embeddedworld2020
Exhibition&Conference
... it's a smarter world

Bao: a modern lightweight embedded hypervisor

José Martins
Universidade do Minho
Guimarães, Portugal
jose.martins@dei.uminho.pt

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

***Abstract*—Virtualization is already a key-enabling technology for mixed-criticality embedded systems. Open-source hypervisors such as KVM or Xen were not originally tailored for embedded constraints and real-time requirements, and depend on Linux, resulting in large TCBs and wide attack-surfaces. Furthermore, they do not address the numerous microarchitectural contention points and side-channels that have been shown to break true VM isolation. Bao is a lightweight, open-source embedded hypervisor which aims at providing strong isolation and real-time guarantees. Similarly to Jailhouse, it is a partitioning hypervisor leveraging hardware virtualization support; unlike Jailhouse, it is completely self-sufficient, not depending on Linux. Currently supporting ARMv8 and RISC-V, Bao was developed from scratch to provide a minimal clean-slate and industry-grade solution and to engage both academia and industry in tackling the challenges of modern automotive and industrial systems.**

Keywords— Virtualization, separation, hypervisor, static partitioning, safety, security, real-time, embedded systems, Arm, RISC-V.

I. INTRODUCTION

Virtualization is already a well-established technology in enabling supervised mixed-criticality in modern high-performance embedded systems. Its ubiquity has been established in fields such as automotive and industrial automation as these are pushing towards a consolidation of multiple subsystems while meeting increasingly tight size, weight, power and cost (SwAP-C) constraints.

In the open-source arena, hypervisors such as KVM [1] or Xen [2] are the go-to solutions. Originally designed targeting the server domain, these hypervisors have been successfully ported to embedded architectures (mainly Arm [3][4]). However, they are not fully suited for the tight embedded constraints and real-time requirements as support for them was implemented as an add-on and not by design. Furthermore, from a security and safety standpoint, they comprise large trusted computing bases (TCBs) as they often depend on large monolithic operating systems (OSes), typically Linux, running on privileged virtual machines (VMs) which management services for the remaining VMs. For example, Xen relies on a privileged VM, called Dom0, to manage non-privileged VMs (DomUs) and interface with peripherals. Moreover, these hypervisors frequently provide a rich set of virtualization services (e.g. virtual networks) through emulation or para-virtualization mechanisms, increasing the number of hypercalls and resulting in a bloated TCB and wide attack surface [5].

Recently, the static partitioning hypervisor architecture, a design recently best embodied by Siemens' Jailhouse [6], has emerged as a viable solution to meet the needs of modern mixed-criticality systems. This architecture follows a minimalist approach, similar to separation kernels. It statically partitions all system resources (i.e., memory, peripherals and CPUs) at initialization time, by directly and exclusively assigning each one to a single partition. The practically nonexistent runtime memory allocation and the physical-to-virtual CPU assignment through a 1:1 mapping, preclude the need for a virtual CPU scheduler. In addition, by leveraging hardware virtualization extensions (available on the majority of modern processor architectures), this approach results in a minimal TCB and virtualization overhead (e.g. VM interrupt latency, VM boot time). Despite its design philosophy, Jailhouse stills incurs on some of the drawbacks of other hypervisors, by depending on a privileged Linux VM, its "root cell", to boot the system and manage other "cells". Nevertheless, given the proven benefits of such an approach, other hypervisors are being extended to provide similar features and benefits. Recently, efforts headed by Xilinx, endowed Xen with Dom0-less execution support [7], allowing VMs to directly access peripherals, boot without Dom0 (drastically reducing boot time), or completely discard it, resulting in full-blown static partitioning.

In spite of the high degree of isolation and determinism guarantees provided by the static partitioning architecture, there are still safety and security issues not addressed by this design. First, in the past few years, the research community has highlighted a number of microarchitectural structures (e.g. last-level caches – LLCs, memory controllers) that remain shared. These act as contention points and implicitly give rise to timing side-channels, compromising (i) determinism, by increasing jitter on access to such resources, (ii) confidentiality, through common cache attack techniques (e.g. *Prime+Probe*), and (iii) availability, as they make VMs susceptible to denial-of-service (DoS) attacks by comprised guests [8]. Cache partitioning techniques (e.g. cache coloring) or memory bandwidth reservations have been proposed to mitigate these issues and implemented in hypervisors showing promising results [9][10]. Second, static partitioning does not directly support multiple isolated trusted execution environments (TEEs), without explicitly dedicating physical CPUs to execute them. TEEs are

an essential component in modern high-performance embedded systems running user-facing OSes such as Android. As TEEs are typically supported by hardware security technologies such as Arm's TrustZone [11], many solutions have been proposed to in some way virtualize the so-called, secure world [12][13]. However, given the prevailing security vulnerabilities in TrustZone-based TEEs [14] suggest this dual-world hardware-enforced isolation is not, per se, an end-all solution. Also, it does not seem to be an inherently more secure approach than using dedicated normal world VMs to execute trusted services. This is the approach followed by Hafnium [15], a security-focused hypervisor targeting Android on IoT and mobile devices. Hafnium supports a single primary VM in addition to a number of lightweight secondary VMs, where the latter are only scheduled by request of the former. Secondary VMs are thought as isolated security domains and meant to provide services to the primary VM.

In this paper, we present Bao, a from-scratch implementation of the static partitioning architecture. Bao was originally developed to serve as a minimal base scaffold to conduct research on, and deepen VM security. We hope that, by opening up Bao's code base, we engage both academic and industry communities on tackling these issues. The paper starts by highlighting Bao's design and implementation principles and then proceeds to provide a short evaluation of its TCB and virtualization overheads.

II. BAO OVERVIEW

Bao (from Mandarin Chinese "bǎohù", meaning "to protect") is a security and safety-oriented, lightweight bare-metal hypervisor. Its design targets mixed-criticality systems, and, as such, is centered on providing fault-containment and real-time behavior. Given its suitability to the target systems, Bao initially implements the static partitioning architecture (Figure 1) where resources are statically partitioned and exclusively assigned to each VM: (i) memory is allocated only at initialization time; (ii) guest IO is pass-through only; (iii) virtual interrupts are directly mapped to physical ones; and (iv) virtual CPUs (vCPUs) are assigned to physical CPUs (pCPUs) following a 1:1 mapping, precluding the need for a scheduler. As in such systems VMs often have the need to interact with each other, the hypervisor also provides simple primitives for inter-VM communication. This mechanism is based on a static shared memory and asynchronous notifications in the form of inter-VM interrupts triggered through an hypercall. Furthermore, and similarly to Hafnium, Bao extends static partitioning to allow multiple vCPUs to execute isolated security functionality in the same pCPU although maintain the partitioning semantics between different criticality subsystems (see Section II.C). Besides standard platform management firmware, Bao has no dependency on external libraries or on privileged VMs running untrustable, large monolithic OSes.

A. Design and Implementation Principles

Bao is designed around a core set of principles which guide its implementation and future direction:

1. ***Minimality and Simplicity***. The code base strives to be as minimal and simple as possible. As such, Bao is implemented only in architectures which provide hardware-assisted virtualization. Taking advantage of mechanisms such as guest-dedicated privilege levels, 2-stage address translation, and IOMMU support, precludes the use of high-overhead and complex techniques such as trap-and-emulation and shadow-page tables. This significantly reduces virtualization overheads and the system's TCB by minimizing code size and complexity. This principle is also applied to Bao's hypercall interface which should only provide essential services, with low complexity semantics.
2. ***Least Privilege***. The implementation strives to ensure that each component in the system has access only to what it absolutely must. Each core has a private address space, only mapping the physical pages it needs. As such, each core only maps the VM and vCPU structures it hosts, not being able to access VM information belonging to different partitions. More importantly, the hypervisor is not able to directly access VM physical memory which mandates that all hypercall arguments passed by value in processor registers and never by reference. Finally, only the essential virtualization mechanisms execute at the hypervisor's privilege mode, and all other functionality must be migrated to VMs.
3. ***Thorough Isolation***. Despite the straightforward logical isolation provided by static virtualization, VMs still interact through shared micro-architectural state. One of Bao's main goals is to implement mechanisms to tackle this issue. As a first step, and given the simplicity of the mechanism, cache-coloring is ingrained in the hypervisor's physical page allocation mechanism which takes into account the colors assigned to a given VM. Also, the hypervisor itself can be configured to only use certain colors. However, this technique has several drawbacks including, for example, memory fragmentation.

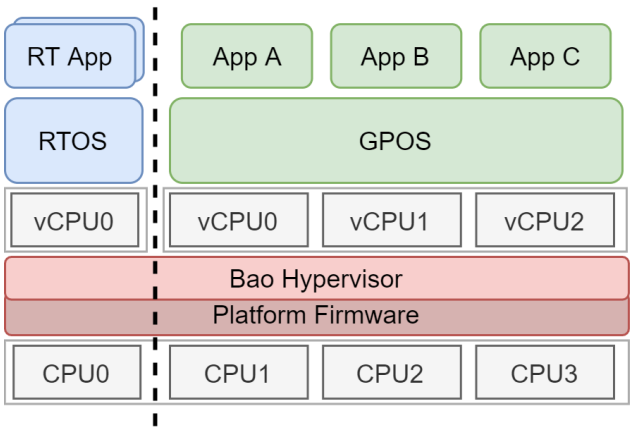

Figure 1. Bao Static Partitioning Architecture featuring a dual-guest configuration,

B. Platform Support

Bao targets only 64-bit architectures. It currently supports Armv8. RISC-V experimental support is also available but, since it depends on the hypervisor extensions, which are not yet ratified, no silicon is available that can run the hypervisor.

Consequently, the RISC-V port was only deployed on the QEMU emulator, which implements the latest version of the draft specification (at the time of this writing, version 0.5). Focusing on Arm platforms, at the time of writing, Bao was ported to two Armv8 platforms: Xilinx's Zynq-US+ on the ZCU102/4 development board and HiSilicon's Kirin 960 on the Hikey 960. So far, Bao was able to host several bare-metal applications, the FreeRTOS and Erikav3 RTOSs, and Linux and Android.

Besides simple serial drivers to enable the log output of its activity, Bao does not rely on platform-specific device drivers. To be ported to a new platform only a simple description detailing the number of available CPUs, available memory, and its location, is needed. For this reason, Bao relies on vendor-provided firmware and/or a generic bootloader to perform low-level hardware initialization, management, and hypervisor and guest image loading. On the supported Arm-based platforms, Bao relies on an implementation of the standard Power State Coordination Interface (PSCI) to perform low-level power control operations, further avoiding the need for platform-dependent drivers. This has been provided by Arm Trusted Firmware (ATF). On such platforms, Linux itself depends on PSCI for CPU hot-plugging. When such guests invoke PSCI services, Bao merely acts as a shim and sanitizer for the call arguments, to guarantee the VM abstraction and isolation, deferring the actual operation to ATF. Although we've been able to boot directly from ATF, we've been also using the well-known U-boot bootloader to load hypervisor and guest images.

In the Arm architecture the GIC (Generic Interrupt Controller) is the main interrupt arbiter and router, which is composed by a central distributor and per-CPU interfaces. In the currently supported platforms, the available GICv2 provides some virtualization support. However, all interrupts are still forward to the hypervisor, which must re-inject the interrupt in the target VM. Furthermore, although the CPU interfaces are completely virtualized by the hardware, access to the distributor must be achieved using a trap-and-emulation approach. The newer GICv4 will provide direct interrupt delivery to VMs. Another peripheral essential for virtualization is the SMMU (Arm's IOMMU). The SMMUv2 available in the supported platforms has several limitations: a limited of currently active stream translation registers which limits the number of simultaneously active DMA-capable devices. Although multiple devices can be grouped in one of these registers, these groupings cannot be arbitrary. It would be possible to context-switch these registers as peripherals issued memory transactions, but this goes against the static nature of resource distribution, and would increase code complexity and severely hurt determinism (as it would be a shared hardware structure). Keeping it simplicity philosophy, at initialization time, Bao checks if the existing number of available registers is sufficient to fulfill the device assignment defined in the configuration, halting if this is not possible. Newer versions of the SMMU spec address these shortcomings.

C. TEE Support

Bao expands on the static partitioning architecture by allowing a N:1 mapping of virtual to physical CPUs, although a vCPU is always pinned to a single pCPU. For each VM in the configuration, it is possible to define a set of auxiliary VM. Auxiliary VMs are meant to allow multiple isolated environments to execute in a single hardware partition. To keep in line with its minimality design principal, no scheduler is added: a vCPU is only scheduled when a currently active vCPU explicitly invokes of one of its auxiliary VMs and, later, the invoked vCPU can yield execution only to its caller. Both operations are issued through two simple hypercalls with very low-level semantics. Further, no vCPU migration exists: only vCPUs assigned to the same pCPU can be invoked. We call this process vCPU or VM stacking, as vCPUs are "scheduled" in a FIFO fashion, in the same way stack frames are created and destroyed on the stack during procedure call and returns. Furthermore, while a given vCPU is executing, if an interrupt targeting a vCPU deeper in the stack is triggered, the stack is unwound, immediately handling execution to the target vCPU. This mechanism effectively builds multiple levels of privilege inside a single partition, as a VM can always preempt its auxiliary VMs.

Note that Bao distributes pCPUs to a set of "root" VMs during system initialization and effectively partitions the available CPUs as typical of static partitioning hypervisors. To clarify, if a quad-core system is configured to execute an RTOS in one of the cores, while running a GPOS and an isolated TEE in the remaining cores, using this mechanism, the RTOS is not affected by it and its physical core is always fully dedicated to this guest. Furthermore, note that, by default, even VMs in the same partition do not share memory. This can be accomplished using the same shared memory mechanism available for inter-partition communication.

VM-stacking allows the execution of widely-used TEEs such as OP-TEE with little to no modifications, depending on how the VM hierarchy is structured. As shown in Figure 3, the first option consists in using a "monitor" VM scheduling both the OS and TEE VMs mimicking the dual-world architecture of TrustZone. A second option illustrated in Figure 2 would be to have a higher privilege TEE VM, while the OS would use the yield hypercall to invoke secure services.

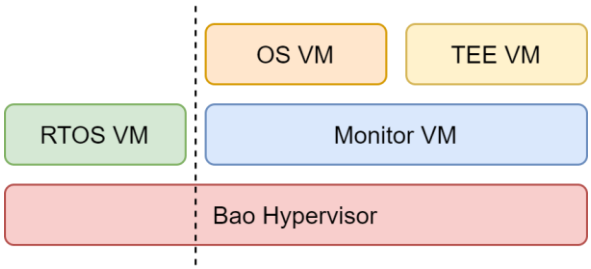

Figure 3. TEE support mimicking TrustZone's dual-world model.

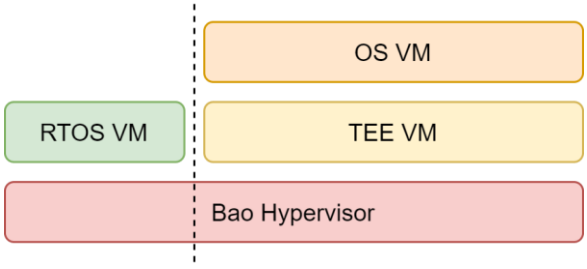

Figure 2. TEE support using a higher privilege TEE VM.

III. EVALUATION

This evaluation targeted the Xilinx ZCU104 board, featuring a Zynq-US+ SoC with a quad-core Cortex-A53 running at 1.2 GHz, per-core 32K L1 data and instruction caches, and a shared unified 1MB L2/LLC cache. Bao was compiled using the Arm GNU Toolchain version 8.2.1 with -O2 optimizations.

A. *Trusted Computing Base*

We evaluate the TCB using source lines of code (sLoC) and binary size as metrics. Table I shows the number of C, assembly, and total lines of code. Table II shows the final binary size, by section, for building Bao for the target platform. The total ~5.6 KSLoC and ~59 KiB final binary reflect the low complexity and small TCB achieved by Bao's implementation.

TABLE I. Source Lines of Code (SLoC)

C	Assembly	Total
5154	447	5601

TABLE II. Binary Size (bytes)

.text	.data	.bss	.rodata	total
39956	1192	17045	1341	59535

B. *Performance Overhead*

The MiBench Embedded Benchmark Suite [16] automotive subset was used to evaluate the virtualization performance overhead of a Linux guest running over Bao. Figure 5 shows the obtained results. The resulting overheads range from negligible to a maximum of about 2%. Given the simplicity of the implemented virtualization mechanisms, we believe these overheads are mainly due to 2-stage address translation.

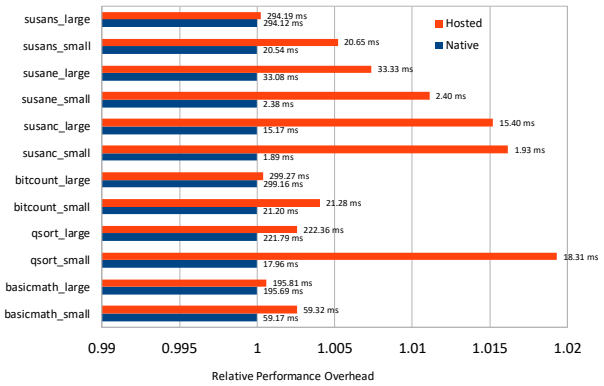

Figure 5. Performance Virtualization Overheads on the MiBench Automotive Benchmarks.

C. *Interrupt Latency*

To measure interrupt latency, we use a custom bare-metal guest which continuously calculates the delay observed on a timer interrupt programmed at a frequency of 100 Hz. Comparing native to hosted executions, Table III shows that the average latency increases by about 430 ns and standard-deviation by about 40.5 ns. This shows the significant overhead that is imposed by GICv2 given the mandatory interrupt re-injection. Furthermore, worst-case latency is also significantly increased. As this value was observed only on the first measurement of our experiments, we believe this is the results of compulsory cache misses on the instruction cache for the hypervisor code used to inject the interrupt, which suggests that other guest execution might significantly affect a VM interrupt latency by forcing the eviction of these cache lines. Although we have not yet verified this empirically, we believe by using cache coloring to dedicate a cache partition to the hypervisor, this effect can be minimized.

TABLE III. Interrupt Latency (ns)

	Average	Std. Dev.	Min.	Max.
Native	140.4	11.1	140.0	490.0
Hosted	571.64	50.63	560.0	2170.0

D. *Interference*

To verify the correctness and effectiveness of the cache coloring mechanism on avoiding VM interference through the LLC, we take a security perspective and employ a simple cache *Prime+Probe* from one of the VMs, while varying the number of cache lines accessed by a "victim" VM. Figure 4 depicts the channel matrix [17] for the assessed channel. This heat map represents the probability for measuring a given probe time, when a certain number of cache lines is accessed. The observed diagonal trend shows that the number of lines accessed by the victim can easily be inferred given the probe time.

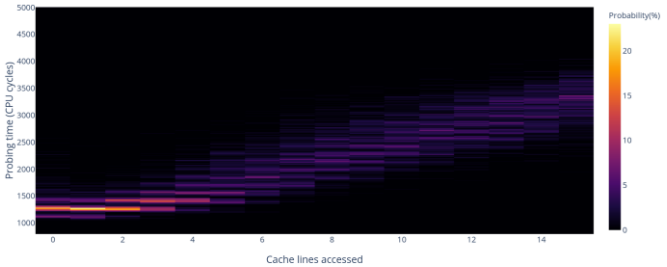

Figure 4. Last-level cache channel matrix between two VMs with no cache partitioning.

We repeat the experiment with coloring enabled, assigning half the LLC to each of the VMs. The results Figure 6 presents the resulting channel matrix which shows no variation on the probe time, independently of the number of lines accessed by the victim. This confirms the effectiveness of cache coloring in partitioning the LLC and avoiding information leakage between VMs through the it.

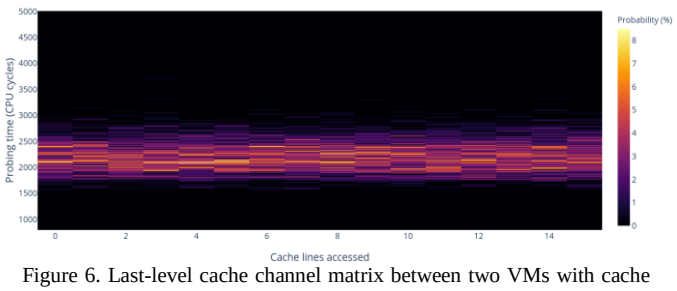

Figure 6. Last-level cache channel matrix between two VMs with cache partitioning through coloring enabled.

IV. Conclusion

In this paper, we have presented Bao, a from-scratch implementation of the static partitioning hypervisor. Our evaluation shows that Bao encompasses a small TCB and low-degree of virtualization overhead. Furthermore, it features cache partitioning mechanisms and we plan to expand it to address other known, or possibly uncovered, sources of contention. It also supports the execution of multiple isolated TEEs of each of the main guest OSes. Bao source code is open [18] in hopes of engaging both the academic and industrial communities in tackling the challenges of virtual machine isolation in modern high-end embedded systems.

Acknowledgments

José Martins has been supported by FCT grant SFRH/BD/138660/2018. We would like to thank the people of HipertLab (Università di Modena e Reggio Emilia, Italy), especially Marko Bertogna, Marco Solieri, and Angelo Ruocco for their valuable input and assistance in Bao's development.

References

[1] U. Lublin, Y. Kamay, D. Laor, and A. Liguori. KVM: the Linux virtual machine monitor. In Proceedings of the Linux Symposium, 2007.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, New York, NY, USA, 2003.

[3] J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, and C. Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In IEEE Consumer Communications and Networking Conference, pages 257–261, 2008.

[4] C. Dall and J. Nieh. KVM/ARM: the design and implementation of the linux ARM hypervisor. In ACM SIGARCH Computer Architecture News, vol. 42, no. 1, pp. 333-348. ACM, 2014.

[5] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security Through Disaggregation. In Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '08, pages 151–160, New York, NY, USA, 2008. ACM.

[6] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look Mum, no VM Exits!(Almost). In Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), 2017.

[7] S. Stabellini. True Static Partitioning With Xen Dom0-Less. Xen Project. 2020. URL: https://xenproject.org/2019/12/16/true-static-partitioning-with-xen-dom0-less/.

[8] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. In Journal of Cryptographic Engineering 8:1–27, 2018.

[9] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2019.

[10] P. Modica, A. Biondi, G. Buttazzo, and A. Patel. Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms. In 2018 IEEE International Conference on Industrial Technology (ICIT), pages 1651–1657, 2018.

[11] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. ACM Comput. Surv., 51(6):130:1–130:36, January 2019.

[12] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM TrustZone. In 26th USENIX Security Symposium (USENIX Security 17), pages 541–556, Vancouver, BC, August 2017. USENIX Association.

[13] G. Cicero, A. Biondi, G. Buttazzo, and A. Patel. Reconciling security with virtualization: A dual-hypervisor design for ARM TrustZone. In 2018 IEEE International Conference on Industrial Technology (ICIT), pages 1628–1633, 2018.

[14] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In IEEE Symposium on Security and Privacy (S&P), Los Alamitos, CA, USA, to appear 2020.

[15] Hafnium. The root Hafnium repo, 2019. URL: https://hafnium.googlesource.com/hafnium/.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), pages 3–14, 2001.

[17] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. 2014. The Last Mile: An Empirical Study of Timing Channels on seL4. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14). Association for Computing Machinery, New York, NY, USA, 57.

[18] Bao Hypervisor. Bao project repo. 2020. URL: https://github.com/bao-project/bao-hypervisor