Isolating Real-Time Safety-Critical Embedded Systems via SGX-based Lightweight Virtualization

Luigi De Simone
Università degli Studi di Napoli Federico II, Italy
luigi.desimone@unina.it

Giovanni Mazzeo
University of Naples "Parthenope", Italy
giovanni.mazzeo@uniparthenope.it

***Abstract*—A promising approach for designing critical embedded systems is based on virtualization technologies and multi-core platforms. These enable the deployment of both real-time and general-purpose systems with different criticalities in a single host. Integrating virtualization while also meeting the real-time and isolation requirements is non-trivial, and poses significant challenges especially in terms of certification. In recent years, researchers proposed hardware-assisted solutions to face issues coming from virtualization, and recently the use of *Operating System (OS) virtualization* as a more lightweight approach. Industries are hampered in leveraging this latter type of virtualization despite the clear benefits it introduces, such as reduced overhead, higher scalability, and effortless certification since there is still lack of approaches to address drawbacks. In this position paper, we propose the usage of Intel's CPU security extension, namely SGX, to enable the adoption of enclaves based on *unikernel*, a flavor of OS-level virtualization, in the context of real-time systems. We present the advantages of leveraging both the SGX isolation and the unikernel features in order to meet the requirements of safety-critical real-time systems and ease the certification process.**

***Index Terms*—Real-time, Intel SGX, Unikernel, Virtualization**

I. Introduction

In recent years, *Critical Real-Time Embedded Systems* (CRTES) are shifting towards an *integrated paradigm*, in which multiple applications share the same physical platform resources [1]. Such an approach allows running applications with different level of criticality on the same embedded platform. The enabling technologies have been multi-core processors and virtualization. The former enhances performance by reducing the overall costs, size, weight, and energy consumption [2]. While the latter is typically adopted to create separate domains of CRTES tasks. Despite promises given by these technologies, there are several drawbacks to be addressed [3]. More precisely, the temporal, spatial, fault, and I/O isolations between virtual domains are at risk in hypervisors' hands. Additionally, the design of CRTESs is still bounded to a rigorous certification process, in which developers must provide evidence about a huge amount of software and its level of partitioning, showing documented proofs about, e.g., fault containment between a virtual domain to another, absence of temporal interferences between critical and non-critical functions. In order to increase the isolation of virtual domains, the research community explored the possibility of leveraging *hardware-assisted* solutions. The most explored approach has been to use security features provided by ARM, i.e., the ARM TrustZone technology [4], indeed enabling a more powerful form of isolation assisted by the hardware. ARM TrustZone, in fact, supports a so-called "dual world" execution [5]–[15]. Usually, the *non-secure world* is used as an environment for running VMs, which are managed by the hypervisor software that runs in the *secure world*. Researchers used the TrustZone dual-guest configuration for running a general-purpose OS (GPOS) within the non-secure world, while a real-time OS (RTOS) runs in the secure-world having a full view of the entire system. In this way, the critical tasks running on top of the RTOS are isolated by non-critical tasks.
Another emerging trend in real-time applications is exploiting *OS-level virtualization* (also named container-based virtualization) [16], which —unlike fully emulating a hardware machine— it abstracts OS processes (called containers) by extending the (host) OS kernel. The main reason behind the usage of containers in real-time domains [17]–[20] is to reduce the overhead affecting VMs and better scale when a larger number of applications of different criticalities are in place. Indeed, the container approach does not require to replicate the entire OS environment for every system. However, the performance benefits introduced by containers come at the cost of reduced isolation, threatening the practicability of OS-level virtualization under real-time and safety requirements. In the context of OS-level virtualization, the sole approach to somehow face isolation issues may be to run a single application in its virtual domain. Such model is known as *unikernel* or library OSes [21], in which the full software stack of a system, including OS components, libraries, language runtime, and applications, is compiled into a single VM that runs directly on a general-purpose hypervisor. This approach introduces benefits such a small code base, low attack surface, and an effortless certification process due to the low amount of software to be verified [21]. However, stronger isolation proofs to be reported to certifiers are still lacking.

In this position paper, we propose a hardware-assisted solution for leveraging *OS* virtualization in the field of CRTES. In particular, we discuss the potential adoption of the well established Intel *Software Guard eXtension* (SGX) extension [22] to enable powerful isolation when OS-level virtualization is pursued in CRTES applications having certification requirements. The SGX is the technological implementation of hardware-assisted trusted execution, as conceived from Intel. SGX belongs to the

978-1-7281-5138-0/19/$31.00 ©2019 IEEE
DOI 10.1109/ISSREW.2019.00089

same category of ARM TrustZone since its *Trusted Execution Environment* (TEE) is internal to the CPU perimeter. Intel SGX provides capabilities for securing user space applications without the need of calling privileged OS code. Basically, SGX provides tools for creating memory areas called *enclaves*, which protect application code and data from accesses by other software, including higher-privileged software. Memory pages which are within an enclave can not be accessed by code outside of the enclave. The proposed approach is intended for executing an RTOS based on *unikernels* into an SGX enclave. Besides the advantages for CRTES, unikernels represent also one of the best ways to bypass the impossibility of issuing *syscalls* (normally executed at *ring0*) from within an enclave that only supports *ring3* functions [23]–[26]. Unfortunately, nowadays, the availability of unikernel-based RTOSes are still poor. *MirageOS* [27] is the sole example. Finally, we examine the impact that such an approach could entail during the certification process, by focusing on the isolation properties that must be fulfilled.

II. OS-level Virtualization in real-time domain

Virtualization technologies are the core enabler of several computer engineering applications, ranging from cloud computing to real-time embedded systems. In contrast to cloud computing, in CRTES development there is a need for specific mechanisms that guarantee and enforce the execution of applications to meet timing and safety requirements [3]. In years, several techniques were developed to abstract physical resources in virtual, from the classical full-virtualization and para-virtualization, to more recent *OS-level* virtualization [28], [29].

OS-level virtualization, allows running multiple appliances without hardware virtualization. The idea behind container-based virtualization is to enhance the abstractions of OS processes (called containers), by extending the (host) OS kernel, in order to have a virtual domain with its own virtual CPU and virtual memory like in traditional OS processes, a virtual filesystem, virtual network, IPCs, PIDs, and users management. These virtual resources are distinct for each container in the system. Currently, the most used container-based virtualization technologies are LXC [30], Docker [31], and OpenVZ [32].

As mentioned before, virtualization has to address the critical problem of guarantee the isolation among virtual instances [33]–[35]. In the more general sense, isolation means the fact that something is independent and disentangled to the behaviors of other things. Thus, the virtualization layer has to be in complete control of virtualized resources, and applications running on a virtual domain must have the illusion to be completely isolated from others. In general, in the virtualization context, we consider two main isolation properties that are mandatory.

The *temporal isolation* (also known as *performance isolation* or *temporal segregation*), is the capability of isolating or limiting the impact of resource consumption (e.g., CPU, network, disk) of a virtual domain on the performance degradation of the other virtual domains, but also against the host. This means that a critical task running on a virtual domain (e.g., task on a VM or within a container) must not cause severe delays of other critical and non-critical tasks, leading to a phenomenon like starvation, reduction of throughput, and increased latency. Temporal isolation is crucial in embedded systems when critical tasks within containers need to assure SLA guarantees about performance. In the context of safety-critical applications, some standards (e.g., IEC 61508-3 Annex F [36], ISO 26262-6 Annex-D [37], ARINC-653 [38], DO 178 6.3.3f [39], CAST-32A [40]) suggest to exploit a cyclic scheduling among virtual domains.

The other crucial isolation property is the *spatial isolation* (also known as *memory isolation* or *spatial separation*). Such property describes the capability of isolating code and data between virtual domains, and between virtual domains and the host kernel. This means that a task should not be able to alter private data belonging to other tasks, including the devices allocated to a specific task. Break spatial isolation will likely lead to unexpected behaviors or worse a system crash [41]. Usually, spatial isolation is enforced by hardware memory protection like Memory Management Unit (MMU), which protects the task's virtual memory space. Furthermore, by considering the case of shared devices, we need to focus also on the *I/O isolation* property. Often, the IOMMU is exploited to properly address the isolation of memory-mapped devices, and in some cases, the access on hardware devices from a different virtual domain is serialized.

Finally, another kind of isolation property that must be addressed is *fault isolation* (also known as *fault/error containment*). Fault isolation means that potential faults that occur in one virtual domain should not be propagated towards hypervisor and/or other partitions, leading to hangs or even halting the entire system.

Bearing in mind the needs of a virtualization approach that is both lightweight and provide the isolation properties mentioned above, *unikernels* are recently gaining a significant attention [42]. The *unikernel* is an approach of linking an application with OS components. In particular, such components include the core services of a kernel like memory management, scheduler, network and disk stack, and device drivers. Thus, the application and the kernel have a unique address space, creating a standalone binary image that is bootable directly on physical and/or virtual hardware [43]. The big advantage that unikernels provide is that the kernel functionalities can be specialized to fit the needs of the target application. For example, the developer would want to increase a specific aspect of performance of its application (e.g., the network throughput) or to improve isolation. Since unikernel is a lightweight solution for virtualization, it could be a promising solution to be adopted in the context of safety-critical embedded systems, where the needs of having a well-defined software component would facilitate the certification process. Another example of the potential of unikernel in the embedded systems is the predictability. Indeed, since moment by moment, only one task/process is running on the unikernel, we can say with good confidence that if an operation is completed in a specific amount of time, it will take the same amount of time every time will be executed. If we compare such behavior against the Linux kernel, for example, there are various factors of unpredictability, like page faults, internal locks, scheduling jitter, and so on. Thus, unikernels would be also a promising solution for the measurement of the worst-case execution time (WCET), which is mandatory and usually is a non-trivial task both in single- and multi-core platforms. Naturally, we need to consider that the predictability of unikernels is valid in the real-time domain as long as the underlying hypervisor schedules the task to the CPU always in the same way. Thus, we need to take into account the possibility of using real-time hypervisors for solving the problem of determinism.

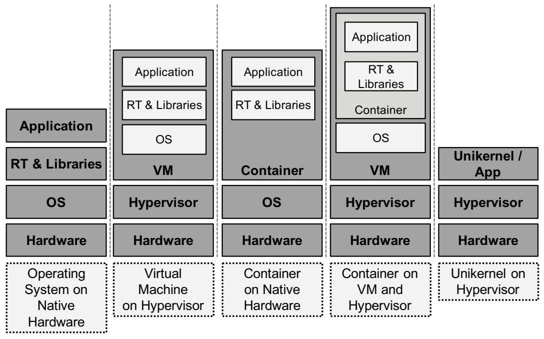

Fig. 1. Virtualization deployments.

Fig. 1 summarizes potential virtualization approaches that could be used in the development of CRTES, from the classical hypervisor-based, passing through the container-based to the unikernel model.

III. Enhancing Isolation via Intel SGX

Leveraging OS virtualization for CRTES is non-trivial as the confinement of containers' domains is more difficult to enforce and demonstrate, due to the weak isolation that LXC *namespace* and *cgroup* create [44]. In this sense, a *unikernel* could represent a solution. Its properties of an extra lightweight OS can certainly facilitate the certification process in terms of software verification. However, *unikernels* are still subjected to isolation problems that could lead to interferences among critical tasks of CRTES. In this paper, we propose the adoption of Intel SGX-enabled *unikernels* in the context of CRTES to provide guarantees on the isolation for real-time tasks running in dedicated domains. In fact, the hardware-assisted security isolation features of Intel SGX can be leveraged to this end as other researchers similarly did with ARM TrustZone to segregate VM domains [5]–[15]. SGX is a security technology that is catching on in both research and industrial communities. It is a set of new hardware instructions that isolate sensitive code and data processing, even from users with *root* privileges. SGX enables a confined region of execution, namely *secure enclave*, whose access is supervised by the hardware that enforces *isolation* of processes. The SGX threat model assumes that most of the host stack is untrusted. Thus, the CPU ensures that the enclave memory is not accessible by any part of the system, except for the code running inside the enclave itself. For the purpose of this paper, it is important to notice that there are two main drawbacks of SGX to be taken into account. That is, *i*) the physical memory size dedicated to all the instantiated enclaves is limited to $128MB$, *ii*) the execution of *syscalls* is forbidden in the enclave as the OS is considered untrusted.

Currently, there are solutions where SGX was used to enhance the security of unmodified applications basically by running them in SGX-secured domains [23]–[26]. These studies pursued two different approaches for providing *syscall* support to the application into an enclave [45]. In particular, in a first case Arnautov et al. [23] use an SGX-extended *libc* library to expose an external (and optimized) *syscall* interface, which is SGX-shielded. In a second case [24]–[26], researchers leverage the single-address space property of *unikernels* (i.e., *Linux-Kernel-Library*, *Graphene*, and *Drawbridge*, respectively) to execute *syscalls* directly inside the SGX enclave. Essentially, the latter category of studies port an entire *unikernel* into an enclave to provide lightweight OS support to the application. Contrariwise, Sfyrakis et al. [46] propose the adoption of SGX to secure only some computations of a *MirageOS unikernel* [27].

In this position paper, we lay the foundations for a solution where *unikernels* —of different or same typology— run tasks of distinct criticality in SGX enclaves. Figure 2 shows a possible architecture of our proposal.

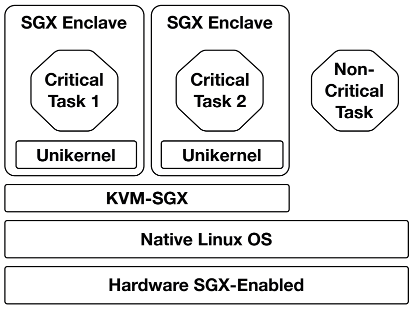

Fig. 2. Architecture of CRTES based on Intel SGX

The proposed approach is intended for only running critical tasks in SGX and the non-critical in the "normal" world. Another approach would have been to launch all the tasks in as many enclaves. However, unlike solutions based on ARM TrustZone, we need to take into account the memory limitation imposed by SGX technology, i.e., $128MB$, which is shared by all the enclaves. Hence, we believe that one enclave per critical task is the only feasible approach. Even in this case, launching too many critical tasks could entail that the memory bounds would be exceeded. A more accurate number highly depends on the *unikernel* solution adopted. For example, *IncludeOS* is a unikernel with one of the lowest memory footprint (i.e., $\approx 10MB$). Currently, *unikernels* footprints are approximately of $8MB$ on average [47]. This means that we could have a number of about 16 running critical tasks, which are in many cases above the real need for a CRTES.

Moreover, the adoption of SGX for CRTES can not disregard an analysis on the *isolation* properties. The *spatial* and *temporal isolation* are essentially ensured by SGX. The CPU realizes architectural isolation by monitoring the accesses to enclave-owned physical memory pages. The mechanism of enclave isolation is mainly provided through the *Enclave Page Cache Map* (EPCM). The EPCM is the table where the SGX processor checks the correctness of the system software's allocation decisions, and refuse to perform any action that would compromise SGX isolation. An EPCM entry identifies the enclave that owns the *Enclave Page Cache* (EPC) page, that is, the enclaves' content and the related data structures information. Since the EPCM identifies one specific owning enclave for each EPC page, enclaves can not communicate via shared memory using EPC pages. Enclaves only share untrusted non-EPC memory. SGX also prevents attempts from the Direct Memory Access (DMA) devices to get access to the enclave. This is of importance as current memory isolation techniques for CRTES are usually based on the MMU, which prevents applications running in one partition to read/write into address space allocated to other partitions. However, such an isolation *as-is* is threatened by the DMA that could bypass the checking procedure of the MMU. In a nutshell, the isolation may be put at risk when, e.g., the *unikernel* running in the enclave needs support from the external world or needs to communicate with another critical task executing in another enclave. In fact, there are some *unikernel syscalls* that must necessarily demand to the external OS. The implemented solution of an SGX-enabled CRTES should take into account the previous considerations. Solving mechanisms might be designed and developed to face the *isolation* of tasks and provide further guarantees in the certification phase.

IV. Certification implications

Regardless of the specific domain, the developing of safety-critical applications in industries raises several concerns from the certification point of view. In order to reach a specific level of safety, indicated as Safety Integrity Level (SIL) (but depending on the standards, SIL appears also as Automotive Safety Integrity Level (ASIL), Software Safety Integrity Level (SSIL), Design Assurance Level (DAL)), standards require performing burdensome tasks that include verification, performance testing, impact analysis, use of coding standards, on both the hardware and software components within the developed systems. Notwithstanding the cost and effort of certifying CRTES increases significantly with the SIL level, the problem is more exacerbated due to the integration of commercial off-the-shelf (COTS) hardware and software in the products. In general, compared to the bare-metal solutions, the use of virtualization brings additional software layers and components in the overall architecture, and this lead to further complicate the certification. Thus, providing evidence for isolation properties is far from to be effortless.

Considering the hardware and software stacks in the development process, we need to address the different level of safety required by the certification process, which lead to analyze isolation properties at different layers. For example, according to the EN 50128 standard that provides guidelines for the certification of the software employed in the railway domain [48], [49], focusing on the temporal and spatial isolation guarantees, the requirements `D.45 Response Timing and Memory Constraints` specifies that is needed an analysis aimed to estimate the resource usage and the latency for each system functionalities, which include all software modules (from the hypervisor to the kernel).

The strategy for developing CRTES based on virtualization and multi-core platform should be based on the guidelines provided by standards. The safety standards impose a precise development process (e.g., the IEC 61508 recommends using the V-model development process for designing safety-related software and hardware [36]), where we need to comply with further steps and requirements for covering safety and certification needs. Normally, such process should be based on the management of systems hazards, meaning that such hazards are eliminated or at least mitigated enough up to tolerable rates for the safety levels assigned to the identified safety functions. A potential approach for identifying the possible threats to safety is an analysis of the failure modes. For example, according to our proposal, we could apply the Failure Mode and Effect Analysis (FMEA), and try to identify the potential causes of failures by introducing disturbances (e.g., faults, anomalous loads) at the different levels of the CRTES software stack. Actually, many studies in literature [50]–[55] and various international standards for software reliability and safety [37], [39], [48], [56], [57] exploit the injection of faults in complex systems in order to assess their behavior and unveil potential bottlenecks and critical components under these abnormal inputs and conditions. Furthermore, fault injection technique is often used to measure the efficiency (e.g., coverage, latency, etc.) of fault tolerance mechanisms, including fault detection and recovery. For example, in the EN 50128 document is clearly stated that for the maximum level of safety (i.e., SIL 4) is highly recommended using Software Error Effect Analysis (SEEA) for identifying the criticality of each software component and improve the overall robustness of the software.

In combination with our proposal, there is a clear need for developing systematic approaches for testing the degree of isolation required by certification. In particular, by leveraging fault injection technique, an approach would be identifying and enumerating all the interfaces and resources involved at different levels of the CRTES software stack (e.g., hypervisor- and unikernel-level), to find suitable locations for injecting faults. Furthermore, it would be necessary defining measures of spatial and temporal isolation that could point out isolation issues; for example, a developer could use the existing performance isolation metrics [58], or adapt other metrics in the context of isolation [59].

Independently from the above considerations, it is important to underline that safety standards do not oblige developers or practitioners to use some particular measure or procedure for evaluating the fault tolerance, because they are not meant for exactly this or that target system. Instead, they suggest general guidelines that apply to a more wide family of systems.

V. Conclusion

In this position paper, we have discussed the adoption of both the Intel SGX hardware extension and the unikernel lightweight virtualization in the context of safety-critical real-time embedded systems. Our proposed idea is to leverage the isolation properties provided by SGX —typically used for security reasons in untrusted systems— in order to provide *temporal* and *spatial* isolation guarantees for critical tasks executing in the trusted execution environment of SGX. Potentially, our work could introduce the following advantages to CRTES:

i) the isolation among critical tasks is stronger thanks to the boundaries checks enforced by SGX hardware;
ii) the reliability increases as the risk of faults (e.g., memory leak) with *unikernel* is much more reduced, which is crucial for safety-related systems;
iii) the performance is higher as the *unikernel* model indeed offers performance improvements;
iv) the software certification process is facilitated thanks to *unikernels*' lightweight properties.

We want to remark that a fundamental requirement is the availability of the SGX hardware. Currently, the ARM architecture is widely used as 32b/64b RISC multi-core processors in embedded systems. However, we are witnessing the trend in the real-time embedded world of adopting virtualization to support multi-purpose OSes. In this regard, the most comprehensive virtualization support comes from Intel's architecture. Hence, the future usage of these CPUs is very likely, as witnessed by Intel itself [60].

Acknowledgment

This work has been founded by RFI - Rete Ferroviaria Italiana - within the Research Project "Analisi ed implementazioni prototipali di funzioni di interfaccia tra kernel e applicativi software di logiche di segnalamento", Grant reference number: DAC n. 552/2017– CIG 710174515F.

References

[1] M. Di Natale and A. L. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools," *Proceedings of the IEEE*, vol. 98, no. 4, pp. 603–620, 2010.
[2] A. Burns and R. I. Davis, "A Survey of Research into Mixed Criticality Systems," *ACM CSUR*, vol. 50, no. 6, p. 82, 2018.
[3] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in Real-Time Virtualization and Predictable Cloud Computing," *Elsevier Journal of Systems Architecture*, vol. 60, no. 9, pp. 726–740, 2014.
[4] ARM. TrustZone Technology for Microcontrollers. https://www.arm.com/why-arm/technologies/trustzone-for-cortex-m.
[5] M. Cereia and I. C. Bertolotti, "Asymmetric Virtualisation for Real-Time Systems," in *Proc. ISIE*. IEEE, 2008, pp. 1680–1685.
[6] H. Douglas, "Thin Hypervisor-based Security Architectures for Embedded Platforms," Ph.D. dissertation, Royal Institute of Technology, 2010.
[7] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig, "ARM TrustZone as a Virtualization Technique in Embedded Systems," in *Proc. RTLWS*, 2010, pp. 29–42.
[8] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves, "Implementing Embedded Security on Dual-Virtual-Cpu Systems," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 582–591, 2007.
[9] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: TrustZone is the Key," in *Proc. ECRTS*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
[10] J. Winter, "Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms," in *Proc. Wksp. STC*. ACM, 2008, pp. 21–30.
[11] M. Cereia and I. C. Bertolotti, "Virtual Machines for Distributed Real-Time Systems," *Elsevier Computer Standards & Interfaces*, vol. 31, no. 1, pp. 30–39, 2009.
[12] D. Sangorrin, S. Honda, and H. Takada, "Dual operating system architecture for Real-Time embedded systems," in *Proc. OSPERT*, 2010, pp. 6–15.
[13] S.-C. Oh, K. Koh, C.-Y. Kim, K. Kim, and S. Kim, "Acceleration of Dual OS Virtualization in Embedded Systems," in *Proc. ICCCT*. IEEE, 2012, pp. 1098–1101.
[14] 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 *Proc. ETFA*. IEEE, 2014, pp. 1–4.
[15] O. Schwarz, C. Gehrmann, and V. Do, "Affordable Separation on Embedded Platforms," in *Proc. TRUST*. Springer, 2014, pp. 37–54.
[16] M. Cinque, R. D. Corte, A. Eliso, and A. Pecchia, "RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets," in *Proc. ECRTS*. LIPICS, 2019, pp. 5:1–5:22.
[17] Frakti. Frakti GitHub page. https://github.com/kubernetes/frakti.
[18] Intel corp. Clear Linux project HomePage. https://clearlinux.org/.
[19] Kata Containers. HomePage. https://katacontainers.io.
[20] Hypercontainer. GitHub page. https://github.com/hyperhq/hyperd.
[21] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library Operating Systems for the Cloud," in *Proc. ASPLOS*. ACM, 2013, pp. 461–472.
[22] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Report 2016/086, 2016, http://eprint.iacr.org/2016/086.
[23] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proc. OSDI)*. USENIX Association, 2016, pp. 689–703.
[24] "SGX-LKL Library OS for Running Java Applications in Intel SGX Enclaves," https://github.com/lsds/sgx-lkl, last accessed 04/27/2018.
[25] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *Proc. ATC*. USENIX Association, 2017, pp. 645–658.
[26] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proc. OSDI*. USENIX Association, 2014, pp. 267–283.
[27] Mirage OS. MirageOS HomePage. https://mirage.io/.
[28] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014.
[29] D. C. van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum, "Towards a Flexible, Lightweight Virtualization Alternative," in *Proc. SYSTOR*, 2014, pp. 8:1–8:7.
[30] LXC. LXC - Linux Containers. https://linuxcontainers.org/.
[31] Docker Inc. Docker HomePage. https://www.docker.com/.
[32] OpenVZ. OpenVZ Main Page. http://openvz.org/Main_Page.
[33] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing Virtualization to the x86 Architecture with the Original VMware Workstation," *ACM TOCS*, vol. 30, no. 4, 2012.
[34] M. Cinque and A. Pecchia, "On the Injection of Hardware Faults in Virtualized Multicore Systems," *Elsevier Journal of Parallel and Distributed Computing*, vol. 106, pp. 50–61, 2017.
[35] J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam, and M. Sjödin, "Testing Performance-Isolation in Multi-core Systems," in *Proc. COMPSAC*, 2019, pp. 604–609.
[36] I. E. Commission, "Software Requirements," *IEC 61508-3*, 1998.
[37] ISO, "Product Development: Software Level," *ISO 26262: Road vehicles – Functional safety*, vol. 6, 2011.
[38] Aeronautical Radio Inc., "ARINC-653: Avionics application Software standard interface part 1," 2010.
[39] RTCA, "DO-178B Software Considerations in Airborne Systems and Equipment Certification," *Requirements and Technical Concepts for Aviation*, 1992.
[40] Certification Authorities Software Team (CAST), "Multi-core Processors," https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32a.pdf.
[41] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow," *IEEE Micro*, vol. 39, no. 3, pp. 66–74, 2019.
[42] MIKELANGELO. MIKELANGELO project HomePage. https://www.mikelangelo-project.eu/.
[43] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the Virtual Library Operating System," *Queue*, vol. 11, no. 11, p. 30, 2013.
[44] D. Cotroneo, L. De Simone, and R. Natella, "NFV-Bench: A Dependability Benchmark for Network Function Virtualization Systems," *IEEE TNSM*, vol. 14, no. 4, pp. 934–948, 2017.

[45] L. Coppolino, S. D'Antonio, G. Mazzeo, and L. Romano, "A Comparative Analysis of Emerging Approaches for Securing Java Software with Intel SGX," *Elsevier FGCS*, vol. 97, pp. 620 – 633, 2019.

[46] I. Sfyrakis and T. Gross, "UniGuard: Protecting Unikernels Using Intel SGX," *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 99–105, 2018.

[47] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate IoT edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, 2018.

[48] CENELEC, "EN 50128," *Railway applications-Communication, Signaling and Processing Systems-Software for Railway Control and Protection Systems*, 2011.

[49] G. Mazzeo, L. Coppolino, S. D'Antonio, C. Mazzariello, and L. Romano, "SIL2 Assessment of an Active/Standby COTS-based Safety-Related System," *Elsevier Reliability Engineering & System Safety*, vol. 176, pp. 125–134, 2018.

[50] D. Cotroneo and R. Natella, "Fault Injection for Software Certification," *IEEE Security & Privacy*, vol. 11, no. 4, pp. 38–45, 2013.

[51] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental Analysis of Binary-level Software Fault Injection in Complex Software," in *Proc. EDCC*. IEEE, 2012, pp. 162–172.

[52] D. Cotroneo, L. De Simone, and R. Natella, "Run-Time Detection of Protocol Bugs in Storage I/O Device Drivers," *IEEE TR*, vol. 67, no. 3, pp. 847–869, 2018.

[53] D. Cotroneo, A. Lanzaro, and R. Natella, "Faultprog: Testing the Accuracy of Binary-level Software Fault Injection," *IEEE TDSC*, vol. 15, no. 1, pp. 40–53, 2016.

[54] D. Cotroneo, L. De Simone, and R. Natella, "Dependability Certification Guidelines for NFVIs through Fault Injection," in *Proc. ISSREW*. IEEE, 2018, pp. 321–328.

[55] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "No PAIN, no gain?: the utility of PArallel fault INjections," in *Proc. ICSE*. IEEE Press, 2015, pp. 494–505.

[56] NASA, "NASA Software Safety Guidebook," *NASA-GB-8719.13*, 2004.

[57] RTcA, RTCA DO, "ISO/IEC 25045," *Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Evaluation module for recoverability*, 2010.

[58] R. Krebs, C. Momm, and S. Kounev, "Metrics and techniques for quantifying performance isolation in cloud environments," in *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, ser. QoSA '12, 2012, pp. 91–100.

[59] M. Hiller, A. Jhumka, and N. Suri, "An approach for analysing the propagation of data errors in software," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2001, pp. 161–170.

[60] Intel corp. Real-Time Systems and Intel Take Industrial Embedded Systems to the Next Level. https://software.intel.com/en-us/articles/real-time-systems-and-intel-take-industrial-embedded-systems-to-the-next-level.