a guest is rescheduled, its internal clocks and related data structures are updated with the time elapsed since its previous execution.

RTZVisor implements device virtualization adopting a pass-through policy: devices are managed directly by guest partitions. To ensure strong isolation between them, devices are not shared between guests and are assigned to respective partitions at design time. To achieve isolation at device level, devices assigned to guest partitions are

dynamically configured as non-secure or secure, using the TZPC. This guarantees an active guest cannot access or compromise the state of a device assigned to another guest. For interrupt managment, RTZVisor configures interrupts of secure devices as FIQs, and interrupts of non-secure devices as IRQs. Secure interrupts are redirected to the hypervisor, while non-secure interrupts are directly sent to the active guest. Interrupts of inactive guest partitions are momentaneously configured as secure, but disabled; as soon as the respective guest becomes active, the interrupt will then be processed.

3. SECSSY HYPERVISOR

SecSSy hypervisor is a refactored version of RTZVisor aiming to achieve a higher degree of safety and security. In terms of system software architecture, the refactoring mainly encompasses the implementation of a microkernel-like architecture, following an object-oriented approach (C++), complemented by the implementation of a secure boot process and secure memory layout.

Besides the microkernel architecture, which is inherently more secure than a monolithic one [15, 16], this implementation targets security from the onset by applying a secure development process and improving code modularity, structure and clarity. This is achieved by employing adapted test-driven development (TDD) techniques [17] throughout the development process, while taking advantage of the benefits provided by object-oriented programming and several C++ features such as a stronger type checking and linkage. Nevertheless, only a subset of the C++ language suitable for secure embedded systems is used, complemented by MISRA C++ coding guidelines [12, 13].

3.1. Microkernel-like Architecture

Figure 2 depicts the SecSSy hypervisor system architecture. This architecture provides two kinds of partitions. Similarly to RTZVisor, guest OS partitions run over VMs in the non-secure world, while inactive guests are kept protected in the secure world. Following the principles of the least privilege and of minimality [18], only the SecSSy core runs with the highest privilege, i.e., in monitor mode. The core implements the basic hardware resource management, partition management, scheduling and inter-partition communication (IPC). Other services and drivers must be placed in a new kind of partition, implemented as secure world tasks, which run in the user mode of the secure world. The services and functionality provided by these tasks can be accessed by other partitions via secure IPC facilities, preventing trusted computing base (TCB) bloating while minimizing fault-propagation between hypervisor modules that would have been, otherwise, included within the core [19].

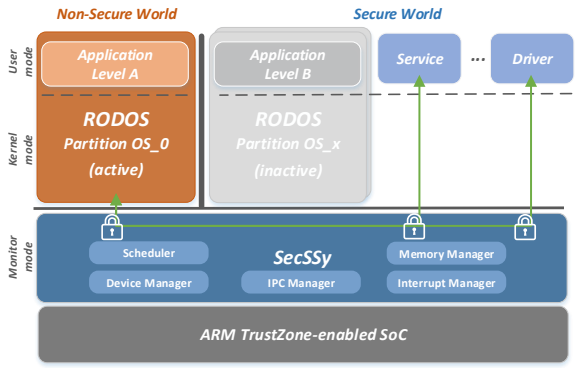

Figure 2. SecSSy System Architecture

Although services and drivers could have been implemented in VMs running in the non-secure world, which would make unnecessary the extra core complexity and the increase systems TCB, implementing them as secure world tasks provides two main benefits. First, considering the existence of the dual-interface of the TrustZone-aware MMU, the hypervisor manages the secure page tables, providing a virtual address space for each task. This enables finer-grained level of control over task memory than that given by the TZASC, which eliminates the need for relocation and recompilation for specific memory regions. Secondly, since world isolation is present at cache level, there is no need to flush caches when switching from a guest VM to a secure world task due to a service request via IPC, which significantly improves performance for RPC-like communication, typical on microkernel systems.

This approach might, however, lead to high levels of IPC traffic to access services and drivers. To mitigate the resultant impact on real-time behavior and overall system performance due to context-switch overhead and scheduling decisions, critical secure world tasks might be migrated to the core, based on a study of the up/down calls frequency for a given application.

3.2. Secure Boot Process

Apart from the system software architecture, security starts by ensuring a secure boot process, which is responsible for establishing a chain of trust that validates all levels of secure software running on the device. For guaranteeing a complete chain of trust, hardware trust anchors must exist, namely, secure storage facilities. Regarding our current target platform, the Xilinx ZC702, a number of secure on-chip storage sources were identified, which include volatile and non-volatile memories. Off-chip memories should only be used to store secure encrypted images (boot time), or non-trusted components such as guest partitions (run time).

After the power-on and reset sequences have completed, code on an on-chip ROM begins to execute. This ROM memory cannot ever be updated, acting as the root of

trust of the system. It starts the whole security chain by ensuring authentication and decryption of the first-stage bootloader (FSBL) image. The decrypted FSBL is then loaded into an on-chip RAM and control is turned over to it. The FSBL is then responsible for the authentication, decryption and loading of the secure system image (containing SecSSy, secure tasks and guest partitions). If any of the steps on the authentication and decryption of the FSBL or the system image is not successful, the CPU is set into a secure lockdown state.

Partition images are not individually encrypted. As aforementioned, they are part of the overall system image. Nevertheless, the addition of another stage of verification, at the partition level, would help to achieve a supplementary level of runtime security for the entire system lifetime. By including an attestation service as a secure task, it would be possible to check and attest partition identity and integrity at boot time, as well as other key components at any time.

3.3. Secure Memory Layout

A secure memory layout, depicted in Figure 3, was devised and tailored specifically for the platform where our system is currently deployed, a ZC702 board. As in RTZVisor, guest partitions are placed in their attributed memory regions which are dynamically configured as secure or non-secure depending on whether the guest is active or inactive. All the code running in the secure world - SecSSy core and secure tasks - is placed in the first 64 MB region, which is always configured as secure memory. Here, only the first 256KB are used, since it corresponds to on-chip memory (OCM) on Zynq. OCM is considered a more secure storage since the memory has no address or data lines in the system on chip device pins, i.e., it cannot be tampered with from the outside, preventing attacks such as cold boot. However, OCM is not safe in the sense it does not provide any error-detection and correction (EDAC) mechanisms which can detect and correct data corruption due to, for example, radiation interferences. Finally, critical data belonging to the SecSSy core is placed apart from other code and data and isolated by unused memory areas populated by magic values, which can later be monitored for integrity (please refer to Section 6). This enables the detection and mitigation of buffer overflow attacks which target core critical data structures.

3.4. Secure IPC

Inter-partition communication is achieved using ports, which are kernel objects that represent endpoints through which information flows. A partition owns a port and is the only one who can read messages from that specific endpoint.

Only asynchronous communication is implemented, meaning that the partition's execution is not blocked

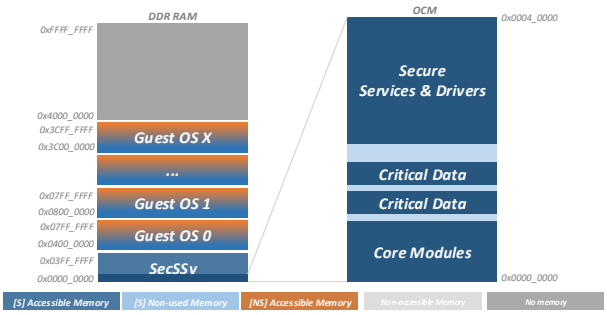

Figure 3. Secure Memory Layout

when sending or receiving a message. In contrast, synchronous communication partitions block at send and receive, resulting in a *rendezvous-style* communication [18]. Thus, our implementation requires double data copy and buffering messages in the kernel, while a synchronous implementation would allow for direct copying between partition address spaces. This extra overhead results in an understandable performance cost. However, by focusing on asynchronous communication, we avoid the asymmetric-trust problem [20]. This issue is specific to synchronous communication and may result in deadlocks or partitions hanging indefinitely while waiting for a communication event from a compromised partition.

To mediate port operations, we have implemented a capability-based access control mechanism [21]. A capability is a reference to a kernel object, in this specific case, a port, which can be assigned to one or more partitions. Owning a capability implies at least some rights over access and operation invocation on the port. Hence, capability aggregates information about its owner, the kernel object it references, and the set of rights over it. As aforementioned, a port owner is the only one who can read messages from the port, thus, its port capability will be the only that references that port with the read right set. Every time a partition invokes an operation on one of its port capabilities, its rights are checked. Each partition operates on a virtual capability-space, and each capability reference is translated to a capability on a global and internal capability-space. Thus, it becomes conceptually impossible for a partition to operate on objects for which it does not possess a capability, as only the capabilities on its capability-space are accessible. The usage of capabilities to mediate IPC port operations provides a fine-grained control over communication, by enabling capability distribution for each kernel object in addition to specific right assignment to each partition. This enables system designers to accurately specify the existing communication channels between partitions while maintaining some flexibility and preserving the principle of least authority [18, 21].

The available IPC primitives, which are always operations on a port, include *Send*, *Receive* and *SendReceive*. When the *Send* operation is invoked, the kernel will store the content of a message in kernel's space, addressing it to the port. For the *Receive*, the partition asks the kernel

if there's some stored message associated with the port. The *SendReceive* extends the *Send* operation by generating a reply capability for the port, i.e, a capability with send rights, which can only be used once. This capability is then inserted in the capability-space of the owner of the port to which the message is being sent to. Reply capabilities are thus leveraged to securely perform client-server type communications, since they remove the need to grant servers full-time access to client ports.

Ports can be configured by its owner to optionally receive asynchronous events upon message arrival. For guest partitions this encompasses a virtual interrupt injection while for secure-world tasks this is accomplished by an Unix signal style upcall. For synchronization purposes, a semaphore kernel object was created, whose access is also managed through capabilities. Finally, as a security preventive measures, system call parameters on IPC operations are subject of intensive sanity checking and data address space verifications. Furthermore message internal buffers are flushed whenever a message is read.

3.5. Coding Standard

For the development of critical systems, the usage of a coding standard is imperative to improve code safety, security, maintainability, and portability. The use of C++ language can be even more troublesome for critical embedded systems as some of its features are not fully specified. Often times, C++ features lead to interpretation mistakes where the code behavior differs from what the programmer expects. Other features are implementation dependent, affecting the portability of the code. Also, some C++ statements may be ambiguous since compilers implicitly perform some operations such as type casting.

To address these issues, SecSSy hypervisor is being developed following the MISRA C++ standard [13]. Due to the various pitfalls of the C++ language, that make it ill-advised for developing critical systems, the main objective of the MISRA C++ guidelines are to define a safer subset of the C++ language suitable for use in safety-related embedded systems. The MISRA guidelines define this safer subset with a series of 228 rules divided across several categories including expressions, namespaces and preprocessing directives, just to name a few. The main purpose of these rules is to restrict the occurrence of known pitfalls and undefined behaviors of the C++ language. Many rules require the programmer to be explicit, especially regarding types used in expressions, solving many of the ambiguities of C++, while others address areas like code portability. MISRA also recommends the use of static analysis tools/techniques, whenever possible, not only for validation but also to enforce the compliance with its guidelines. The compliance of SecSSy code has been enforced with QA-C++ version 4.1.0 from Programming Research [1], which is a static analysis tool for C++ used by industry-leading companies. By itself, the QA-C++ tool does not provide MISRA checks. To achieve that, it was necessary to use the MISRA compliance module version 1.5.1 that integrates with the QA-C++ static analyzer and covers 92% of all the subset's enforceable rules.

[1] http://www.programmingresearch.com/static-analysis-software/qac-qacpp-static-analyzers/

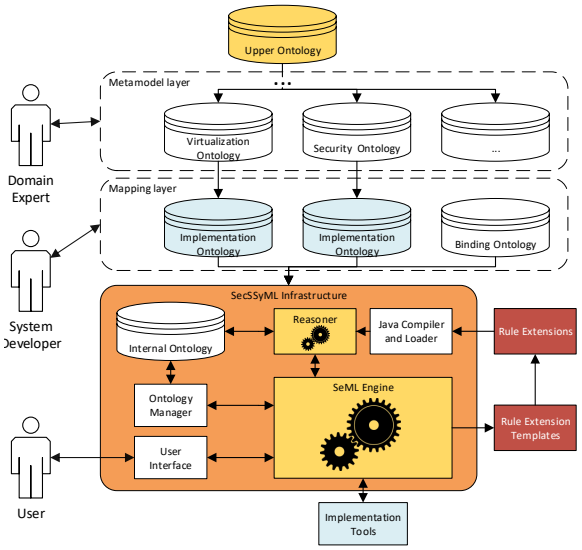

Figure 4. SecSSyML Infrastructure Architecture

4. SECSSYML

Designed to cope with the increasing complexity and variability of hypervisor related modules, the SecSSyML is the modeling language that supports our semantically-enriched modeling infrastructure. Its architecture can be seen in Figure 4.

To achieve a high degree of reliability and extensibility, the infrastructure manages constructs described in Web Ontology Language (OWL), which is a well-known, widely used knowledge representation language. A hypervisor metamodel is derived from an upper ontology, which provides semantic interoperability across all hypervisor domains [23]. Rules can be externally extended to cope with very specific scenarios and increase the overall expressiveness of the ontologies. Variabilities may be asserted as domain knowledge and are solved when the metamodel is instantiated, while inferable model paths are realized without human intervention. This approach minimizes human error and maximizes productivity.

The infrastructure's main idea is to create a model, based on the asserted knowledge, in a guided way, while guaranteeing its validation and implementation viability. In the implementation stage, it also acts as a bridge to convey valid and suitable knowledge to the implementation tools, automating code refactoring and implementation deployment. These tools are independent from the infrastructure and must be provided by the user. The different layers of system design abstract the model designer from

ontological axioms and domain details. By removing this burden from the user, variabilities become easier to manage and automate, avoiding what would otherwise be a tedious, error-prone process.

The next step in the modeling infrastructure is to develop a tool which assists the conversion of independent ontologies to comply with the normalized concepts provided by the upper ontology. This process will generate a considerable expansion of the domain knowledge, granting system developers a higher degree of detail and semantic richness. This tool will also play a critical role in avoiding human errors, which are a substantial problem when dealing with monotonous and repetitive tasks. By guiding the user and detecting incoherencies, the operation will also increase the performance of the infrastructure's reasoner, by following design patterns which minimize knowledge redundancy and maximize inference efficiency.

5. EVALUATION

SecSSy was evaluated on a Xilinx ZC702 board targeting a dual ARM Cortex-A9 running at 600MHz. In spite of using a multicore hardware architecture, the current implementation only supports a single-core configuration. We focus our evaluation on memory footprint and performance overhead. We also evaluate the current level of compliance towards the MISRA C++ coding guidelines.

5.1. Memory footprint

To assess memory footprint results, we used the size tool of ARM Xilinx Toolchain. Table 1 presents the collected measurements for both RTZVisor and SecSSy when compiled with the lowest (-O0) and highest (-O3) level of optimization. Boot code and drivers were not take into consideration. As it can be seen, for -O3, the memory overhead introduced by the SecSSy C++ implementation is only of about 20 % (1.5 KB) when compared with the C implementation. In fact, both implementations show a very minimal TCB of about 7.2 KB and 8.7 KB for RTZVisor and SecSSy, respectively. The main reasons behind this low memory footprint are (i) the hardware support of TrustZone technology for virtualization and (ii) the careful design and static configuration of each hypervisor component. It is also worth mentioning that the size of SecSSy's TCB is not likely to increase significantly in the future, since new services and drivers will be added as secure world tasks.

Table 1. Memory footprint results (bytes)

	.text	***.data***	***.bss***	***Total***
RTZVisor (-O0)	11808	196	8	**12012**
RTZVisor (-O3)	7072	196	4	**7272**
SecSSy (-O0)	16372	28	1472	**17872**
SecSSy (-O3)	7276	12	1472	**8760**

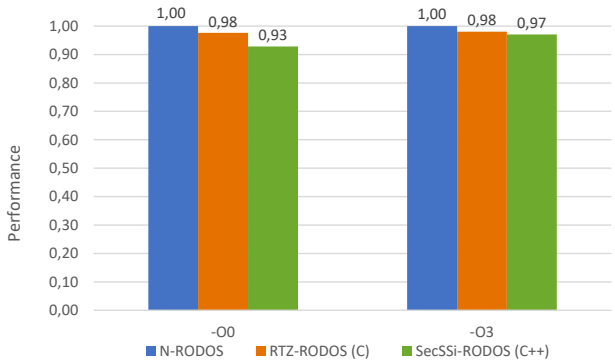

Figure 5. Thread-Metric Benchmark results: relative performance.

5.2. Performance

The Thread-Metric Benchmark Suite consists of a set of benchmarks specific to evaluate real-time operating systems (RTOSes) performance. The suite comprises 7 benchmarks. For each benchmark the score represents the RTOS impact on the running application, where higher scores correspond to a smaller impact.

RTZVisor and SecSSy were configured with a 10 milliseconds guest switching rate. Caches and MMU support for guest OS partitions were enabled, which means MMU and caches maintenance operations were needed at each guest switch. Both systems were set to run one single guest partition, and the hypervisor scheduler was forced to reschedule the same guest, so that results can translate the full overhead of the complete guest-switching operation. We ran benchmarks in the native version of RODOS and compared them against the virtualized versions in RTZVisor and SecSSy for the two compilation optimization scenarios (-O0 and -O3). Figure 5 presents the achieved results, corresponding to the normalized values of an average of 100 collected samples for each benchmark. From the experiments it is clear that the virtualized versions of RODOS only present a small performance degradation when compared with its native execution. For -O3 optimization, the performance degredation for RTZVisor is about 2% while for SecSSy it is about 3%. Hence, we can conclude that the C++ does not bring a high performance penalty, specially when exploiting compiler optimizations.

5.3. MISRA C++ Compliance

To evaluate the evolution of the level of MISRA C++ compliance, we compared the state of the project from

when we first ported the hypervisor to C++ to the current state of the implementation, after correcting many of the violations reported by the static analyzer. For this we use the PRQA rule compliance reports generated by the QA-C++ tool.

The total count of rule violations decreased from 1151 to 273, while the total number of compliant rules increased from 160 to 196 (out of 219 enforced rules). This is reflected by the principal metric given by the compliance report, the total project compliance index, which shows an increase from 73% to 90%. The real value of the index is further increased by the number of deviations that are documented. Deviations are intentional and explicit non-adherence to rules justified by some specific reasons. Also, we believe that some of the rule violations reported by the tool are false positives. We are working together with PRQA to confirm and work around these issues.

The largest number of violated rules appertains to group 5, which concerns rules related to expressions. Since SecSSy originates from a C implementation, a considerable number of violations originate from type conversions in expressions which are not yet completely resolved. We expect to achieve a much higher degree of compliance in the near future by analyzing and correcting them.

6. ONGOING WORK

At the moment of writing of this paper, SecSSy is still being extended to provide a higher degree of design flexibility, performance and security.

The capability-based access control mechanism is being extended beyond IPC ports to a wide range of kernel objects. Such extension includes objects to control hardware resources such as interrupts, memory or devices. This is essential to assign resources to secure tasks. For example, an interrupt can be easily assigned to a guest partition by configuring it as non-secure interrupt source when the guest executes. The guest then accesses the GIC to configure the interrupt. It will receive it through its interrupt vector as it expects. However, tasks cannot directly access the GIC nor receive interrupts. By assigning an interrupt capability to a task (e.g., a driver), it can then invoke the object operations to configure the interrupt and assign it one of its ports which receives a message when the interrupt is triggered.

Furthermore, a mechanism that allows the sharing of capabilities is being developed. Partitions will be able to grant capabilities with modified access rights to other partitions [21]. These granted capabilities can later be revoked. This mechanism is useful for implementing shared memory, where partitions share capabilities for their memory segments, which the receiver can use to ask the core to map to their own address space. Using this sharing mechanism a performance boost is obtained for data processing service provision. Guest partitions can share the segment containing the data to be processed with a service (e.g., encryption/decryption service), which can then operate directly on the data, eliminating the needed to transport the data via normal IPC message passing. When the task is done, it signals the client partition, which revokes the capability for the memory region.

Software attacks often sabotage the legal control flow or critical data in a vulnerable program. Using unsafe languages (e.g., C and C++), attackers exploit memory related errors (e.g., buffer overflows) to write data to unintended locations. The expressiveness of an attack varies from executing newly injected code to not changing program's control-flow at all. Control flow integrity (CFI) [22] is being implemented using a software shadow stack, which records every address of every branch operation for function calls and returns. The collected data is then compared to a control-flow graph to detect illegal branch addresses, in a separate core. SecSSy's code is absent of indirect branch instructions, so this CFI implementation mostly tries to detect stack corruption attacks on the function's return address through comparison. While CFI tackles control related attacks to a certain extent, i.e. only works at function call/return granularity, the hypervisor's data plane must also be secured. Data integrity (DI) is a technique which identifies store operations to critical static variables and logs written values using instrumentation. Logged values are then tested against rules provided by the developer, also in a separate core. The developer must identify variables whose value affect hypervisor's normal behavior and what conditions must be met to attest their validity. CFI and DI must be implemented together in order to secure both data and control-planes of SecSSy. Both techniques will use a separate core to perform their respective security operations, minimizing performance penalties. To communicate between the two cores, several circular buffers are used by both CFI and DI to log relevant data for parallel processing. The instrumentation, inserted using GNU GCC compiler plugin API, writes all relevant data to these data structures while the respective security application reads and processes them. The existing MMU ensures read-only permissions for these data structures to avoid log tampering, only allowing write operations when logs must be performed in the hypervisor's code. W $\oplus$ E (write xor execute) security feature also avoids attacks trying to subvert instrumentation purposes.

7. CONCLUSION

Aerospace industry is evolving at a frenetic rate. The upward trend for interconnectivity has demonstrated security vulnerabilities can impact the safe operation of such systems, and that an urgent change in the mindset is needed: safety and security can no longer be siloed functions - they must be aligned. To address this problematic we proposed SecSSy as a hardware-assisted virtualization solution which addresses safety and security at several stages of system development. The performed experiments demonstrate security countermeasures do not

have impact on the operation of the system. To the best of our knowledge, SecSSy is the first hypervisor offering such a complete security-safety synergy.

Current research aims at porting SecSSy for other platforms, as well as supporting other RTOSes as guest OSes. We are also currently implementing support for asymmetric multiprocessing (AMP), but we plan to explore other multicore configurations. Work in the near future will focus on an extensive and exhaustive evaluation of real-time aspects with short-term and long-term tests. Extension of SecSSy for new generation Cortex-M platforms is also at the top of our priorities, but we still need to wait until the release of the first ARMv8-M boards.

ACKNOWLEDGMENTS

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT - Fundação para a Ciência e Tecnologia - (grant SFRH/BD/91530/2012 and UID/CEC/00319/2013). The authors would like to thank PRQA for providing tools and support for MISRA C++ compliance, as well as their research colleagues João Alves and Miguel Araújo for the valuable work in the SecSSy infrastructure and proofreading of the paper.

REFERENCES

[1] Dessiatnikoff, A., et al. "Potential attacks on onboard aerospace systems." IEEE Security & Privacy 10.4 (2012): 71-74.

[2] VanderLeest, S. H., et al. "MPSoC hypervisor: The safe & secure future of avionics." Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th. IEEE, 2015.

[3] Zetter, K. "'Feds say that banned researcher commandeered a plane"', Wired. [Online]: https://www.wired.com/2015/05/feds-say-banned-researcher-commandeered-plane/

[4] Paulitsch, M., et al. "Evidence-based security in aerospace: From safety to security and back again." Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on. IEEE, 2012.

[5] Windsor, J., et al. "Time and space partitioning in spacecraft avionics." Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on. IEEE, 2009.

[6] Crespo, A., et al. "Xtratum an open source hypervisor for TSP embedded systems in aerospace." Data Systems In Aerospace (DASIA), Istanbul, Turkey, 2009.

[7] VanderLeest, S. H. "ARINC 653 hypervisor." Digital Avionics Systems Conference, 2010 IEEE/AIAA 29th. IEEE, 2010.

[8] Joe, H., et al. "Full virtualizing micro hypervisor for spacecraft flight computer." Digital Avionics Systems Conference, 2012 IEEE/AIAA 31st. IEEE, 2012.

[9] Tavares, A., et al. "Rodosvisor - an object-oriented and customizable hypervisor: The CPU virtualization." IFAC Proceedings Volumes 45.4 (2012): 200-205.

[10] Pinto, S., et al. "Space and Time Partitioning with Hardware Support for Space Applications." Data Systems In Aerospace (DASIA). Vol. 736. 2016.

[11] Pinto, S., et al. "LTZVisor: TrustZone is the Key." 29th Euromicro Conference on Real-Time Systems (Leibniz International Proceedings in Informatics), Marko Bertogna (Ed.), Vol. 76. Schloss DagstuhlLeibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:14:22.

[12] Delic, E., et al. "Validation of a SIL3 middleware for safety-related system-on-chips." Information & Communication Technology Electronics & Microelectronics (MIPRO), 2013 36th International Convention on. IEEE, 2013.

[13] Motor Industry Software Reliability Association. MISRA C++: 2008: guidelines for the use of the C++ language in critical systems. MIRA, 2008.

[14] ARM, ARM Security Technology - Building a Secure System using TrustZone Technology, Technical Report PRD29-GENC-009492C, 2009.

[15] Heiser, G. "Secure embedded systems need microkernels." USENIX 30.6 (2005): 9-13.

[16] Wang, X., et al. "'Enhanced Security of Building Automation Systems Through Microkernel-Based Controller Platforms."'. Technical report 2017-3, Argus Cybersecurity Lab, University of South Florida. February, 2017.

[17] Grenning, J. "Applying test driven development to embedded software." IEEE Instrumentation & Measurement Magazine 10.6 (2007).

[18] Heiser, G., et al. "L4 microkernels: The lessons from 20 years of research and deployment." ACM Transactions on Computer Systems (TOCS) 34.1 (2016): 1.

[19] Hohmuth, M., et al. "Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors." Proceedings of the 11th workshop on ACM SIGOPS European workshop. ACM, 2004.

[20] Shapiro, Jonathan S. "Vulnerabilities in synchronous IPC designs." Security and Privacy, 2003. Proceedings. 2003 Symposium on. IEEE, 2003.

[21] Lackorzynski, A., et al. "Taming subsystems: capabilities as universal resource access control in L4." Proceedings of the second Workshop on Isolation and Integration in Embedded Systems. ACM, 2009.

[22] Abadi, M., et al. "Control-flow integrity principles, implementations, and applications." ACM Transactions on Information and System Security (TISSEC) 13.1 (2009): 4.

[23] Niles, I., et al. "Towards a standard upper ontology." Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001. ACM, 2001.