Snape: The Dark Art of Handling Heterogeneous Enclaves

Zahra Tarkhani
University of Cambridge, UK
zahra.tarkhani@cl.cam.ac.uk

Anil Madhavapeddy
University of Cambridge, UK
anil.madhavapeddy@cl.cam.ac.uk

Richard Mortier
University of Cambridge, UK
richard.mortier@cl.cam.ac.uk

Abstract

Code executing on the edge needs to run on hardware platforms that feature different memory architectures, virtualization extensions, and using a range of security features. Forcing application code to conform to a monolithic API such as POSIX, or ABI such as Linux, ties developers into large, complex platforms that make it difficult to use such hardware-specific features effectively as well as coming with their own baggage and the attendant security issues. As edge computing proliferates, handling increasingly sensitive and intimate data in our everyday lives, it becomes important for developers to be able to use *all* the hardware resources of their particular platform, correctly and efficiently.

To this end, we propose Snape, an API and composable platform for matching applications' needs to the available hardware features in a heterogeneous environment. Unlike existing solutions, Snape provides applications with a flexible trust model and replaces untrusted host OS services with corresponding hw-assisted secured services. We report experience with our proof-of-concept implementation that enables Solo5 unikernels on Raspberry Pi 3 boards to make effective use of ARM TrustZone security technology.

CCS Concepts • **Security and privacy** → **Virtualization and security**; *Systems security*; Security in hardware; • **Computer systems organization** → *Cloud computing*.

Keywords unikernels, enclaves, secure execution

ACM Reference Format:
Zahra Tarkhani, Anil Madhavapeddy, and Richard Mortier. 2019. Snape: The Dark Art of Handling Heterogeneous Enclaves. In *2nd International Workshop on Edge Systems, Analytics and Networking (EdgeSys '19), March 25, 2019, Dresden, Germany.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3301418.3313945

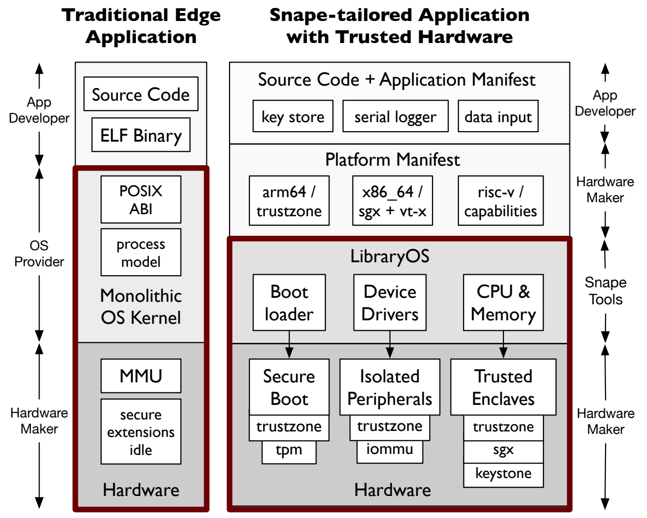

Figure 1. Snape vs Traditional application layout

1 Introduction

Computing at the edge is proliferating and, due to high variability in the deployment environment and application demands, the hardware available is highly heterogeneous. With growing awareness of the likelihood and cost of security and privacy breaches, many such hardware platforms now support one of several different security features that can be used to create a secure execution environment for applications, e.g. encrypted memory [3, 13] or secure peripherals [4]. These provide extra layers of protection to prevent attackers, who may have full access to the hardware and privilege software, from accessing confidential application data once the host operating system (OS) or hypervisor is compromised. Table 1 gives some examples of recent CVEs issued against common host OSs and hypervisors where a malicious host OS or hypervisor could gain access to private application data or code.

Given the significant amount of research into security hardware features for protecting applications on untrusted hosts [6, 9, 10, 12, 14, 20], it is likely that the heterogeneity of edge hardware will continue to increase. At the same time, most of these hardware platforms expose their features through relatively large, monolithic, inflexible Trusted Computing Bases (TCBs), whether in the form of a hypervisor or an in-enclave library OS (LibOS). For example, *Haven* [6] and *trustshadow* [10] show how unmodified applications can be executed inside a secure enclave either by linking against an in-enclave libOS or using in-enclave mechanism for trap-and-verifying untrusted system calls. As the number of different hardware features continues to grow, we believe this approach is not a long term solution and will lead to unresolved security, efficiency, and compatibility problems that will be particularly acute at the edge.

We propose an alternative: *Snape* is a declarative platform mapping applications' fine-grained security requirements to available hardware features. Figure 1 depicts Snape's high-level architecture: applications need not trust the underlying

EdgeSys '19, March 25, 2019, Dresden, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6275-7/19/03...$15.00
https://doi.org/10.1145/3301418.3313945

execution environment as a monolithic whole. Instead, by configuring a manifest that expresses their trust requirements, they can choose (*i*) the data objects they want to protect, (*ii*) the OS components that need to be protected, and (*iii*) the isolation level of trusted services they will rely upon. Snape compartmentalizes sensitive components in separate enclaves to provide protection against breach of in-enclave compartments.

In this paper, after giving the motivation and background to Snape (§2), we describe the design of Snape and in particular the threat and trust models we assume (§3). Snape provides fine-grained utilization of the underlying hardware security features for security-critical applications. Our approach allowing applications to easily manage, limit, or extend their trust depending on available heterogeneous hardware. We then present our proof-of-concept (PoC) implementation that contains a tiny microkernel-style hypervisor or unikernel monitor, and uses ARM TrustZone to offer different levels of isolation for applications enabling developers to appropriately balance their security and performance requirements (§4). Finally, we present some initial evaluation of Snape using ARM TrustZone on Raspberry Pi 3 boards (§5) before discussing related work (§6) and concluding (§7). In doing so we show how performance overhead depends on the Snape users' trust model and isolation levels and so it should be possible for the user to control that balance.

2 Motivation & Background

We now discuss the motivations behind Snape and the necessary technical background.

2.1 The need for fine-grained flexible trust

Current efforts on building OS and runtime support for hardware-enclaves shows the limitations and differences in the various implementations, and designing an efficient and secure software abstraction for these enclaves is challenging. Snape's finer-grained security model is motivated by many unresolved issues in previous work [6, 10, 23].

- **Handling heterogeneity:** Snape does not have a single inflexible monolithic TCB and contains relatively standalone security services depending on the host hardware features. For example, some application modules such as in-enclave file systems can be supported on more hardware platforms than other modules such as secure peripherals. But providing a lightweight declarative model is the first step to secure execution on the heterogeneous environment
- **Fragile security:** The approach of shielding entire application inside a protected environment result in a large complicated TCB that includes the language runtime[1] and most of the applications' dependencies[2] that may not even have access to sensitive information. Porting these dependencies into the isolated environment reduces the number of interfaces to the untrusted world. On the other hand, this makes the secure world more and more similar to the untrusted world; a single breach and it is vulnerable to a variety of attacks [1, 2] that fails in exactly the way our current commodity OS fails. Also, there is no way of limiting access from one sensitive module to another one, or any mechanism for limiting the scope of damage when a breach occurs and the enclave is no longer fully trusted. Snape is motivated by the fact that hardware-enforced compartmentalization of underlying TCB can resolve some of these security issues[7].
- **Inefficiency & fixed performance overhead:** An inflexible trust model forces applications to only use the limited resources inside the enclave–e.g. insufficient protected memory, or inefficient disk IO, networking access and multithreading. This is especially bad form when an application only needs to trust a small part of the host OS, for example, an application that needs a trusted filesystem should not incur a large overhead on networking or UI.

[1] https://github.com/oscarlab/graphene/issues/396
[2] https://github.com/oscarlab/graphene/issues/325

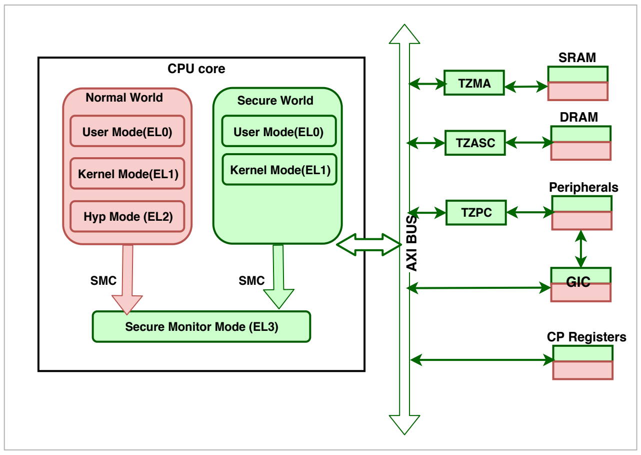

Figure 2. TrustZone in ARMv8-A systems:

2.2 ARM TrustZone overview

Our prototype uses ARM trustZone because of its rich support of secure hardware features including secure IO path and peripherals that helps to consider various scenarios of applications requirements.

As a short description of the ARM security extensions [5] or TrustZone, this System-On-Chip (SoC) security mechanism introduced with ARMv6 in most ARM Cortex-A and Cortex-M processors. The TrustZone hardware architecture can be seen as a dual-virtual system that allows partitioning of all hardware resources including the CPU, memory, and peripherals into two execution environments (see Figure 2). The resource partitioning to secure and non-secure modes (S/NS) can be done using TrustZone AXI components. For example, the TrustZone Address Space Controller (TZASC) partitions a single AXI slave such as an off-SoC DRAM, while TrustZone Memory Adapter (TZMA) is used for partitioning on-SoC memory such SRAM. The TrustZone Protection Controller (TZPC) signal configures peripherals. The Generic Interrupt Controller (GIC) supports both Secure and Non-secure prioritized interrupt sources. Both worlds have their

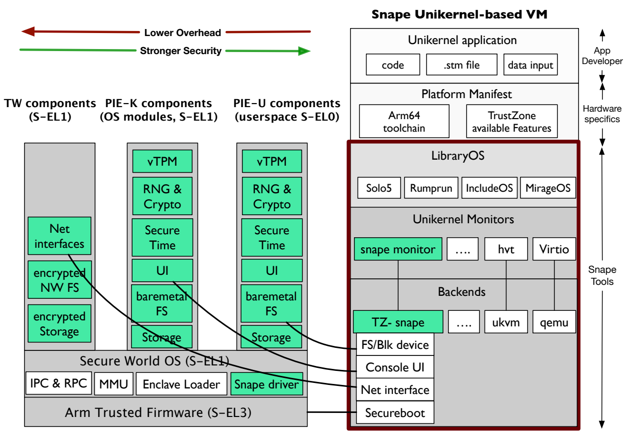

Figure 3. An ARM64-based example of Snape architecture

CVEs	guest VMs	Snape VMs
Host malicious OS:		
CVE-2018-8781	yes	no
CVE-2017-1000251	yes	no
CVE-2017-17712	yes	no
CVE-2017-13715	yes	no
Host malicious hypervisor:		
CVE-2018-14678	yes	no
CVE-2017-12188	yes	no
CVE-2017-12137	yes	no
CVE-2017-10918	yes	no
Channel attacks:		
CVE-2017-5753	yes	yes
CVE-2017-5715	yes	yes
CVE-2017-5754	yes	yes

Table 1. A representative selection of recent vulnerabilities that can cause a malicious OS/hypervisor to access private data from the guest VMs, but can be prevented on the Snape architecture

own userspace and kernel space, and only the normal world has `hyp` mode[3]. The two worlds can communicate to each other under the strict supervision of a Secure Monitor running in monitor mode (EL3) via a "Secure Monitor Call" (SMC) instruction.

3 Snape overview

Figure 3 shows an example of a Snape application on ARMv8 platforms that choose various private OS features with a different level of isolation. The application requests: (*i*) a protected filesystem that runs as a standalone binary in secure world userspace; (*ii*) a secure console UI that runs in secure world kernel mode; and (*iii*) a trusted interface around the underlying networking stack for accessing `tap` devices.
Snape is a unikernel-based VM that runs on top of a tiny hypervisor or unikernel monitor that enforces memory isolation and provides hardware resources such as virtual CPU, disk IO, and networking via a small number of hypercalls [18, 25]. By default, the hypervisor may depend on a commodity OS for providing these services. For instance, Solo5 hvt monitor [25] depends on the host Linux kernel for creating block storage or accessing tap devices for networking. The underlying libraryOS provides necessary OS functionality and is statically linked to the application as a unikernel image which will be loaded by the hypervisor. If an application does not have any sensitive data, it would be executed with default monitor backend that depends on the untrusted host OS/hypervisor. However, when an application has private data that needs to be protected against the host OS/hypervisor, the application developer needs to specify its trust model by configuring Snape Trust Model (`.stm`) file (§3.3). Then, depending on the trust model, Snape initializes the requested enclaves and maps the required services to dedicated enclaves and does not change anything for non-sensitive calls. Snape provides essential trusted services and depends on the hardware support to provide different levels of isolation for the trusted modules (§3.2). As demonstrated in Figure 3, Snape contains a small OS running in the TrustZone secure world that manages enclaves resources, runs enclaves, and handles secure interactions between enclaves and to the outside world.

[3]Secure EL2 support is introduced in Armv8.4, which provides virtualization support inside the secure world

3.1 Threat Model

We assume adversaries that can gain full control over the host OS and hypervisor. The malicious host OS or hypervisor can execute arbitrary code and access VM data (see Table 1). We assume the secure world minimal OS or runtime is carefully verified, preinstalled and loaded with a secure boot mechanism. The minimal OS loads all enclaves statically and no dynamic loading is allowed. Snape covers against Iago attacks [8] from the host compromised OS. Recent study shows practical side channel attacks against enclaves [16, 17, 26]. These attacks are out of scope in this paper.

3.2 Snape fine-grained security services

In this section, we explain how Snape security services are protected from the untrusted host and how it supports different levels of isolation on TrustZone-enabled ARMv8 platforms.

- **Pure In-enclave (PIE) Services:** Each PIE services implemented as a baremetal component that compiles into separate enclave binaries without any dependencies to the untrusted world. Depending on the available secure hardware, these services can be PIE storage, filesystem, user interface, system time, randomness, and cryptography. As Figure 3 shows, the difference between PIE-U and PIE-K is the level of isolation between enclaves and execution in either secure world user mode or kernel mode. PIE-U services that run in secure world user mode (SEL0) are in fact, isolated processes through the secure world MMU. While PIE-K services are executing as secure world drivers (SEL1). So it is clear that PIE-U provides stronger security and largest overhead due to extra context switches to usersapce.

- **Trusted Wrapper (TW) Services:** These services follow a forward-and-verify approach similar to TrustShadow [10] for providing trusted OS system calls, or Nizza [11] that provides trusted wrappers around device drivers for secure reuse of legacy drivers. Snape supports TW filesystem and networking wrappers around the Linux kernel to protect the confidentiality and integrity of private data by on-the-fly data encryption inside enclave before passing it to the untrusted host. TW components act as a proxy between application and untrusted host, and verify inputs and outputs of untrusted calls. These services may provide better performance and richer functionality, but arguably weaker security than PIE equivalents.

The performance overhead of Snape depends on the trust model and isolation type of trusted services as we will discuss in Section 5.

3.3 Snape trust model

Snape requires application developers to build a trust model via an `stm` manifest. Unlike systems like ProxOS [24], there is no need to specify all private system calls. Only the name of private modules are enough in most cases. For example, Figure 4 is the `stm` file for two unikernels that shows unikernel 1 asking for a PIE-U block disk, a private file "keys.txt" as well as TW networking. Similarly, unikernel 2 asks for a PIE-K console UI and TW Storage. To provide a usable development environment all trusted APIs are hidden from the developer and aggregated by the Snape hypervisor.

```
#Trust Model#
//Unikernel #1:
//Private in-enclave FS & Block Device
//trusted_wrapped (tw) Networking
p_disk:(pie-u,"/tmp/unik1");
p_file:keys.txt
p_networking(tw);

//Unikernel #2:
//Private in-enclave UI
//trusted_wrapped (tw) FS
p_disk:(tw,"/tmp/unik2");
p_file:secret.txt
p_ui:(pie-k);
```

Figure 4. Snape Trust Model (stm) of two unikernels

4 Proof-of-concept implementation

We decided to implement Snape PoC using existing unikernel systems since these lightweight libOSes already significantly reduce applications dependencies to the complex untrusted host that makes them suitable for many edge-cloud based deployments [19, 25]. We also decided to focus on building the Snape for the ARM architecture not only because it is widely used in edge devices, but also since ARM security extension provides a rich set of hardware security options (§2.2). However, to show how our design can be portable to other platforms, we partially ported Snape to x86-64 architecture that utilizes Intel SGX as we briefly discuss in (§4.3). We implemented Snape hypervisor as a unikernel monitor and a backend for Solo5[4] that supports several unikernel monitors to run different unikernels such as Rumprun, MirageOS, IncludeOS, and Solo5. Hence, Snape's hypervisor follows the similar interfaces that make support of these unikernels straightforward with some minor modifications. Specifically, the high-level interfaces of Snape hypervisor are similar to Solo5 hvt [25]; that is a tiny Type-2 hypervisor ($\sim$ 1500LoC) that exposes a minimal interface to the host OS and KVM ($\sim$ 10 hypercalls) for unikernel isolation and accessing IO such as network and block devices. However, Snape hypervisor provides the same functionalities using TrustZone based security services. Snape PoC supports only essential security services that (as Table 2 shows) have a lightweight code base. Our prototype runs on a Raspberry Pi 3 which supports a TrustZone capable CPU. However, it currently lacks support for TZASC to isolate secure-world memory, so the secure memory is statically configured.

4.1 Snape Secure World

Snape's secure world OS is based on the core kernel of OPTEE-OS[5]. The secure monitor runs Arm Trusted Firmware and initializes secure world resources. Communication between the two worlds is possible through message passing via SMC calls. The secure kernel follows ARM SMC calling conventions to implement a simple RPC protocol. The ELF loader calls an RPC to load each enclave, that is a signed static ELF binary stored encrypted on the host Linux filesystem. Then depends on enclave type, it runs each enclave as a secure kernel module in S-EL1 or a separate process in S-EL0. The secure kernel MMU configures TTBRC registers that point to several L1 32 MB translation tables that cover virtual memory mapping of each S-EL0 enclave. Each enclave is internally identified by its UUID and signature (hash of the binary) and the secure kernel ensures the same enclave is not already loaded. We have implemented the followings Snape secure services on top of the secure OS with a focus on minimalism.

PIE – Filesystem This is a simplified baremetal filesystem implemented inside an enclave with no dependency to the untrusted world. Private disks are implemented as RAM disks using 512-byte or 4096-byte sectors. Disk layout begins with a filesystem header that describes the offsets of the allocation table, root directory, and other details such as size and filesystem signature. The disk driver provides minimal disk interfaces (e.g. read, write, seek). The filesystem files are contiguous and use 2MB disk blocks by default that can be optimized depends on uses cases.

PIE – User Interface Snape implements minimal user interface including console operations, GPIO access, and a

[4]https://github.com/Solo5/solo5
[5]https://github.com/OP-TEE/optee_os

Modules	LoC
Secure Kernel	20k
Snape hypervisor	3k
PIE FS	3.6K
PIE Crypto	7.4k
TW FS + TW Net	1.9K

Table 2. Snape major modules' LoC

mailbox interface to the rPi GPU based on rPi firmware[6][7]. To implement these in S-EL0, we use RPCs to the Snape driver in S-EL1 to read/write hardware registers as well as direct access to secure world physical memory.

TW– Filesystem & Networking These services act as a proxy between Snape VMs and the untrusted host to protect against Iago attacks. Snape drivers, within the secure kernel and Linux kernels, communicate through a secure RPC path and provide trusted wrappers around the host FS and networking stack. To reduce performance overhead, parameter marshalling is done through temporary shared memory between secure and normal worlds. For each system call, Snape drivers allocate temporary shared memory objects via RPC threads. After parameter validation, parameters are copied to the shared memory. The results from the host are checked, and after the validation, the output will be copied to registered secure memory. Snape drivers clean up the temporarily shared memory.

PIE– Crypto The Snape "tcrypt" enclave provides access to a hardware-based Random Number Generator (RNG) engine in the secure kernel through secure RPCs. It supports a variety of cryptography operations including modern cryptography protocols such as ChaCha20, Poly1305, SipHash, and BLAKE2.

4.2 Snape startup

Applications with no private section can run on Snape with no extra work and a default trust model. However, when the developer wants to use secure services, they configure the trusted model (an `stm` file) that will be loaded by Snape via a hypercall. Then based on the registered model, the Snape hypervisor initialises secure sessions to the secure world to ask for new enclaves and allocating private objects like disks and files. Upon successful initialisation, the secure kernel returns valid enclave context for invoking enclave functions. Snape redirects the relevant calls to the requested enclaves. For ease-of-programming, this aggregation phase and enclave specific calls are hidden from the developer via libraries. The hypervisor also saves the metadata of registered enclave so applications also can have access to it via hypercalls.

4.3 Snape portability to other platforms

The modular and declarative nature of Snape makes it relatively easy to support other platforms. Most of Snape security services and Snape hypervisor are standalone components written in C that are easy to port to other types of enclaves with minor modifications in external interfaces. For example, in our SGX-Snape, we ported both PIE and TW FS from our TrustZone PoC with total modification of 20 interfaces to/from SGX enclaves based on SGX SDK requirements. However, since SGX only provides protected memory in userspace, it is not suitable for providing all features in TrustZone-Snape .

5 Evaluation

To support real-world POSIX compatible applications, Snape needs some modifications over unikernels like Rumprun[8] for Aarch64-rPi3 support. We are therefore using microbenchmarks to evaluate the overhead of the major Snape security services: the PIE filesystem, PIE user interface, TW filesystem, and TW networking (see Table 3). We developed microbenchmarks for Solo5 unikernels and evaluated on Raspberry Pi 3, except for the PIE UI module that is evaluated on QEMU. The table shows the worst-case overhead of each module on solo5 unikernels with snape backend compares to the corresponding unikernels with Linux kernel backend. The PoC is not particularly optimized. Hence, the overhead of individual modules may be discouraging, however, it does not necessarily reflect the actual performance overhead when each module can be an only a small part of a real-world application that now has a flexible way to use all available optimizations from the untrusted host as well. We are working on proper evaluation and performance improvement of our system on real-world applications.

trusted services	slowdown
TW FS & Networking	2.7x-4x
PIE-K FS	<9x
PIE-U FS	<60x
PIE-K UI (on qemu)	<9x
PIE-U UI (on qemu)	<40x

Table 3. Snape average latency overhead

6 Related Work

Many systems utilise hypervisors for constructing private execution environments for applications [9, 12, 20, 21, 24]. Systems like Terra [9] and Proxos [24] introduce the concept of isolating private parts of applications in a dedicated VM that cannot be suitable for resource constrained edge devices. Snape and Proxos share a similar model for users trust management, though Snape does not require developers to specify all private syscalls. Systems like Haven [6] and Trustshadow [10] run unmodified applications inside SGX and Trustzone enclaves. As we discussed earlier (§1), running unmodified applications in an enclave on top of a large monolithic software stack has fundamental issues including exposing a single point of breach. Snape resolves these issues while providing a fine-grained flexible trust model.
Finally, we share the motivation of compartmentalized and

[6]https://github.com/raspberrypi/userland
[7]https://github.com/raspberrypi/firmware

[8]https://github.com/rumpkernel/rumprun

isolated OS components with microkernels [15]. This popular approach is used for handling heterogeneous architectures, for example, in HeliOS [22] satellite kernels. Microkernels like SeL4 [14] are small enough for formal verification, and widely used in embedded and mobile devices. However, formal verification is expensive and it is not always practical to formally verify all systems sensitive modules. Snape shares a similar isolated architecture while leveraging enclave-assisted isolation instead.

7 Conclusion & Future Work

We propose Snape, a declarative platform for applications to easily take advantage of various hardware security features on the heterogeneous edge. Our system takes the first steps toward an alternative approach from shielding applications inside a large complex monolithic software stack that limits applications from utilizing all hardware resources both from the protected and untrusted worlds. Our new design provides a flexible trust model for developers to manage their trust in the underlying system. We describe our prototype on ARMv8-based platforms to effectively use ARM TrustZone technology. Our approach can be extended to different hardware platforms in the future.

One immediate future work is to improve the efficiency and performance of our system and build more realistic applications. The Snape prototype can also benefit from optimizations in different areas such as optimized PIE services, reducing context switches, faster RPC communications, and automatic multicore parallelism.

References

[1] [n. d.]. CVE Details. https://www.cvedetails.com/cve/CVE-2016-2431/. Publish Date : 2016-05-09.

[2] [n. d.]. CVE Details. https://www.cvedetails.com/cve/CVE-2016-8763/. Publish Date : 2017-04-02.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[4] ARM. 2009. Security technology building a secure system using trustzone technology (white paper). *ARM Limited* (2009).

[5] ARM Security Technology. 2009. Building a Secure System using TrustZoneÂő Technology.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[7] Simon Biggs, Damon Lee, and Gernot Heiser. 2018. The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security. In *Proc. 9th Asia-Pacific Workshop on Systems (APSys '18)*. ACM, Article 16, 7 pages. https://doi.org/10.1145/3265723.3265733

[8] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, 253–264. https://doi.org/10.1145/2451116.2451145

[9] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. 2003. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 193–206.

[10] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. 2017. TrustShadow: Secure execution of unmodified applications with ARM TrustZone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 488–501.

[11] Hermann Hartig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. 2005. The Nizza secure-system architecture. In *Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on*. IEEE, 10–pp.

[12] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. 2013. Inktag: Secure applications on an untrusted operating system. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 265–278.

[13] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *ACM SOSP*. ACM, 207–220.

[15] J. Liedtke. 1995. On Micro-kernel Construction. In *Proc. ACM SOSP*. 237–250. https://doi.org/10.1145/224056.224075

[16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices.. In *USENIX Security Symposium*. 549–564.

[17] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[18] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proc. ACM ASPLOS*. 461–472. https://doi.org/10.1145/2451116.2451167

[19] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proc. ACM SOSP*. ACM, 218–233.

[20] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 143–158.

[21] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. In *ACM SIGOPS Operating Systems Review*, Vol. 42. ACM, 315–328.

[22] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 221–234.

[23] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *ACM SIGARCH Computer Architecture News*, Vol. 42. ACM, 67–80.

[24] Richard Ta-Min, Lionel Litty, and David Lie. 2006. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proc. USENIX OSDI*. USENIX Association, 279–292.

[25] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box.. In *HotCloud*.

[26] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 640–656.