Towards Modular Trusted Execution Environments

Carsten Weinhold
Nils Asmussen
carsten.weinhold@barkhauseninstitut.org
nils.asmussen@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Diana Göhringer
diana.goehringer@tu-dresden.de
Technische Universität Dresden
Dresden, Germany

Michael Roitzsch
michael.roitzsch@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Abstract

State-of-the-art implementations of Trusted Execution Environments (TEEs) present system designers and users with several problems: First, it is not possible to choose a TEE implementation independently from the instruction set architecture. Second, the security-critical functionality of such TEEs is deeply integrated into the micro-architecture of complex processor cores, making programs running in such TEEs vulnerable to side-channel attacks. And third, the interface and execution model of certain types of TEEs make it hard to integrate these TEEs with the system software. To address these issues, we propose a modular TEE design. We apply this modular design to the M³ hardware/software co-design platform and demonstrate how TEE support can be made a first-class feature at the system-architecture level.

CCS Concepts: • **Security and privacy → Hardware-based security protocols**; • **Software and its engineering → Operating systems**.

Keywords: tile-based architecture, trusted execution

ACM Reference Format:
Carsten Weinhold, Nils Asmussen, Diana Göhringer, and Michael Roitzsch. 2023. Towards Modular Trusted Execution Environments. In *6th Workshop on System Software for Trusted Execution (SysTEX '23), May 8, 2023, Rome, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3578359.3593037

1 Introduction

Today, the interface, functionality, and implementation of a Trusted Execution Environment (TEE) are deeply intertwined with both the instruction set and the hardware architecture of a processor. By deciding on an instruction-set architecture (ISA), one also chooses a specific type of TEE, and vice versa. This inter-dependency limits options for system designers. With Intel SGX, there is a TEE variant (called an *enclave*) that supports running application code without any additional system-level code inside the TEE. However, SGX is only available for x86-64 and not, for example, for Arm. And even on x86-64, SGX support is mostly limited to high-performance and high-power CPUs. TEE flavors isolating entire virtual machines (VMs) are also limited to datacenter-class CPUs. Such VM-type TEEs are available from Intel (Trusted Domain Extensions, TDX), AMD (Secure Encrypted Virtualization, SEV), or Arm (Confidential Compute Architecture, CCA).

However, not only are the interface and execution model of a TEE tied to a specific ISA, but their implementation also depends on the processor architecture. SGX, TDX, SEV, and CCA are each deeply integrated into the micro-architecture of a highly-complex processor core. Such cores have been shown to be vulnerable to side-channel attacks [8], which can also affect code running inside a TEE [13]. In the case of SGX, research even demonstrated full compromise of the secret signature keys inside the SGX quoting enclave [12]. AMD processors with support for SEV improve TEE security by isolating the key material and cryptographic operations required for attestation on a dedicated Platform Security Processor (PSP) [2]. The PSP is separate from the x86-64 cores used by the operating system (OS), applications, and VM-based TEEs. Therefore, it is harder to attack the PSP using side channel attacks like those that can compromise SGX's quoting enclave. In our view, a step in the right direction.

Finally, the TEE implementations that are available today lack system-level integration. They have been designed under the assumption that the OS (for enclave-type TEEs) or the hypervisor (for VM-type TEEs) cannot be trusted. The reasoning behind this assumption is that the system software is a huge, highly complex, and often monolithic code base that represents an unacceptably large attack surface. While the OS running in a VM-type TEE may be self-sufficient, this is not the case for user programs running in an enclave-type TEE. In practice, most of them depend on services provided by the host OS. This need is addressed by security wrappers such as Scone [3], which either enable secure reuse of certain OS functionality or duplicate it inside the TEE.

In light of these observations, we argue that the interface and feature set of TEEs, the enforcement of their security guarantees, and the hardware that enables program execution should be decoupled from each other. This separation of

SysTEX '23, May 8, 2023, Rome, Italy

ACM ISBN 979-8-4007-0087-3/23/05.
https://doi.org/10.1145/3578359.3593037

interfaces, functionality, and ISA enables both customization of TEE implementations and more control over architectural decision that determine their security properties.

The contributions we make in this paper are the following: (1) We propose to modularize TEEs into six separate concerns (Section 2). (2) We map these six concerns onto the M³ [5] hardware/software co-design platform. We discuss which hardware and software building blocks of M³ can be reused and propose extensions that enable TEEs as a first-class platform feature (Section 3). (3) We discuss our modular TEE design in the context of related work and explore challenges for further extending TEE integration (Sections 4 and 5).

2 Motivation and Background

We start by summarizing our understanding of the term TEE. We discuss the six concerns of a TEE implementation and show, how typical representatives lump them together in one monolithic implementation. We then identify microkernels and tiled architectures as a promising way forward.

2.1 Trusted Execution

In our view, a TEE is an execution container for code that an external party wants to associate with security assertions in order to establish trust. The external party therefore requests cryptographic evidence from the TEE platform that demonstrates the integrity of the TEE's state, for example that the intended program (and not a modified version) is being executed on the expected hardware platform (and not on an emulated processor). Remote attestation is therefore a building block of a TEE.

A TEE however extends pure remote attestation with a temporal assertion: The evidence collected from the platform contains proof of mechanisms that ensure, that the integrity of the TEE's state will endure. The platform must provide sufficiently strong isolation features, that the TEE's integrity not only holds at the time of evidence collection, but that this integrity is ensured in the future. If, after inspecting the evidence, the external party trusts this long-lived integrity, then we have reason to call the execution container a TEE.

2.2 Separation of Concerns

We identify six concerns every TEE platform must provide:

- **Computation:** The TEE is an execution container for code, so this code needs to run on a processor.
- **Measurement:** The cryptographic evidence supplied to the external party is collected by the platform in a measurement process, which includes a hash of the loaded executable.
- **Root of trust:** The measurement alone is worthless, because the execution container could lie about it. Thus, the measurement is signed using key material that is embedded in hardware in a root of trust (RoT).
- **Isolation:** To maintain integrity over time, the platform must protect TEEs from manipulation. Protection is typically implemented by memory isolation (e.g., virtual memory) and memory encryption to prevent physical attacks.
- **Management:** These isolation and encryption mechanisms need to be managed in order to multiplex the platform to run multiple TEEs.
- **Environment:** A TEE running in perfect isolation is rather useless. At least an output channel is required to communicate results. More broadly, TEEs need controlled access to system services.

We observe that current TEE implementations are often monolithic, lumping many of these concerns together. SGX for example ties together an Intel processor for computation with specific measurement code accessing a dedicated RoT. Isolation mechanisms are managed by the opaque SGX firmware. Environment access from an SGX enclave has been notoriously difficult and spurred research work such as Scone [3].

2.3 Deconstructing Trusted Execution

We argue that monolithic TEE implementations can be deconstructed into orthogonal building blocks. Such a modular TEE would have two advantages: First, as the individual building blocks become smaller and have clear interfaces, it becomes easier for external parties to inspect them and be convinced of their correctness. Second, the platform can offer more flexibility as different implementations of the TEE concerns can be combined to configure different TEE flavors.

For example, one external party might desire an SGX-enclave style TEE that laterally interacts with outside services, whereas others prefer a TDX-style virtual machine bringing its own operating system. Regarding isolation, some parties may require memory encryption with costly freshness protection, while others choose to disregard physical attacks. Measurement may need to cover only the TEE address space for some use cases, while other users want to integrate an external device into the TEE, thus requiring more complex measurement logic. As to the RoT, different parties may request different cryptographic algorithms like traditional or quantum-safe options. And ultimately, for computation of their TEE, some users may want to restrict their most critical use cases to hardened in-order processors.

All these choices are currently only possible by completely switching platforms, because each specific platform has one instance of each concern baked in. A modular TEE would be able to offer such choices within a single platform.

2.4 Microkernels as a Starting Point

Current TEE implementations are not far away from microkernel systems. Intel TDX for example manages its TEEs using a hidden software-implemented microkernel running

in a new processor mode [7]. This hidden microkernel programs page tables and memory encryption to isolate the TEEs and it interfaces with a hardware-embedded RoT to sign an attestation measurement. The microkernel becomes part of the attested software stack such that an external party can be convinced of the correct usage of isolation features.

Including a microkernel in the trusted platform may seem counter-intuitive, because a selling point of TEEs has been to remove trust in the operating system. But this was motivated by large and complex monolithic operating systems. A microkernel is orders of magnitude smaller and can be formally verified. Many existing TEE implementations like SGX or TDX include a hidden microkernel in firmware that is implicitly trusted because it is provided by the hardware vendor.

Using an actual, explicit microkernel however improves transparency as the code can be inspected. Exposing the kernel enables TEEs to use APIs for cross-TEE communication and security mechanisms. Furthermore, it allows to separate hardware and software building blocks to enable the customization choices we desire. For example, it enables co-existence of process-style and VM-style TEEs on the same system as today's microkernels offer both abstractions. Process-style TEEs no longer require a bespoke processor enclave mode and can therefore naturally access system services via the microkernel's native communication primitives.

However, a microkernel-based system cannot easily offer a choice of different processors to execute the TEEs. A traditional microkernel runs in kernel mode on the same processor as the TEEs it protects. This construction is vulnerable to side-channel attacks, because all TEEs as well as critical components with access to the RoT share processor hardware.

In addition to a composable software layer, our modular TEE platform also needs composable hardware. Therefore, we build upon the M³ architecture, which we briefly explain before presenting our modular TEE design in Section 3.

2.5 The M³ Architecture

M³ [5] proposes a new system architecture based on a hardware/software co-design. On the hardware side, M³ builds upon a tiled architecture, as shown in Figure 1. M³ extends its tiles by adding a new hardware component called data transfer unit (DTU) to them. Each tile contains a DTU and either a core, an accelerator, or memory (e.g., a memory interface to off-chip DRAM) and the tiles are connected via a network-on-chip. As the DTU is the only way to access tile-external resources, the DTU controls the tile's access permissions. By default, all tiles are isolated from each other. To perform message-passing between tiles or access memory, a corresponding *communication channel* (thick black lines in the figure) needs to be established. These communication channels are represented as *endpoints* in the DTU (orange dots).

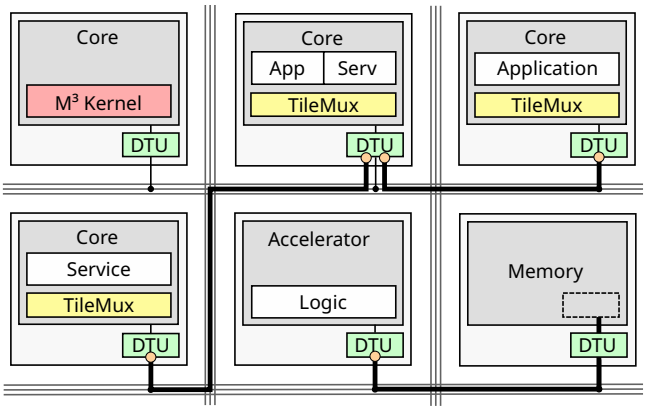

Figure 1. System architecture of M³: one DTU per tile isolates tiles from each other and selectively allows communication. TileMux multiplexes its tile among the applications on it.

On the software side, M³ runs a microkernel (red) on a dedicated *kernel tile*, and applications and OS services on the remaining *user tiles*. Applications and OS services on user tiles are represented as *activities*, comparable to processes. An activity on a general-purpose tile executes code, whereas an activity on an accelerator tile uses the accelerator's logic. Activities can use existing communication channels, but only the M³ kernel is allowed to establish such channels. Applications are placed on different tiles by default, but as shown by M³v [4], tiles with general-purpose cores can also be shared efficiently and securely among multiple applications. For that reason, every core-based user tile runs a multiplexer called *TileMux* (yellow), which is responsible for isolating and scheduling the applications on its own tile, similar to a traditional microkernel. However, in contrast to a kernel, each TileMux instance has no permissions beyond its own tile. Instead, only the M³ microkernel can make system-wide decisions, hence its name.

3 Modular TEEs as a Platform Feature

Because of its disaggregated hardware architecture and its microkernel-based OS, the M³ platform is an ideal starting point for a modular TEE design. Before we discuss this design, we define the threat model and security assumptions.

3.1 Threat Model

Our threat model considers the hardware and software building blocks of M³ that are critical to providing the six TEE concerns defined in the preceding section.

We assume that the DTUs and the network-on-chip through which they are connected cannot be manipulated or accessed directly at the hardware level. We also assume that secret key material belonging to the RoT is embedded into the hardware and therefore inaccessible to an attacker. These

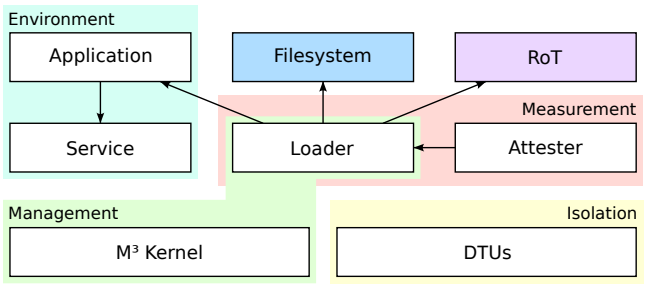

Figure 2. Mapping of TEE concerns onto hardware and software building blocks of M³.

assumptions are comparable to those for contemporary TEE-enabled CPUs. We further assume the kernel tile, its processor core, and the memory used by the M³ kernel to be trusted. The same applies to user tiles that host software-based functionality that is critical to implementing any of the six TEE concerns. The system services that provide such functionality will be detailed in Subsections 3.2.5 and 3.2.3. All other user tile processor cores and the software running on them need not be trusted.

From an application's point of view, additional system services or applications may be considered security-critical. We discuss application-specific trust in Subsection 3.2.4.

3.2 Mapping of TEE Concerns onto M³

Figure 2 visualizes how the six TEE concerns map onto hardware and software components of M³. Below, we explain the roles of these building blocks and what extensions we made to M³ to implement TEEs as a first-class feature.

3.2.1 Compute and Isolation. An activity executing on a user tile is similar to what is called a process in other OSes. The processor core on a user tile enables program execution. Between activities, M³ offers two levels of isolation: Inter-tile isolation is enforced by the DTU, which enforces communication and memory-access permissions without relying on cores to self-restrain by means of page tables. If multiple activities share the processing core of a user tile, isolation among them is enforced by TileMux. This intra-tile isolation is based on page tables and support for multiple privilege levels as provided by the processor core on that tile.

We can reuse the activity abstraction to provide the Compute and Isolation concerns for a modular TEE. If a user tile is assigned exclusively to only one TEE, the program running in that TEE is less susceptible to side-channel weaknesses in the core's micro-architecture. Since no untrusted code shares the core, side-channel attacks can only originate from outside the tile [11]. Alternatively, by colocating a TEE with other activities, tile utilization can be increased at the cost of weaker isolation.

3.2.2 Root of Trust. Inter-tile isolation is ideal to protect the platform's RoT in our modular TEE design. We propose to add a new, dedicated *RoT tile* that stores an identity key in hardware. The key itself is not accessible from any user tile or the kernel tile. However, a software-implemented *RoT service*, which runs on a core that is also part of the RoT tile, can make use of the RoT's key to perform signature operations. A TEE signature is created in two steps: First, the measurement components (see Subsection 3.2.5) report to the RoT what software is running in a TEE. Second, the RoT creates a signature over the TEE measurement using its identity key, thus producing cryptographic evidence of the previously reported software configuration in that TEE. Such attestation evidence is sent as the reply to a remote attestation request.

To minimize the attack surface, the RoT service's complexity should be reduced. Since it is the only software running on the tile, there is no need to run TileMux on the RoT tile's core. In fact, it is possible to exploit M³'s support for heterogeneous cores to reduce hardware complexity as well: System designers can configure the RoT tile with an in-order processor core without support for virtual memory and privilege separation to reduce the risk of hardware bugs and timing side-channels.

3.2.3 Management. In M³, a *loader* service is responsible for making the code and data sections of the program binary available in an activity's address space. To accomplish this task, the loader asks the M³ kernel to configure DTU endpoints that allow the program to access appropriate memory regions in off-tile DRAM. Additionally, the loader requests the kernel to configure DTU endpoints for message-passing channels between the TEE and other activities; for example, to allow an application to communicate with system services.

The M³ loader already has all information about the startup state of an activity, including input and output channels to other activities on the system. This information is precisely what an external party can use to assess the trustworthiness of the program running in this activity.

3.2.4 Environment. No special privilege level (like SGX' enclave mode) or adapter (like Scone [3]) are needed to let a user program in an M³-based TEE interact with the rest of the system. Also, software running in such a TEE does not need to bring its own OS, as is required for the VM-type TEEs like those provided by Intel TDX [7] and other vendors' solutions. However, there are no architectural limitations that prevent it either; for example, a complete Linux OS including user programs could run on a user tile. In this case, TileMux would be replaced by the Linux kernel, which could enable Linux user programs to interact with M³ activities on other tiles by providing a device driver that abstracts the DTU for cross-tile communication. Thus, a modular TEE design based on M³ can support both native user programs (running as activities) and VM-type TEEs. In both cases, access control for off-tile resources would be managed in the

same way. Any software running in an M³-based TEE can use all APIs and access any service on any tile, as long as it has permission to do so (i.e., DTU endpoints have been configured accordingly).

However, programs that depend on certain system services or other applications, may have to trust in the integrity and correct behavior of other activities beside their own TEE. On M³, such trusted system services can easily be considered as a TEE themselves, since there is no downside to doing so regarding cross-activity interaction. In practice, we therefore consider groups of cooperating TEEs rather than single TEEs.

3.2.5 Measurement. To report the state of one or more TEEs to an external party, a cryptographic protocol called remote attestation is used. This protocol is executed between two computer systems: an *attester* that measures and reports evidence of the state of a set of TEEs, and a *verifier*, which is a remote entity that wants to learn about the attester's TEE state. In this subsection, we explain the role of the attester in a modular TEE implementation on M³. To do so, we describe how measurement reporting works, when a remote-attestation request is received by an application over the network.

In such a scenario, the application program, running as an activity (i.e., TEE) on the attester system, forwards the attestation request through an API to the loader service that started the application. For the reasons outlined in the preceding subsection, the configuration of memory and message-passing endpoints among groups of cooperating TEEs must be part of TEE measurement. Hence, the loader must able to provide a reply with an evidence report that includes the state of multiple TEEs. We propose to extend the already existing loader service of M³ with the aforementioned API to forward attestation requests. Furthermore, we introduce a new *attester* service, to which the loader outsources the measurement of all relevant program binaries that it needs to start as TEEs. The attester service computes cryptographic hashes of the programs before they are mapped into their respective TEE address spaces. In our current design, the attester service does so by acting as an interposition layer between the M³ file-system service and the loader. The following steps take place to perform a measured start of an application program in a modular, M³-based TEE:

1. The loader opens the application's binary.
2. The attester service intercepts this open request, creates a copy of the complete binary file in private memory, and computes the hash over this copy of the file.
3. The attester service makes the in-memory copy of the file available to the loader in response to the open operation. It retains the measurement hash for future requests.

By creating a copy of the program binary, the attester service prevents time-of-check-time-of-use (TOCTOU) attacks on the original file in the file system, where the binary is modified after measurement. Therefore, the file system need not be trusted for integrity. The above steps are executed, when TEEs are started. The following steps are performed, when the attestation request arrives over the network.

4. The loader passes the hashes of the binary programs running in the TEE group and the configuration of their message-passing and memory endpoints to the RoT.
5. The RoT signs the set of hashes and the associated configuration with its identity key, thereby creating cryptographic evidence that describes the group of cooperating TEEs and the programs executing in them.
6. The loader receives the signed evidence and passes it back to the application, which then sends it to the verifier.

Finally, the verifier (i.e., the external party) can now validate the signature over the received evidence using the public part of the RoT's signature key. If the signature is found to be valid, and the TEE configuration described in the evidence report is deemed acceptable, it can place trust in the integrity and correct behavior of the TEEs in the attester device.

3.2.6 Open Issues. Our design, as described so far, does not address the problem of securely starting the kernel, the RoT service, and other services such as the loader. This includes the issue of how to include cryptographic hashes over the kernel and basic services in the evidence report. We also did not discuss the need for software updates, how to derive RoT identity keys from hardware secrets so as to enable revocation and re-provisioning, and the possible involvement of certificate authorities that vouch for security properties of an M³-based hardware platform. We believe these issues can be solved using existing technologies and we plan to address them in future work.

4 Comparison with Related Work

To show the unique benefits of M³-based modular TEEs, we qualitatively compare the discussed design with existing commercial and research solutions regarding their modularity.

All-in-one Solutions. Products like Intel SGX [1] are monolithic solutions, disallowing separate consideration of the individual concerns. Although some aspects of SGX are implemented in firmware, the interaction with the hardware remains opaque. Sancus [9] is less complex and targeted at microcontroller cores, but it explicitly addresses all TEE concerns in hardware. Such all-in-one solutions reduce trust to a single artifact: the processor core. While this can simplify trust decisions, it is deliberately non-modular.

Microkernels. A number of current solutions are based on a microkernel or micro-hypervisor to manage TEEs, program the hardware isolation features, and interact with measurement and RoT. Intel TDX [7] runs a signed micro-hypervisor

provided by Intel. Arm CCA equally uses software to switch between their flavor of TEEs called Realms. Sanctum [6] is a research extension to the RISC-V ISA, which runs a security monitor in the RISC-V machine mode.

The microkernel approach is conceptually much closer to our modular design. It can accommodate different measurement approaches or different RoT implementations. Whether vendors offer such configurability is merely a business decision, not an architectural one. However, these microkernels run in bespoke hardware modes on the same processor. This design has two consequences: First, being a mode on the same processor cores, the microkernel is susceptible to side channel attacks. This can have devastating consequences, when key material from the RoT can be accessed. Second, this bespoke CPU mode has access to architectural features, which the processor vendor intends to be used for inter-TEE isolation. These isolation features may dictate a certain TEE flavor (process-style of VM-style) and may complicate TEE cooperation.

The Sanctum authors identified these two limitations and addressed them by hardening the core against side-channel attacks using cache partitioning and flushing, as well as by offering synchronous communication primitives for TEE interaction in their security monitor.

Tiled Architectures. AMD SEV externalizes TEE management, measurement, and RoT interaction to a separate processor, the Platform Security Processor (PSP) [2]. This way, TEE computation on the main processor is separated from TEE management on the PSP. The main processor no longer has access to any key material, mitigating a number of potential attacks. In a way, M³ extends this idea further. AMDs model still relies on parts of the main processor for inter-TEE isolation. The M³ DTU offers a way to separate inter-TEE isolation and communication concerns from the processor cores entirely. TEEs become possible with any unmodified core, offering new platform configuration options.

Two caveats apply: Context-switching among multiple TEEs on the same M³ tile still relies on core-internal isolation features. However, M³ could include a Sanctum-style hardened core for this purpose. Multithreaded TEEs currently require a multicore processor within one tile. Cache coherence between tiles is future work. We still believe the M³ design constitutes the most modular point in the solution space.

5 Challenges

The main security benefit of a tiled architecture like M³ stems from the physical separation of the compute units on the chip. An interesting question for the security of TEEs therefore is, how this isolation could be compromised. Traditional processors are often attacked through cache-based side-channels, because the cache is a high-throughput shared resource. In M³, two such shared resources remain: shared DRAM and the network-on-chip (NoC). For DRAM, existing solutions like memory encryption and Rowhammer protection are available. The shared NoC however constitutes an interesting target for timing side-channels and thus requires further study.

The NoC topology, switching mechanism, and routing algorithm each have a strong influence on non-functional parameters, such as observed message latency or aggregate throughput. The recent trend of in-network computing would exacerbate these problems. In-NoC computing allows to see the NoC as a large array of routers with computing capabilities [10]. Similar as for in- and near-memory computing, the idea here is to process the data as close to its current location as possible and to use all available computing components for it. This allows leveraging the communication latencies within the router for executing simple computations on the data. Examples are reduction instructions on data elements of a network packet, e.g. by summing up all data elements.

In-NoC computing brings additional challenges to a tile-based TEE implementation, because the NoC constitutes a high-throughput shared resource and needs to be trusted.

6 Conclusion

To overcome the inflexibilities of existing TEE implementations, we have presented a modular design approach. We extended the M³ hardware/software co-design platform with a TEE infrastructure, where building blocks like measurement, isolation, management, and environment interaction can be freely configured and exchanged. The modular design uses a microkernel and existing cores rather than complex core modifications to enable trusted execution.

Acknowledgments

This research is funded by the European Union's Horizon 2020 research and innovation program under grant agreement No. 957216 (iNGENIOUS), as well as grant agreement No. 101092598 (COREnext) under the Horizon Europe program. It is also financed on the basis of the budget passed by the Saxon State Parliament in Germany.

References

[1] 2021. Intel Software Guard Extensions (Intel SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. (Accessed on April 12, 2023).

[2] Advanced Micro Devices, Inc. 2016. *Platform Security Processor (PSP).* https://www.amd.com/system/files/TechDocs/52740_16h_Models_30h-3Fh_BKDG.pdf, 156–157.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. Scone: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems*

Design and Implementation (OSDI) (Savannah, GA, USA). USENIX, 689–703.

[4] Nils Asmussen, Sebastian Haas, Carsten Weinhold, Till Miemietz, and Michael Roitzsch. 2022. Efficient and Scalable Core Multiplexing with M³v. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Lausanne, Switzerland). ACM, 452–466. https://doi.org/10.1145/3503222.3507741

[5] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Atlanta, GA, USA). ACM, 189–203. https://doi.org/10.1145/2872362.2872371

[6] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. 857–874.

[7] Intel. 2021. *Intel Trust Domain Extensions*. Technical Report.

[8] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *meltdownattack.com* (2018). https://spectreattack.com/spectre.pdf

[9] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. 2017. Sancus 2.0: A low-Cost Security Architecture for IoT Devices. *ACM Transactions on Privacy and Security (TOPS)* 20, 3 (2017), 1–33.

[10] Jens Rettkowski and Diana Göhringer. 2021. Wormhole Computing in Networks-on-Chip. In *31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 273–274.

[11] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11735)*, Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan (Eds.). Springer, 279–299. https://doi.org/10.1007/978-3-030-29959-0_14

[12] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.

[13] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy (SP)*. 88–105. https://doi.org/10.1109/SP.2019.00087