HARES: A Framework for Transparent and Cross-Architecture Enclave Offloading

Abilesh Sundarasamy
Virginia Tech
Blacksburg, USA

Xiaoguang Wang
University of Illinois Chicago
Chicago, USA

Carlos Bilbao
Virginia Tech
Blacksburg, USA

Binoy Ravindran
Virginia Tech
Blacksburg, USA

Abstract

Hardware vendors continue to introduce architecture-specific security extensions to enhance software protection. Yet, due to intellectual property concerns and specific architectural designs, these crucial features are often limited to select CPU families, creating a significant availability gap. This problem is particularly pronounced in embedded devices that lack native TEE support.

This paper tackles the limited availability of security-related CPU extensions, specifically by investigating how embedded devices lacking native TEE features can gain access to hardware trusted execution environments (TEEs). We present HARES, a novel framework that enables these resource-constrained devices to transparently offload security-sensitive workloads to remote TEEs hosted on centralized edge servers. By doing so, client devices can benefit from advanced hardware security features available on more capable machines. A key advantage of HARES is its compatibility with existing TEE SDKs, such as Open Enclave and Intel SGX, allowing unmodified applications to run securely on remote enclaves with no changes to their source code.

We evaluate HARES in terms of security and performance by deploying it in an edge computing environment and testing one microbenchmark and six popular open-source applications. Our results show that HARES offers meaningful runtime security enhancements while incurring acceptable overhead, making remote enclave offloading both practical and cost-effective for real-world use cases.

CCS Concepts

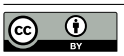
• **Security and privacy → Software and application security**; **Systems security**.

Keywords

Enclave Offloading, Heterogeneous Architecture, Remote TEE, Software Security

ACM Reference Format:

Abilesh Sundarasamy, Xiaoguang Wang, Carlos Bilbao, and Binoy Ravindran. 2026. HARES: A Framework for Transparent and Cross-Architecture Enclave Offloading. In *27th International Middleware Conference (Middleware '26), December 14–18, 2026, Tarragona, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3801927.3810471



Middleware '26, Tarragona, Spain

ACM ISBN 979-8-4007-2621-7/2026/12
https://doi.org/10.1145/3801927.3810471

1 Introduction

The hybrid-CPU architecture environment has gained more attention due to the growing popularity of mobile computing, the internet of things (IoT), machine learning, and data centers. Many new computing patterns have been developed. For example, an edge server can be a hub for smart home devices to process a large amount of data [53]. An ARM-based smart NIC can process distributed transactions [49] or run network server applications [51]. The workloads on these new computing environments are often split across the hybrid-CPU architecture boundary. This brings a new challenge for applying existing software security models to this new computation scenario.

Some barriers hinder securing a program on a hybrid-CPU architecture. In particular, many existing software protection techniques require hardware-specific security extensions [5, 20, 71]. These hardware extensions are either architecture specific (only available on a particular architecture) or CPU model (or vendor) specific. For example, the software guard extension (SGX) is only available on some particular Intel x86 CPUs, not for any AMD x86 CPUs [24]. There lacks an effective way to utilize them to protect software running on a hybrid-CPU architecture. Furthermore, code splitting and data movement are complex for programs running on a hybrid-CPU architecture. This is partly because many heterogeneous architecture nodes are cache- and memory-incoherent; moving sensitive data closer to a CPU core with particular security extensions requires manual code instrumentation and extensive data synchronization. Thus, very few existing works have considered software security enhancement for executing code across heterogeneous cores.

One way to solve this problem is to build a universal API for different CPU types and security extensions [13, 34, 41]. *Open Enclave* is such a project aiming to build a universal programming interface by abstracting different enclave types of various architectures [41]. Although the goal is ambitious, supporting multiple enclave types with a single API seems complicated as, *e.g., Open Enclave* developers recently confirmed that ARM TEE support is still under development [12]. Other research efforts, such as FlexOS [34], build an abstraction for each memory isolation extension so that users can switch between different protection primitives at deployment time. However, FlexOS's approach only works on a single machine node (one architecture) with different isolation primitives,

such as Intel MPK and EPT [34]. It cannot be easily applied to a hybrid cloud or an IoT/edge environment of varying CPU types.

This paper presents our exploration efforts on bridging the availability gap of security-related CPU extensions in a hybrid-CPU environment. We use the hardware enclave as an example and present the design and implementation of Hares (Hybrid-Architecture Extensions for Security), a framework to securely offload security-sensitive computations from TEE-lacking CPU cores to remote CPU cores with hardware enclave support, in a Security as a Service (SECaaS) approach. Hares mainly consists of two parts: an extended version of the Open Enclave SDK and a userspace Hares code monitor. Applications written using the Open Enclave SDK [41] can directly run on TEE-lacking CPU cores and transparently offload their confidential computations to a remote node with SGX support. Additionally, Hares supports unmodified enclave applications written in vanilla Intel SGX SDK [24] to execute on a non-SGX x86 node; Hares transparently runs the enclave part of the program on an SGX-enabled node.

The Hares monitor is implemented as a userspace program and runs in a separate address space from the target process. The Hares monitor transparently maintains a synchronized shared memory region between memory- and cache-incoherent computing nodes. A page update on one node will be transparently synchronized with the other node. The Hares monitor leverages a recent Linux kernel feature, `userfaultfd` [47], to delegate page faults to the userspace monitor; thus, Hares does not require any kernel modifications. We have supported several open-source SGX applications for machine learning, data sealing, and file encryption that run on a non-SGX machine node, with the enclave offloaded to an SGX-enabled node. We ran the experiments on an IoT/edge computing environment (a Raspberry Pi and an SGX-enabled laptop) and a cloud computing environment (two x86 machines from the cloud) using Hares. In summary, we make the following contributions:

- We explore the research space and design requirements of utilizing hardware security extensions on memory- and cache-incoherent, heterogeneous architectures to secure program code and data;
- We use the hardware enclave as an example to demonstrate the feasibility of such a vision and present Hares for securely offloading confidential computations to a remote enclave across different machine nodes;
- We extend a popular hardware-agnostic open-source Open Enclave SDK [41] and the widely used Intel SGX SDK [24] to ease the integration of Hares with security-sensitive applications across different machine nodes;
- We evaluate Hares with real-world machine learning and data encryption/sealing applications and demonstrate the reasonable performance overhead.

2 Background and Threat Model

Recent commodity CPUs offer numerous security hardware extensions, such as hardware-assisted pointer validation (*e.g.,* ARM PA [46], Intel CET [25]), random number generators and cryptographic acceleration (*e.g.,* Intel AES-NI [26], ARM TRNG [3]), and trusted computing (*e.g.,* Intel SGX [24], AMD SEV [1], ARM TrustZone [4]), to name a few [9]. One example is the *Trusted Execution Environment (TEE)*, a hardware-based sandbox designed to protect an application's security-sensitive code and data [61]. In the TEE model, application code and data are divided into a trusted part (enclave) and an untrusted part (host). Code that handles confidential data is placed within the enclave. At runtime, hardware TEEs like Intel SGX enforce memory and computation safety for the trusted component. If the host needs to access enclave data, it must obtain explicit authorization from the enclave, which exposes a set of functions accessible from the outside, known as `ecall`s. Similarly, the host includes functions that the enclave can call, known as `ocall`s. The `ecall` and `ocall` interfaces are defined in an enclave definition language (EDL) file. Both the Intel SGX SDK and the Open Enclave SDK provide tools (*e.g.,* `oeedger8r` in the Open Enclave SDK) to convert a `.edl` file into interface files, enabling communication between the enclave and host code.

Although the general idea of TEE has gained traction, *different hardware vendors have their own hardware implementation of enclaves* (*e.g.,* Intel SGX [24], ARM TrustZone [4], AMD SEV [1], RISC-V Keystone [31]). The varying API specifications across these implementations increase the challenge of developing trusted applications. This challenge has motivated projects like Open Enclave [41] and Asylo [13], which aim to make enclave deployment more flexible. For example, Open Enclave is an open-source and hardware-agnostic API interface (SDK) for enclave development [41]. It was initially designed to support Intel SGX and ARM TrustZone as enclave targets. However, due to differing hardware interfaces, ARM TrustZone is not fully supported at the time of writing [12, 45]. Besides the various hardware interfaces, *hardware enclaves are not ubiquitous.* There is a lack of enclave support for embedded devices. Moreover, Intel plans only to support SGX on server CPUs while deprecating SGX support on its 11th and 12th-gen desktop CPUs [52]. Thus, there is a need to support existing confidential computation applications and their use scenarios running on devices without hardware TEE support.

A motivating example: Edge devices, such as wearable health monitors, often handle sensitive data that must be protected during computation to comply with privacy regulations like GDPR [19] and HIPAA [22]. However, their limited computational power and lack of TEEs make secure processing challenging. In our target setting, the edge device acts as the client that initiates a security-sensitive computation. The client is trusted to initiate execution and verify the remote enclave, but lacks TEE hardware to protect the computation locally. The client therefore offloads the security-sensitive portion of execution to a remote SGX enclave hosted on a server node, whose OS and software stack may be compromised. Before provisioning any secrets, the client verifies the enclave identity via SGX remote attestation. For instance, privacy-preserving machine learning, as explored by Ohrimenko et al. [44], leverages TEEs to secure data during training. By extending such capabilities to hybrid architectures, Hares enables edge devices to benefit from hardware-backed protection even when TEE support is unavailable locally.

Checkpoint/Restore in Userspace (CRIU) [10] is an open-source project that enables checkpoint/restore functionality in Linux. It allows applications to be frozen at any point, with their state saved as an image on the disk. Later, the same application can be restored

and executed exactly as during the freeze. CRIU has a family of APIs to manipulate a running program, including the CRIU/*compel* utility. The *compel* API [11] allows users to transparently execute a small piece of code in the context of a foreign process, also known as parasite code. Once compiled with compel flags and packed, the parasite code can be executed in another task's context. However, it should be noted that the code runs in an environment without `glibc`, meaning it cannot call the usual stdio/stdlib functions.

CRIU also utilizes the `userfaultfd` for restoring pages during the process restoration. `Userfaultfd` is a Linux kernel facility that allows code to handle page faults in userspace [27]. It allows an application to register regions of memory with a file descriptor. When a page fault happens (*e.g.,* when some memory pages have not been loaded into a process's context), the application is notified of it and is able to serve the fault. Once a `userfaultfd` file descriptor is opened, it can also be passed to a manager process using UNIX domain sockets so that the same manager process can handle the page faults of a multitude of different processes [27].

Distributed Shared Memory (DSM) is a memory abstraction that allows physically distributed memory across multiple nodes to be accessed as if it were a single shared address space [35, 43]. DSM systems can be implemented in hardware or software. Hardware-based DSM solutions include architectures like cache-coherent Non-Uniform Memory Access (cc-NUMA) [64] and specialized interconnects via network interface controllers [63], which manage memory coherence across nodes at the hardware level. Software-based DSM, by contrast, operates at the OS or user level [6, 30, 42, 69]. One approach involves extending the OS kernel with DSM support, typically by intercepting page faults using a modified page fault handler [6, 30]. These systems enforce memory consistency, often using invalidation-based protocols such as MSI (Modified, Shared, Invalid) [62], to synchronize data across distributed nodes. For example, when a page is modified by one node, other replicas are either invalidated or updated, depending on the protocol, to preserve a coherent view of the global memory.

We argue that existing designs for distributed execution and shared memory are too heavyweight to be directly applicable to low-end CPU cores [6, 30], particularly in heterogeneous environments where they interoperate with high-end processors. In contrast, our design leverages lightweight and widely available Linux kernel interfaces, such as `userfaultfd` [27] and `ptrace`, to implement efficient cross-node shared memory, enabling transparent enclave offloading without requiring specialized hardware or kernel modifications. Alternatively, `process_vm_readv/writev` can be used for efficient bulk memory transfer when appropriate. However, it cannot replace `ptrace`, which is required to synchronize execution state and to perform fine-grained control-flow interception.

Threat Model and Assumptions. Hares assumes a threat model consistent with standard SGX-based confidential computing systems, while making the client/server roles explicit. The client device is trusted to initiate the remote enclave execution and verify the enclave identity, but it lacks TEE hardware to protect the computation locally. The client provisions secrets only after successful remote attestation. On the server side, only the SGX enclave and the SGX hardware root of trust are trusted; the server OS, hypervisor, user-space Hares monitor, and communication channel are untrusted. An adversary may control the software stack outside the enclave, observe or modify memory and network traffic, and interfere with the system's execution. Despite this, confidentiality and integrity of code and data within the enclave are preserved, assuming that the enclave is initialized with verified code and its identity is validated through SGX remote attestation. Sensitive data and secrets are provisioned only after successful remote attestation and are delivered through a secure channel that terminates inside the enclave, ensuring end-to-end protection regardless of server-side host compromise.

Hares does not aim to defend against certain classes of attacks that are outside the SGX security model, such as *denial-of-service*, *side-channel attacks*, or *interface-level exploits* like Iago attacks [7, 17, 29, 33]. For example, an adversary may delay or block communication, exploit timing differences to extract information, or craft malicious inputs to trigger undefined behavior at the enclave boundary. These threats are orthogonal and may be addressed by complementary mitigation techniques.

Moreover, Hares targets environments with reasonably stable client-server connectivity (*e.g.,* LAN/edge Ethernet or Wi-Fi). Its latency overhead is primarily influenced by network round trips; consequently, performance degrades as latency increases or connectivity becomes unstable. Supporting highly intermittent network conditions is beyond the scope of this work. Our current implementation further assumes little-endian architectures, which is satisfied by the `ARM64` and `x86_64` platforms evaluated in this paper.

3 System Design

Hares aims to transparently offload confidential computations to a computing node with SGX support in a lightweight way. To achieve this, Hares is designed to seamlessly divide code execution and data across different nodes and subsequently synchronize the computation results. It provides support for TEE-based applications (or containers) developed using various SDKs, such as Open Enclave [41], Intel SGX [24]. Figure 1 illustrates an overview of Hares operating on cache-incoherent heterogeneous computing nodes. Hares consists primarily of two components: a Hares program monitor for overseeing and synchronizing enclave code execution, and an extended enclave SDK for distributing `enclave` and `host` code across different machine nodes. In this architecture, a security-sensitive application or container running on non-SGX CPU cores acts as a client, while the actual enclave code executes on SGX-enabled cores. Hares intercepts `ecall/ocall` executions and manages control flow redirection across the nodes.

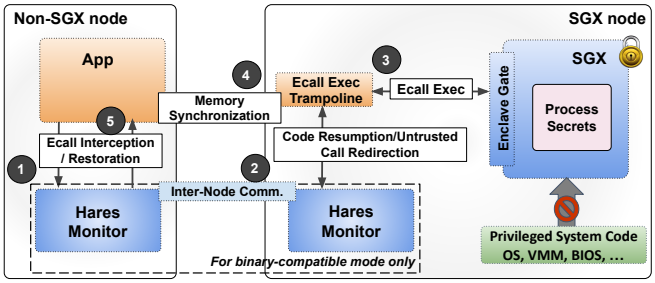


Figure 1: Overview of the Hares system design.

To transparently support enclave offloading of *unmodified SGX binaries* (*e.g.,* from an AMD x86 machine to an SGX machine), a Hares monitor handles page faults from specific memory regions of the target process and maintains a distributed shared memory region to synchronize data. This enables the target application to seamlessly track any memory updates caused by remote enclave execution. The Hares monitor is designed as a userspace program, akin to a Linux debugger (*e.g.,* `GDB`), and operates on commodity software/hardware stacks for simplified deployment.

Hares also provides a remote procedure call (RPC) mode, enabling the offloading of enclaves from architectures not natively supported by the Intel SGX SDK, such as non-x86 architectures. This capability addresses the limitation of the Intel SGX SDK, which can only compile programs for the x86 architecture. In RPC mode, there is no need to modify the source code of SGX applications. Instead, an extended version of the Open Enclave SDK is used to partition the `enclave` and `host` code across different architectures and generate the corresponding binaries. The primary motivation for introducing these two offloading modes is to overcome the architectural restriction of the Intel SGX SDK, which exclusively supports Intel `x86_64` systems.

3.1 Code Generation and Execution Split

Hares supports enclave offloading for both *homogeneous architectures* (*e.g.,* an Intel SGX-enabled CPU and an AMD CPU) and *heterogeneous architectures* (*e.g.,* an Intel SGX-enabled CPU and an ARM CPU). However, the methods for code generation differ slightly. For a homogeneous architecture scenario, Hares supports vanilla SGX applications compiled using both Intel SGX SDK [24] and the Open Enclave SDK [41]. In particular, Hares supports unmodified SGX binaries to transparently execute on non-SGX x86 machines. The Hares monitor automatically instruments `ecall`s and `ocall`s to redirect the control flow between machine nodes of the same x86 architecture. This allows confidential computations to be isolated from the rest of the program and seamlessly offloaded to an SGX-enabled CPU core.

An issue arises when generating an SGX-enabled program binary for a *heterogeneous architecture* scenario. The Intel SGX SDK does not support code generation on architectures other than Intel x86. Similarly, the Open Enclave SDK, despite aiming for multi-platform support, currently only generates `enclave`/`host` code for Intel architectures [41]. Therefore, to support confidential computation on embedded devices in heterogeneous environments, application code must be compiled and split across different architectures. To address this challenge, we extended the Open Enclave SDK to automatically generate cross-architecture `enclave`/`host` code using cross-architecture compilation.

Both SGX SDKs allow users to define enclave interfaces and provide tools to transform these interfaces into auxiliary interface files (`*_u.c, *_t.c, *_args.h`) [41]. These auxiliary files include trampoline code for enclave functions, a function table for enclave-host communication (including system calls), data structures for operating system resources, and code for parameter marshaling. As previously mentioned, the `enclave`/`host` code generation typically involves two steps: generating the auxiliary interface files from the *.edl* file, and compiling the `enclave`/`host` code to generate binaries.

In Hares's RPC mode, this process is modified to support execution on heterogeneous architectures. Specifically, the RPC mode introduces a helper library for host code generation. This library replaces `ecall` functions with a dispatcher function. Upon each `ecall` to the enclave, the dispatcher function identifies the caller, marshals function parameters (including memory dereferences for pointer types), and forwards the `ecall` request. It's important to note that replacing `ecall` functions is achieved by modifying the `oeedger8r` tool, which is responsible for generating SGX auxiliary interface files. This modification process is transparent to the application source code.

3.2 The Hares Code Monitor

To transparently offload unmodified SGX binaries, Hares deploys a code monitor on each machine node to manage the execution of partitioned enclave programs (❶ in Figure 1, Hares Monitor). The Hares monitor is responsible for several core tasks, including launching the host and enclave components, establishing communication channels between distributed nodes, and synchronizing memory pages shared between the host and enclave instances.

To launch an enclave program, the Hares monitor first establishes a TCP/IP connection with the client (❷ in Figure 1). Optionally, this channel can be secured using TLS to ensure the confidentiality and integrity of communication. Once connected, the Hares monitor spawns the SGX application as a child process and executes it until it reaches an `ecall` (❸ in Figure 1). Currently, we require users to manually specify the addresses of `ecall`s to enable Hares to intercept the execution of unmodified enclave binaries. In future work, this step could be fully automated using binary disassembly and reverse engineering techniques, supplemented by interface information from the `.edl` file [41].

The Hares monitor is also responsible for synchronizing necessary memory pages between the client and the enclave server (❹ in Figure 1). This step is essential because function calls between the enclave and the host may involve arguments that include both immediate values and memory pointers. A pointer may reference a virtual memory page that is not yet mapped on the remote node. To ensure correct execution, the Hares monitor synchronizes all memory pages that are dereferenced during each transition between the client and the enclave.

One approach to transparently synchronize memory pages is to modify the memory management code within the operating system (OS) kernel. This modification enables the OS kernel to track unmapped pages or any data pages that are written by one node (dirty pages) [6, 30]. However, deploying and maintaining a customized OS kernel is often impractical, especially on embedded devices or systems where users lack administrative privileges.

To address this challenge, Hares leverages a user-space page fault handling mechanism in Linux called `userfaultfd` [47]. This interface allows an application to register a file descriptor for a designated virtual memory area (VMA), such that any page fault occurring within that VMA generates an event on the registered file descriptor, enabling user-space handling of the fault. To enable transparent memory synchronization, Hares initially marks all mapped data pages as non-writable. When a write occurs, the Hares monitor receives a page fault notification identifying the

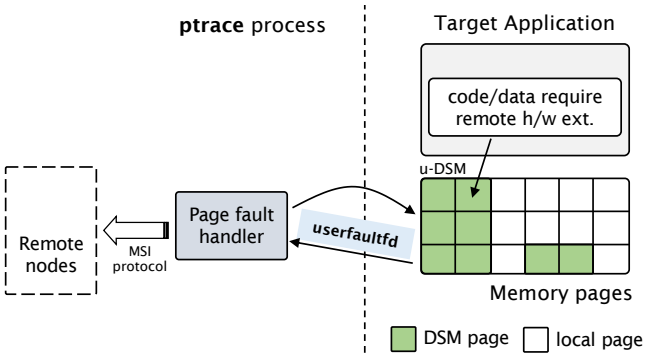


Figure 2: Hares monitor for userspace DSM support.

modified page (Figure 2). Using this information, Hares selectively synchronizes only the modified pages, significantly reducing the overhead associated with memory synchronization.

This design introduces an additional challenge: a `userfaultfd` file descriptor must be registered from within the target address space, while Hares handles page faults externally from a separate monitor process. Furthermore, the `enclave`/`host` code may modify the virtual memory layout at runtime. Such behaviors complicate our objective of transparently splitting and offloading enclave execution, as they violate the requirement of supporting unmodified applications without altering their source code.

To address these issues, we extended the CRIU/`compel` utility [11] to inject a piece of parasite code into the target process. This injection process remains transparent to the target process and external observers. Within our design, the parasite code creates new anonymous memory regions using `mmap()` or extends the heap region using the `sbrk()` syscall from the application context. Additionally, the parasite code allocates the `userfaultfd` descriptor and passes it to the Hares monitor via the `compel` utility [11]. As a result, the Hares monitor runs in a separate address space from the target process while still transparently intercepting its page faults and synchronizing the required memory pages with the remote node. For enclaves that access host memory outside explicit `ocall` interfaces, Hares similarly relies on `userfaultfd` to intercept page faults, provided that the accessed memory resides within VMAs tracked by the runtime.

Hares leverages a lightweight invalidation-based protocol to keep memory coherent between two nodes by adopting a simplified version of the MSI (Modified, Shared, Invalid) coherence model [62]. In this design, memory pages are initially shared in a read-only state. When a write is attempted on a page, the protocol ensures that the page is marked as *Modified* on the writing node and *Invalid* on the other, maintaining a consistent view of memory. Page ownership is transferred on demand via user-space page fault handling, allowing Hares to synchronize only the necessary pages during execution transitions. This approach enables transparent and efficient shared memory across nodes without requiring any changes to the application or underlying operating system.

3.3 The Hares RPC mode

Hares also facilitates applications running on different architectures (*e.g.,* ARM64) to offload enclave execution to a remote SGX enclave. We extend the Open Enclave SDK to support cross-architecture compilation, generating a `host` program binary in ARM64 format and an `enclave` binary in Intel x86_64 format. Additionally, Hares instruments the remote procedure call (RPC) stubs at each `ecall` site with a code snippet that serializes the arguments.

However, a naive approach to marshaling arguments fails when pointers are involved, due to potential differences in code and data layouts across architectures [6, 55, 59]. For instance, stack objects are likely allocated at different memory locations due to variations in available registers [55, 59]. Fortunately, Open Enclave provides a robust solution for marshaling `ecall` arguments, which Hares can effectively leverage. Specifically, the `oeedger8r` offers an API that distinguishes the type of `ecall` arguments, such as whether a field is a pointer, a non-pointer value, or a buffer address. Hares extends this capability within the `oeedger8r` to automatically generate an `ecall` marshaling structure. This structure includes fields that point to a *deep copy buffer* of data structures in the argument list.

Figure 3 depicts an example of such an automatically generated struct from an enclave call with the following declaration:

```
int seal_data(int sealPolicy,
              unsigned char* opt_mgs,
              ... ,
              data_t* sealed_data);
```

The `oeedger8r` recursively scans through data structures in the argument list. When encountering a field of pointer type, the `oeedger8r` appends the pointer dereference to a deep copy buffer and updates the pointer to reference the corresponding memory object within this buffer (as illustrated in Figure 4). Copying this marshaling structure and deep copy buffer directly to a remote node is impractical due to differing architectures and memory layouts. Instead, Hares extends the `oeedger8r` to convert pointer values within the deep copy buffer into offset values. These offsets indicate the position of the memory object relative to the start of the deep copy buffer address. Upon receiving this serialized data, the remote node deserializes the deep copy buffer and converts the offset values back to actual memory addresses. This approach enables Hares to execute enclave calls as remote procedure calls on a machine with a different CPU architecture without necessitating modifications to the application source code.

4 Implementation

We implemented a prototype of Hares from scratch on Linux.[1] The Hares framework consists of 3,350 lines of C code, with the majority dedicated to memory synchronization. As described earlier, Hares initially forks the SGX application as a child process. After the child process is created, the Hares monitor running on both the non-SGX node (Hares-c) and the SGX node (Hares-s) pauses the SGX application's execution at the beginning of the main function. This pause allows the operating system to set up the necessary program state, such as stack frames and library loading. Once the

[1] Source code: https://github.com/ssrg-vt/hares.

```
/**** ECALL marshaling struct. ****/
typedef struct _seal_data_args_t
{
    oe_result_t oe_result;
    uint8_t* deepcopy_out_buffer;
    size_t deepcopy_out_buffer_size;
    int oe_retval;
    int sealPolicy;
    unsigned char* opt_mgs;
    ... ...
    data_t* sealed_data;
} seal_data_args_t;
```

Figure 3: An example of the marshaling struct auto-generated by the `oeedger8r`.

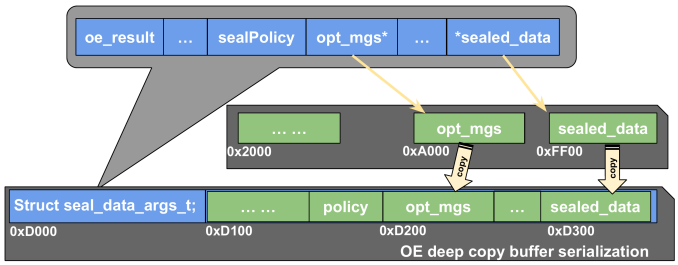


Figure 4: Serialization of a data structure using a deep copy buffer and the pointer to offset conversion.

initialization is complete, both Hares monitors establish socket communication to facilitate data exchange.

While Hares-s waits for requests from Hares-c, Hares-c sets breakpoints on user-provided instruction addresses that issue enclave calls. Before resuming the application, Hares-c determines which pages need to be migrated in the event of a transition for efficient memory synchronization. To accomplish this, Hares-c uses the CRIU/`compel` utility and the `userfaultfd` mechanism in Linux. On the non-SGX node side, Hares-c scans the application's memory for write-permitted VMA (virtual memory area) regions and registers the `userfaultfd` for each of the write-permitted VMA from the application's context as write-protected pages using compel's parasitic injection. Once the `userfaultfd` descriptors are registered, Hares-c retrieves them through a socket established with the parasitic code and resumes the application to execute until the process reaches any of the breakpoints set by Hares-c. At the same time, Hares-c forks a `userfaultfd` handler thread to handle any write protection faults by unprotecting pages and noting any changes incurred. All the `userfaultfd` descriptors registered for taking write faults are processed by the `userfaultfd` handler thread with write unprotection.

Another important consideration is ensuring that, during the registration of the `userfaultfd` for all the write-permitted regions, we ignore write-protecting the VMA of the application that belongs to the *parasitic code*. Otherwise, the parasitic code would encounter a write fault and wait for the application to handle it through the `userfaultfd`, which the application would be unaware of, resulting in a deadlock.

To solve this issue, once a breakpoint is hit, Hares-c must take note of any new VMA regions created or deleted, as the application can create or delete VMAs dynamically. Along with the modified pages, Hares-c must monitor these changes but exclude the VMA changes due to the parasite code creation and destruction. After noting any new VMA regions, Hares-c initiates a request to the remote Hares-s using internode APIs. The information about all the modified pages belonging to each VMA region and the executed instructions, along with the register states, is sent to the remote node. This communication occurs synchronously, meaning the client blocks until the server acknowledges and executes the complete enclave calls. Upon receiving the request from Hares-c, Hares-s sets a breakpoint on the instruction to be executed and migrates the necessary pages from the client to the server. The first step in the migration process is to check for the presence and boundaries of the process's VMA. For any VMA regions that are absent or extended, Hares-s uses parasitic code to align them accordingly. Once all the VMA regions are aligned, Hares-s updates the pages in the corresponding process's address space using `ptrace` APIs.

To ensure consistency, Hares-s employs the same `userfaultfd`-based technique used by Hares-c to monitor modified pages. After the enclave call execution is completed, the modified pages are migrated back to the client side using the same internode APIs but in the reverse direction. The migrated pages are reintegrated into the client process's memory space, and control is returned to the original application. After each transition, Hares on both sides must set all the write-unprotected pages back to write-protected. This is necessary to track pages written during subsequent execution phases. This transition process is repeated for each enclave call that needs to be executed until the application completes its execution.

Inter-node communication: The Hares framework utilizes inter-node TCP/IP APIs to transfer essential information for remote application execution, such as memory and register states, and to exchange synchronization messages. Optionally, Hares allows establishing a TLS communication channel to secure the messages. Table 1 lists the synchronization APIs used and their corresponding payload sizes.

Transitions can occur in both directions, with the `REMOTE_EXECUTE` header being the initial message. When transitioning from Hares-c to Hares-s, the instruction address up to which execution must proceed is sent as the payload. Hares-s reads this payload to set a breakpoint in the application and facilitate the transition back to Hares-c. It is worth noting that when transitioning from Hares-s to Hares-c, the payload could be empty because the client-side application will execute until it hits the breakpoint set by Hares-c. The second message header is `REMOTE_REGS`, with the payload consisting of the register states themselves. This information is necessary to correctly set the application's register sets and execute the required portion of the code.

The following message header is `VMA_FROM_REMOTE`, which indicates the number of VMA regions or pages that will follow in subsequent messages. The message header `VMA_BUFFER_HEADER` then provides additional information, such as the number of pages and the starting address of the application's memory to follow in the

subsequent message. The subsequent message, VMA_BUFFER, contains the page's payload. Finally, the communication ends with the receiver sending VMA_TRANS_ACK to acknowledge the previously mentioned message headers. In Table 1, the largest message is the VMA_BUFFER message, which is 5004 bytes in size. This message is the largest because it contains the page itself, which is typically 4096 bytes, along with additional information such as the page address.

Table 1: Inter-node TCP/IP messages with its payload size.

Messages	Bytes transferred
REMOTE_EXECUTE	16
REMOTE_EXECUTE_REPLY	8
REMOTE_REGS	220
REMOTE_REGS_REPLY	8
VMA_FROM_REMOTE	16
VMA_FROM_REMOTE_REPLY	8
VMA_BUFFER_HEADER	28
VMA_BUFFER_HEADER_ACK	8
VMA_BUFFER (Per page)	5004
VMA_BUFFER_ACK (Per Page)	8
VMA_TRANS_ACK	8

5 Evaluation

We evaluated the performance overhead and security benefits of Hares in both cloud and IoT/edge computing scenarios. The evaluation setup consisted of two machine nodes: an SGX-enabled machine and a non-SGX machine. For the IoT/Edge environment, we used a Raspberry Pi 4b and a Lenovo laptop with SGX support. To simulate the cloud environment, we conducted experiments using two machines from CloudLab [15].

Table 2 shows the hardware configurations used in our experiments. We performed evaluations with all benchmark applications running inside *Docker containers* (using Ubuntu 20.04 LTS as the base image), hosted across two nodes running on Ubuntu 20.04 LTS with Linux Kernel 5.15.

Table 2: Experimental hardware setup.

Settings	Heterogeneous offloading			Homogeneous offloading
CPU	Cortex-A53 (ARMv8)	Intel Core i7-8650C	Intel Core i7-9700	Intel Xeon P-8370C
Cores	4	2	8	2
Clock	1.4 GHz	1.9 GHz	3.0 GHz	2.8 GHz
RAM	1 GB	16 GB	32 GB	16 GB
Intel SGX	No	Yes	Yes	No
Interconnect	1 Gbps Ethernet			10 Gbps Infiniband
Network latency	1.700 ms			0.150 ms

5.1 Performance Evaluation

We first evaluated Hares's overhead by measuring the time cost breakdown for each phase of the enclave offloading. Next, we reported the performance overhead of running SGX-based machine learning applications and file encryption applications on non-SGX machines using Hares. Specifically, we compared the performance of *SGX-DNet* [67], *Plinius* [68], applications from the Open Enclave SDK, and *bw-mem*, *lat-rand*, two applications from an SGX-ported version of *lmbench* [21] with and without enclave offloading.

Table 3: Execution time breakdown of Hares APIs in a cloud computing setting.

Hares APIs	Avg. time taken
Compel APIs	0.5 *ms*
UFFD APIs	12 *us*
PTRACE_GETREGS / SETREGS APIs	10 *us*
PTRACE_POKETEXT / PEEKTEXT APIs	6 *us*
process_vm_readv() / writev() APIs	5 *us*
REMOTE_EXECUTE	2 *us*
VMA_FROM_REMOTE	4 *us*
VMA_BUFFER_HEADER	20 *us*
VMA_BUFFER	50 *us*
VMA_TRANS_ACK	10 *us*
REMOTE_REGS	20 *us*

5.1.1 Performance breakdown of a remote enclave call. Hares uses several key mechanisms to transparently offload enclaves to an SGX-enabled machine, such as the ptrace APIs, the userspace page fault handling (*i.e.,* userfaultfd), and the CRIU/compel framework for extending virtual memory ranges. Understanding these components' interplay is essential to identify potential bottlenecks and better optimize Hares. We inserted timestamps in the Hares debug mode and printed the time consumed by each Hares API call into a log file. We experimented on two bare-metal cloud machines (using 10 Gbps Infiniband as interconnection) from the CloudLab testbed [15].

As reported in Table 3, most Hares internal API calls consume less than 50 *us*. One exception is the compel API for injecting parasite code into a running program consuming 0.5 *ms*. The compel APIs (whether steal the userfaultfd descriptor from the target or extend the target's virtual memory) take around 0.5 *ms* to complete. This latency can be attributed to the number of tasks needed to infect the victim process with the parasitic code, such as stopping the victim process, preparing the infection handler, executing the remote code, curing the victim, and finally, resuming the victim process. Fortunately, we don't need to frequently invoke compel once the target process is launched. The VMA_BUFFER message sending is the second time-consuming API in Hares, taking about 50 *us* to complete, because it transfers memory pages. Hares's memory synchronization overhead mainly arises from dirty-page tracking, page transfer, and page protection. As shown in Table 1 and Table 3, per-page transfer via VMA_BUFFER dominates this cost (5004 bytes/page, about 50 *us*), whereas userfaultfd handling is lightweight (about 12 *us*). Combined with Figure 6 and Figure 7, these results suggest that page transfer is a major contributor to the end-to-end overhead: workloads with more transferred pages, such as *File-Encryptor* and *Data-Sealing*, incur higher yet acceptable slowdown.

We also wrote a micro-benchmark to evaluate the overhead of remote enclave calls with different numbers of ecalls on both homogeneous and heterogeneous-architecture settings. The ecalls are empty function calls with a simple return. The result from the

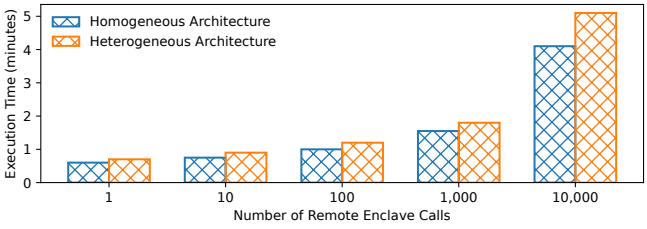


Figure 5: Execution of raw-overhead microbenchmark for different number of remote enclave calls (from 1 to 10,000) on homogeneous and heterogeneous setups.

Table 4: Application execution time.

Application	Native	Offloading
SGX_DNet	97.6 s	98.6 s
Plinius	210 s	210.6 s
bw-mem	34.2 ms	34.7 ms
lat-rand	34.3 ms	34.6 ms
File_Encryptor	408 ms	562 ms
Data_Sealing	562 ms	672 ms

raw-latency overhead micro-benchmark is shown in Figure 5. Noticeably, the variation of executing one, ten, or even a hundred enclave calls is relatively small. This is because most of the execution time consists of the creation of the enclave on the server side, which happens only once, no matter the number of enclave calls that follow after that (unless, of course, there is a request to destroy and create a new enclave, but that is not the case on these experiments). We can also observe that executing empty `ecalls` in a homogeneous setting (*i.e.,* two `x86_64` nodes) is faster than executing empty `ecalls` in a heterogeneous setting (*i.e.,* an `ARM64` and an `x86_64` node). This is possibly because the network bandwidth and computation power is better in the homogeneous setting.

5.1.2 Performance of Hares with unmodified SGX application binaries. We conducted comprehensive performance tests using unmodified SGX binaries generated from the Intel SGX SDK. We used the same machines from the CloudLab for an end-to-end evaluation. Unfortunately, the CloudLab administrators disabled the SGX support in the BIOS[2]. Instead, we used a local machine with SGX enabled and ran the same docker instances under the same network latency as the CloudLab, emulated by the Linux `tc` utility [28]. Our tests included *SGX-DNet* [67] (an Intel SGX version of the Darknet machine learning library), *Plinius* [68] (a secure machine learning framework that uses Intel SGX for secure training of neural network models and persistent memory for fault tolerance), two memory intensive benchmarks – *bw_mem* & *latrand* (from an SGX ported *lmbench* [21]) and a *file encryptor*, a *data sealing* programs from the Open Enclave SDK [41]. We present the normalized results of offloading the enclave from a docker instance running on a non-SGX machine to an SGX-enabled machine node, with a baseline of running the vanilla enclave on an SGX-enabled docker instance

[2]We sent them requests in the user group but didn't get a response.

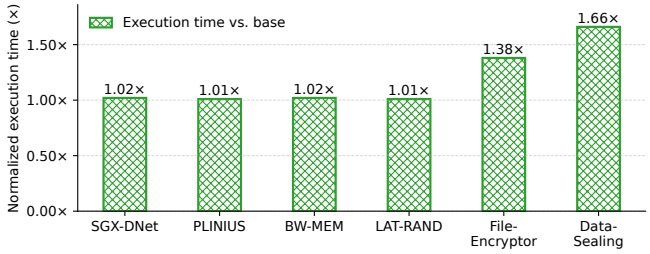


Figure 6: Normalized performance overhead of running offloaded enclave applications using Hares against local execution in the cloud setting.

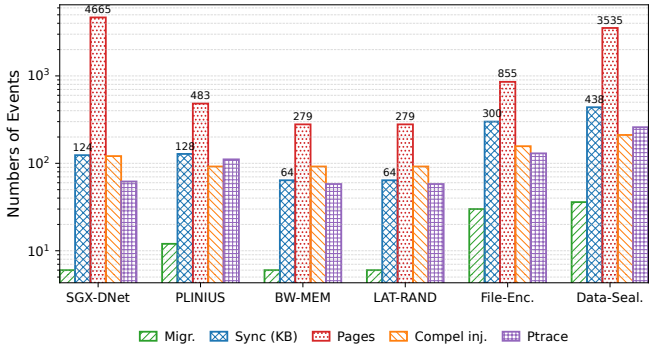


Figure 7: Profile of 5 events with application benchmarks.

alone. Figure 6 shows the normalized result. We also reported the total execution time in Table 4.

For applications with longer execution time (*e.g., SGX-DNet* and *Plinius*), Hares does not bring significant performance overhead. This is because the overhead of remote enclave execution becomes negligible in the total execution time and thus makes the overhead less significant. For applications with shorter execution time, the cross-node enclave offloading and control flow transformation constitute a more significant portion of the overall execution time (Table 4). Thus, applications such as *file encryptor* and *data sealing* have a larger normalized performance overhead.

To better explain the performance overhead that varies among different benchmarks, we further measured several other metrics, such as the number of enclave calls executed, synchronization messages used, pages transferred, injected parasite code calls, and the number of `ptrace` calls during the evaluation process (Figure 7). We measured the memory bandwidth and memory read latency using micro-benchmarks from the *lmbench* test suite [40]. The *lmbench* benchmark suite is a well-known set of benchmarks designed to measure basic operating system and hardware metrics. For our evaluation, we selected two benchmarks from an SGX-ported version of *lmbench* [21]: *bw_mem* and *lat_rand*. The *bw_mem* benchmark measures memory bandwidth, while *lat_rand* measures read latency. Both benchmarks exhibited similar characteristics, with a few hundred pages that needed to be synced and a few sync messages that needed to be sent (Figure 7). This resulted in a low overall overhead of about *1.01x* compared to native enclave execution.

We also evaluated Hares using machine learning and AI inference applications running with remote SGX enclaves [67, 68]. *SGX-Dnet* is a secure machine learning library that has been ported from the widely used Darknet library into Intel SGX [67]. By following the Darknet API, *SGX-Dnet* enables the secure training and testing of neural network models. In our performance evaluation, we utilized *SGX-Dnet* to conduct image classification through a pre-trained model. As shown in Figure 7, Hares exhibited a reasonable number of `ptrace` calls while handling *SGX-Dnet* because of the process's less frequent virtual memory layout changes and lesser cross-node control flow transformations. However, there was a larger number of pages, approximately 4665 pages, that had to be migrated between nodes, resulting in a higher number of synchronization messages as well. Despite this, the overhead was lower in comparison to other applications such as *file-Encryptor* or *Data-Sealing*. This lower overhead is primarily due to the framework's actual computations overshadowing its migrations, which is ideal for most computation-intensive real-world applications.

Plinius is another secure machine learning framework that leverages Intel SGX and persistent memory (PM) storage for fault tolerance [68]. During our evaluation, we simulated the persistent memory using DRAM (ramdisk), created a temporary filesystem (`tmpfs`), and mounted it at `/mnt/pmem0`. The *Plinius* framework utilizes two key components: `sgx-romulus`, an Intel SGX-compatible PM library that they ported from the *Romulus* PM library, and `sgx-dnet`, a port of the *Darknet* ML framework into Intel SGX. For our evaluation, we used *Plinius* to store encrypted data in the enclave, decrypt it within the enclave, and train the data in batches. By default, training is done in 500 batches with a total data size of 188 MB. However, we reduced the batch iteration number to one for our evaluation, as it takes a really long time to complete the entire training process. The application had about 12 cross-node control flow transformations and 483 page transfers, resulting in a negligible overhead.

We further tested Hares's performance using small applications with faster execution time. The *file-Encryptor* is a sample application from the Open Enclave SDK. It employs the mbedTLS library to carry out cryptographic computations on files within an enclave. To encrypt a file, it is transmitted to the enclave, undergoes encryption, and then returned to the user space. Similarly, a decrypted file is sent back to the enclave for decryption before being returned to the user space. Figure 6 indicates that the offloaded version takes approximately *1.38* times longer than native execution. Specifically, we observed about 30 migrations that must occur alongside the transmission of 300 synchronization messages with 855 page payloads in the *file-Encryptor* evaluation. *Data-Sealing* is another application included in the Open Enclave SDK that involves three enclaves: enclave_a_v1, enclave_a_v2, and enclave_b. Each enclave is signed by a different signer and has a distinct product identity. The application's primary function is to seal and unseal data using various policies while verifying the seal's integrity. This is achieved by creating a seal associated with specific hardware or software attributes and then verifying those attributes during the unsealing process. The application features *18* `ecalls`, which means there are 36 migrations between two nodes. The results were consistent with the *file-encryptor* application, with a more pronounced slowdown observed in the *data-sealing* application, resulting in about *1.66* times overhead to the baseline.

5.1.3 Performance of Hares with SGX applications running on different architectures. Hares supports SGX applications run on heterogeneous architectures without modifying the source code. Consequently, applications are required to compile using the Open Enclave SDK. During our evaluation, we found that converting applications from the Intel SDK to the Open Enclave SDK is not trivial. Moreover, very few SGX projects written in the Open Enclave SDK have their source code publicly available. Therefore, we chose five applications from the Open Enclave SDK as our evaluation targets. We also need to note that an IoT/edge environment with heterogeneous architectures typically has weaker processor cores and slower network connection; directly comparing the performance of heterogeneous offloading to homogeneous offloading is an apples-to-oranges comparison.

Figure 8 reports the normalized results of enclave execution from an IoT device to an `x86_64` laptop with SGX support. The baseline is to run the enclave/host code only on that `x86_64` laptop. Applications, such as *file-encryption* and *data-sealing*, take *4x* execution time compared to running them locally on an `x86_64` laptop. The main reason is due to the weaker CPU power of the IoT device. The split code execution also contributes about 40% to 66% of the overhead according to Figure 6. To further confirm our conjecture, we profiled the time consumption in each execution phase of all five applications by adding the timestamps. The results are shown in Figure 9. Networking performance significantly impacts the application's overall performance, with a considerable amount of time spent transferring bytes back and forth. Another interesting observation is that serialization and deserialization of RPC calls takes minimal time, just a few milliseconds.

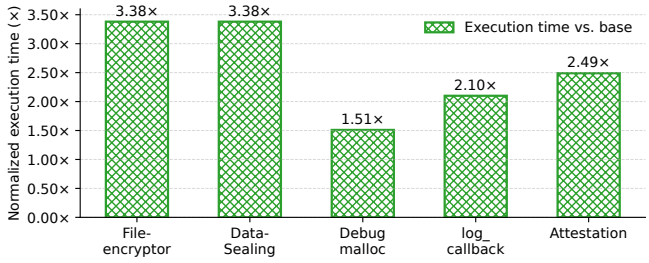


Figure 8: Normalized performance overhead of running heterogeneous offloaded enclave applications using Hares RPC mode against local execution in the cloud setting.

Power consumption: We measured the wall-plug power of a small cluster consisting of a Raspberry Pi 4 and an off-the-shelf Lenovo ThinkCentre M920s desktop (Intel Core i7-9700 CPU) using an external hardware power meter (a Suraielec watt meter), comparing local execution on the Raspberry Pi with enclave offloading. Across the evaluated workloads, enclave offloading increased average cluster-wide power consumption by less than 2%. This small difference is consistent with the fact that remote enclave invocations in our workloads were short and infrequent, and thus did not noticeably increase overall energy usage.

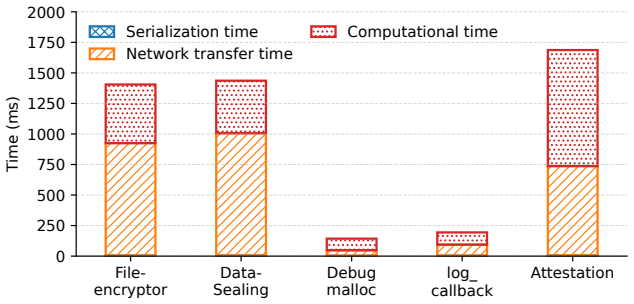


Figure 9: Performance Characterization of Applications with Hares RPC Mode.

5.2 Security Analysis

The main goal of Hares is to bridge the security hardware availability gap in an interconnected, hybrid-CPU environment. Hares provides a distributed shared memory to transparently hide the memory inconsistency of different machine nodes while still being able to expose the confidential computation capability across the machine boundary. With Hares, applications that run on non-SGX devices can still utilize hardware extensions available on a remote machine with SGX enabled.

When there is a need for confidential computation, Hares forwards the `ecall` request to a remote SGX-enabled node. Then Hares redirects the control flow to the remote enclave code. The remote enclave code will not be aware of anything different even when it tries to access memory pages not mapped on the remote side. Once there is a data access, the Hares monitor receives a page fault and retrieves the missing memory page from the original node. Although Hares involves multiple components in a remote enclave call, the security guarantee for the enclave code and data remains the same as in a local enclave execution. The only difference is that Hares brings a relatively larger attack surface for the *enclave interface attack* [29, 33], as a privileged attacker can hijack the Hares code for conducting an enclave interface attack. However, we must note that for an unprivileged attacker or an attacker only with the privilege to access the target process space, Hares does not bring an additional attack surface. Because such attackers cannot prevent or intercept the page fault handling as the page fault handler runs within the Hares code monitor's address space. Whenever the confidential computation is finished, the control flow transfers back to the original node, and the target application can only see the encrypted memory pages.

We further enumerate and discuss the potential types of CVEs that can be protected by leveraging the remote SGX as offered by Hares. The enclave plays a pivotal role in mitigating prevalent weaknesses and vulnerabilities, such as CWE-200 (Information Exposure), CWE-798 (Use of Hard-coded Credentials), CWE-321 (Hard-coded Cryptography Key), CWE-326 (Inadequate Encryption Strength), and numerous others. Unfortunately, these kinds of vulnerabilities remain prevalent in IoT systems. By offloading security-critical code and data to a remote SGX enclave, Hares can substantially reduce the attack surface and limit the consequences of their exploitation.

Table 5: CVEs in an IoT/edge environment that could be mitigated with Hares.

CWE	CVE
Buffer overflows	CVE-2017-10720, CVE-2017-10722, CVE-2017-9392, CVE-2017-9391, CVE-2023-25668, CVE-2023-37458, CVE-2022-41910
Hardcoded Credentials	CVE-2021-33220, CVE-2023-37468, CVE-2021-33219, CVE-2021-33218, CVE-2015-4080, CVE-2017-13717, CVE-2017-8229
Information exposure	CVE-2019-0741
Hardcoded cryptographic keys	CVE-2015-4080
Inadequate encryption strength	CVE-2019-13604, CVE-2019-13603

Table 5 lists some of these CVEs in IoT environment that could be mitigated by deploying Hares. For example, CVE-2019-13604 represents a vulnerability arising from a short key weakness. It enables potential attackers to employ brute force techniques to extract the key for image obfuscation, granting them access to decrypt the protected image. With Hares, the encryption key can be initially generated within the enclave and securely sealed in protected storage for future encryption and decryption of images. In this case, even if an attacker manages to compromise the host system, they cannot access the enclave or extract the encryption key, rendering the brute force attack ineffective. Similarly, consider CVE-2023-37468, where passwords are stored in clear text within the database, devoid of encryption. This vulnerability can also be averted by utilizing Hares SGX to encrypt the passwords and the database using the SGX-based SQLite-like databases that Hares support, thereby enhancing their security.

Another commonly encountered attack is the Return-Oriented Programming (ROP) attack [48]. In this attack, the attacker adeptly manipulates the stack, exploiting buffer overflow vulnerabilities in the program to execute arbitrary code. Successful ROP attacks often require knowledge of the state of registers and memory addresses. However, SGX offers protection by encrypting the entire memory contents of the enclave program along with an additional feature of providing an encrypted enclave code. This will significantly raise the bar for exploiting vulnerable code within an enclave unless specified through the allowed `ecall` entries. Furthermore, it's important to note that while SGX itself can be vulnerable to ROP-based attacks stemming from memory corruption vulnerabilities within the code residing in an enclave as illustrated by [32], our focus here is specifically on addressing the protection mechanisms of the Hares enclave against memory vulnerabilities originating from untrusted code. This approach can reduce exploitability of CVEs such as CVE-2005-0633 and CVE-2023-25668.

In addition to the CVEs mentioned above, we also conducted a synthetic attack using the OpenSSL heartbleed (CVE-2014-0160) vulnerability. In particular, it is a buffer overflow that exposes the server's secret key. The flaw is triggered when an attacker tells the server to send a specific message of a larger size but sends a shorter message; an un-patched version of the OpenSSL library will not check the number of bytes. We first replicate the attack with a vulnerable version of the OpenSSL tool (version 1.0.1). Subsequently, we repeat the experiment but set as server our own C implementation so that we can protect it later with Hares. Finally, we secure the server communication with the malicious client using Hares.

In particular, this requires making the vulnerable custom server a client of the framework and writing an external server that can back up the vulnerable memory via trusted enclave protection. Using Hares to create a server that backs up the client's memory and then sends it back after the `OpenSSL_read()` effectively protects the vulnerable server from the Heartbleed attack.

Remote Attestation: With Hares, remote attestation functions are similar to other use cases. Initially, the host process, residing in the non-SGX node, receives the challenge from the remote attestation service. This challenge is subsequently decrypted and signed by the enclave in the SGX node, facilitated by Hares. Following the completion of the enclave's operation, the enclave report, along with the signature from the quoting enclave, will be sent to the remote attestation service from the host process in the non-SGX node. Finally, the host process in the non-SGX receives the verification report from the remote attestation service.

Middleware and Monitor Attack Surface. The Hares monitor operates outside the SGX enclave and is therefore part of the untrusted computing base. A compromised or malicious monitor may attempt to disrupt execution, replay or drop messages, or interfere with control-flow transitions and memory synchronization.

However, even if the monitor is compromised, the confidentiality and integrity of enclave-resident code and data are preserved. All sensitive memory pages remain protected within the SGX enclave boundary, and secrets are never exposed to the monitor. Moreover, an adversarial monitor may replay, delay, or drop messages exchanged between the client and the remote enclave. These attacks can be mitigated using standard integrity-protected communication mechanisms (*e.g.,* attestation-anchored TLS channels with session identifiers and nonces), which ensure message authenticity and freshness. Lastly, a compromised monitor can induce denial-of-service by delaying enclave calls, suppressing synchronization messages, or halting execution. Such availability attacks do not violate the core confidentiality or integrity guarantees of Hares and are inherent to the SGX threat model. Addressing availability under a malicious middleware is orthogonal to this work and left to future exploration.

6 Discussion and Limitations

Hares demonstrates that hardware-based trusted execution environments (TEEs) can be abstracted and shared across different nodes. This allows embedded devices, which typically lack direct TEE hardware support, to benefit from advanced security guarantees. This capability naturally leads to a broader, more promising question: *what other hardware security primitives can be remotely shared in a similar way?*

Through our work, we have found that several hardware extensions appear to be promising candidates for remote offloading. *Stateless and modular primitives* like secure key storage modules (e.g., secure elements [66]), hardware random number generators (RNGs) [65], and cryptographic accelerators (such as AES-NI [38] and ARM Crypto Extensions [37]) are particularly well-suited. These can often be abstracted via authenticated remote procedure call interfaces, anchored by attestation. Because these extensions are relatively self-contained, they can often be accessed securely without tight integration with the client's local execution context.

We also observed that other extensions, such as monotonic counters [39], and pointer authentication mechanisms [46], might be offloadable under specific conditions. However, these generally require stronger trust assumptions, more careful coordination, or workload-specific considerations. In contrast, some hardware features, including speculative execution mitigations [23] and memory tagging [36], are tightly bound to the physical device and processor environment. This makes them significantly more difficult to offload in a secure and meaningful way. Our observations suggest that while not all hardware security features are amenable to remote use, a significant subset can be effectively shared when combined with remote attestation and secure communication. The viability of this approach hinges on two critical factors: first, whether the remote hardware exposes an interface that can be securely virtualized and multiplexed, and second, whether the latency and bandwidth characteristics are acceptable for the specific workload.

Stateless or latency-tolerant primitives, such as digital signing, key derivation, or RNG sampling, are especially well-suited for remote usage. Conversely, real-time or interactive primitives, like secure display, are more challenging to virtualize without dedicated secure channels or client-side hardware enclaves. This vision aligns with ongoing efforts in Hardware-as-a-Service (HaaS) [18] and enclave virtualization [70], which both aim to decouple secure computation from physical hardware locality. Our approach complements these directions by proposing a runtime system that enables transparent, fine-grained offloading without requiring hypervisor support or kernel modifications. This significantly expands the deployment surface for TEEs to edge and embedded environments.

The Role of Emerging Interconnects: Beyond IP Networks. While Hares currently focuses on remote offloading over conventional IP networks, emerging interconnects like Compute Express Link (CXL) [8] are rapidly redefining disaggregated hardware. CXL enables high-bandwidth, low-latency, and cache-coherent memory and device sharing across CPUs and accelerators, blurring the lines of "remote" hardware [69]. This allows secure components, such as enclaves, to be physically decoupled yet perform close to local access. Integrating CXL could dramatically reduce overhead for memory-intensive workloads, facilitating direct mapping of remote enclave-backed memory pools or secure accelerators into a local address space. This positions Hares as a bridge between current software-based offloading and future hardware-assisted memory pooling, leveraging existing infrastructure while preparing for advanced interconnect standards. However, employing CXL for trusted offloading introduces new challenges, including managing shared trust domains and integrating attestation into the memory fabric, promising avenues for future exploration in composable, fabric-based architectures.

Limitations and Future Work. Hares currently requires manual specification of enclave entry points for intercepting `ecalls` in unmodified binaries. This step could be automated using enclave interface metadata, symbol information, or binary analysis techniques; thus, it does not represent a fundamental limitation of the design, but it does reduce deployment transparency in the current prototype. Exploring fully interception is left to future work.

Moreover, Hares is designed for embedded and edge devices that can run a Linux-based operating system with virtual memory support. Its transparent enclave offloading relies on mechanisms

such as user-space page fault handling and process introspection, which are not available on deeply constrained microcontroller-class platforms (*e.g.,* Cortex-M devices). Evaluating Hares on such hardware would require a substantially different design based on explicit RPC-style interactions rather than transparent shared-memory execution. Investigating this design point is an interesting direction for future work but is beyond the scope of this paper.

Finally, Hares currently assumes single-threaded execution during enclave offloading; extending it to multi-threaded applications would require coordinated thread suspension, register-state capture, and memory synchronization across concurrent threads. Hares protects communication contents but does not address network side channels, such as message sizes, timing, or enclave-transition patterns. Both issues are left to future work.

7 Related Work

The first category of related work aims to improve the usability and portability of hardware enclaves. Projects such as Open Enclave [41] and Asylo [13] provide frameworks for building enclave applications that are portable across different TEE platforms. Asylo, for instance, supports multiple security backends, including TEEs and virtual machines, and facilitates communication through secure enclave-to-enclave gRPC channels. In contrast, Hares focuses on transparent offloading of existing SGX applications without requiring source code changes. It achieves this by implementing a user-space distributed shared memory (DSM) layer, allowing unmodified enclave binaries to execute remotely while preserving the original application semantics.

Rust-SGX [56] and EGo [50] bring SGX support to memory-safe languages. Rust-SGX provides secure Rust bindings for Intel SGX APIs and C/C++ libraries, along with a formally verified memory model [56], while EGo enables Go programs to run inside SGX enclaves. In contrast, Hares targets transparent offloading of SGX workloads from embedded devices without native TEE support to remote enclaves. Although our current prototype supports only C/C++ applications, extending the SGX SDK and Hares monitor to Rust or Go should require only minor design changes.

SGXJail [60] presents another line of related work. It enforces enclave isolation by sandboxing the enclave within a restricted host process and forwarding ecall/ocall operations through a dispatcher. This design protects against malicious enclaves that may attempt to compromise host code. In contrast, Hares addresses a different challenge: bridging the availability gap of hardware-based security features across heterogeneous CPU architectures by enabling remote enclave execution. GAROTA [2] is a more recent system that provides root-of-trust (RoT) capabilities for low-end embedded devices. While it focuses on secure bootstrapping and device attestation, GAROTA does not offer a general-purpose trusted execution environment for unmodified applications. We view Hares as complementary to GAROTA, extending the RoT foundation to support runtime protection for legacy and off-the-shelf software.

The second category of related work focuses on supporting applications across heterogeneous processors with different instruction set architectures (ISAs). Various techniques have been developed to bridge ISA differences, including dynamic binary translation (DBT) [16], code offloading [57], and cross-ISA code migration [6, 14]. DBT translates small code units, typically basic blocks, from a source ISA to a target ISA at runtime [16], allowing binaries compiled for one architecture to execute on another.

Wang et al. [57] extend this idea by enabling cross-ISA binary offloading, where mobile devices offload computation to more powerful servers using a DBT engine, effectively reducing energy consumption. HIPStR [54] further combines native compilation with system-level simulation to support register state transformation and migration across ISAs. These efforts aim to maintain execution continuity across heterogeneous platforms. In contrast, Hares addresses a different but related problem: enabling security-sensitive execution on remote TEEs without requiring architectural compatibility, binary translation, or ISA unification. Hares maintains application transparency by leveraging runtime instrumentation and memory synchronization rather than full binary migration or transformation.

Other approaches have explored direct cross-architecture code migration on native hardware, avoiding the need for full-system simulation or emulation. For example, Popcorn Linux [6, 30, 58] addresses the programmability challenges of heterogeneous-ISA systems by compiling applications into migratable binaries and implementing a kernel-level distributed shared memory (DSM) mechanism. This allows synchronization of key memory regions such as the stack, heap, and global data during execution migration across nodes with different ISAs. HeterSec [59] investigates leveraging heterogeneous CPU architectures to diversify program states for enhanced security. However, it encounters challenges related to maintaining correctness across architecture-specific behavior and binary layouts. In contrast, Hares focuses on a different problem: offloading confidential computation to remote TEEs in heterogeneous CPU environments. Rather than migrating binaries across architectures, Hares enables unmodified applications (developed using the Open Enclave SDK or Intel SGX SDK) to execute securely on remote SGX-enabled servers, without requiring source-level changes or architecture-aware adaptations.

8 Conclusion

We presented Hares, a framework for transparently offloading confidential computing workloads across distributed, heterogeneous nodes. Hares combines an extended Open Enclave SGX SDK with a user-space runtime monitor built on `ptrace` and `userfaultfd`, enabling efficient interception of enclave transitions and memory synchronization. This design supports unmodified enclave applications across different CPU architectures. Our evaluation on open-source workloads shows that Hares provides strong security guarantees with modest performance overhead, making it practical for heterogeneous and resource-constrained environments.

Acknowledgments

We thank the anonymous reviewers and our shepherd Valerio Schiavoni for their insightful comments, which have greatly improved the paper. This work is partly supported by the US Office of Naval Research under grants N00014-18-1-2022, N00014-19-1-2493, and N00014-22-1-2672, and by the US National Science Foundation (NSF) under grant CNS 2127491. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

References

[1] Advanced Micro Devices, Inc. 2021. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.

[2] Esmerald Aliaj, Ivan De Oliveira Nunes, and Gene Tsudik. 2022. {GAROTA}: Generalized Active {Root-Of-Trust} Architecture (for Tiny Embedded Devices). In *31st USENIX Security Symposium (USENIX Security 22)*. 2243–2260.

[3] ARM, Inc. 2013. *True Random Number Generator (TRNG).* https://www.arm.com/products/silicon-ip-security/random-number-generator

[4] Arm Limited. 2021. Arm TrustZone Technology. https://developer.arm.com/ip-products/security-ip/trustzone.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. 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 16)*. USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[6] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the boundaries in heterogeneous-ISA datacenters. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 645–659.

[7] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: Why the system call API is a bad untrusted RPC interface. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 253–264.

[8] computeexpresslink.org. 2024. *CXL: Compute Express Link.* https://computeexpresslink.org/about-cxl/

[9] Luigi Coppolino, Salvatore D'Antonio, Giovanni Mazzeo, and Luigi Romano. 2019. A comprehensive survey of hardware-assisted security: From the edge to the cloud. *Internet Things* 6 (2019). https://doi.org/10.1016/J.IOT.2019.100055

[10] CRIU. 2025. Checkpoint Restore in Userspace. https://criu.org/Main_Page.

[11] CRIU Dev. 2025. Compel. https://criu.org/Compel.

[12] OpenEnclave Dev. Accessed: 2024-05-12. Open Enclave SDK. https://github.com/openenclave/openenclave/issues/4255.

[13] Asylo Developers. 2023. Asylo: An open and flexible framework for enclave applications. https://asylo.dev/.

[14] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. 2012. Execution migration in a heterogeneous-ISA chip multiprocessor. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, Tim Harris and Michael L. Scott (Eds.). ACM, 261–272. https://doi.org/10.1145/2150976.2151004

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[16] Kemal Ebcioglu, Erik R. Altman, Michael Gschwind, and Sumedh W. Sathaye. 2001. Dynamic Binary Translation and Optimization. *IEEE Trans. Computers* 50, 6 (2001), 529–548. https://doi.org/10.1109/12.931892

[17] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. 2021. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–36.

[18] Mesh Flinders and Ian Smalley. 2025. What is hardware as a service (HaaS)? https://www.ibm.com/think/topics/hardware-as-a-service.

[19] GDPR 2018. *General Data Protection Regulation - GDPR.* https://gdpr-info.eu/

[20] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 585–598. https://doi.org/10.1145/3037697.3037716

[21] Aisha Hasan, Ryan Riley, and Dmitry Ponomarev. [n. d.]. Port or Shim? (The Source Code). https://github.com/vsecurity-research/sgx-bench.

[22] HIPAA 1996. *Health Insurance Portability and Accountability Act of 1996 (HIPAA).* https://www.cdc.gov/phlp/php/resources/health-insurance-portability-and-accountability-act-of-1996-hipaa.html

[23] Guangyuan Hu, Zecheng He, and Ruby B Lee. 2021. Sok: Hardware defenses against speculative execution attacks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 108–120.

[24] Intel 2013. *Software Guard Extensions Programming Reference.* Intel.

[25] Intel 2019. *Control-flow Enforcement Technology Specification.* Intel.

[26] Intel 2019. *Intel 64 and IA-32 Architectures Software Developerś Manual.* Intel.

[27] The kernel development community. 2021. The Linux kernel user's and administrator's guide – Userfaultfd. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html.

[28] Michael Kerrisk. 2023. tc(8) — Linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html

[29] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 971–985. https://doi.org/10.1145/3373376.3378486

[30] Sang-Hoon Kim, Ho-Ren Chuang, Robert Lyerly, Pierre Olivier, Changwoo Min, and Binoy Ravindran. 2020. DeX: Scaling Applications Beyond Machine Boundaries. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020*. IEEE, 864–876. https://doi.org/10.1109/ICDCS47774.2020.00021

[31] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 38:1–38:16. https://doi.org/10.1145/3342195.3387532

[32] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in Darkness: Return-Oriented Programming against Secure Enclaves. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 523–539.

[33] Hugo Lefeuvre, Vlad-Andrei Badoiu, Yi Chen, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2023. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/assessing-the-impact-of-interface-vulnerabilities-in-compartmentalized-software/

[34] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. Flexos: Towards flexible os isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–482.

[35] Kai Li and Paul Hudak. 1989. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.* 7, 4 (Nov. 1989), 321–359. https://doi.org/10.1145/75104.75105

[36] ARM Limited. 2022. Armv8.5-A Memory Tagging Extension – White Paper. (2022).

[37] ARM LLC. Accessed: 2025-05-30. Cryptographic extensions. *https://developer.arm.com/documentation/101754/0624/armclang-Reference/Other-Compiler-specific-Features/Supported-architecture-features/Cryptographic-extensions* (Accessed: 2025-05-30).

[38] Intel LLC. Accessed: 2025-05-30. Intel® Advanced Encryption Standard Instructions (AES-NI). *https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html* (Accessed: 2025-05-30).

[39] Intel LLC. Accessed: 2025-05-30. Monotonic Counters. *https://www.intel.com/content/www/us/en/docs/dynamic-application-loader/developer-guide/1-0/monotonic-counters.html* (Accessed: 2025-05-30).

[40] Larry W. McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996*. USENIX Association, 279–294.

[41] Microsoft. Accessed: 2024-05-12. Open Enclave SDK. https://openenclave.io/sdk/.

[42] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. {Latency-Tolerant} software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 291–305.

[43] Bill Nitzberg and Virginia Mary Lo. 1991. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer* 24, 8 (1991), 52–60. https://doi.org/10.1109/2.84877

[44] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious {Multi-Party} machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*. 619–636.

[45] OpenEnclave. 2020. *Open Enclave SDK for OP-TEE OS.* https://github.com/openenclave/openenclave/blob/master/docs/GettingStartedDocs/OP-TEE/Introduction.md

[46] Qualcomm Technologies, Inc. 2017. *Pointer Authentication on ARMv8.3.* https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf

[47] Mike Rapoport. 2021. userfaultfd(2) — Linux manual page. https://man7.org/linux/man-pages/man2/userfaultfd.2.html.

[48] Rop September, 2023. *Return-oriented programming.* https://en.wikipedia.org/wiki/Return-oriented_programming

[49] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event /*

Koblenz, Germany, October 26-29, 2021, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 740–755. https://doi.org/10.1145/3477132.3483555

[50] Edgeless Systems. 2022. Build Confidential Go Apps with Ease. *https://www.ego.dev/* (2022).

[51] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 117–131. https://doi.org/10.1145/3373376.3378528

[52] Bill Toulas. 2022. New Intel chips won't play Blu-ray disks due to SGX deprecation. https://www.bleepingcomputer.com/news/security/new-intel-chips-wont-play-blu-ray-disks-due-to-sgx-deprecation/

[53] Rahmadi Trimananda, Ali Younis, Bojun Wang, Bin Xu, Brian Demsky, and Guoqing Harry Xu. 2018. Vigilia: Securing Smart Home Edge Computing. In *2018 IEEE/ACM Symposium on Edge Computing, SEC 2018, Seattle, WA, USA, October 25-27, 2018*. IEEE, 74–89. https://doi.org/10.1109/SEC.2018.00013

[54] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M Tullsen. 2016. Hipstr: Heterogeneous-isa program state relocation. In *ACM SIGARCH Computer Architecture News*, Vol. 44. ACM, 727–741.

[55] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. 2019. DMON: A Distributed Heterogeneous N-Variant System. *arXiv preprint arXiv:1903.03643* (2019).

[56] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 2333–2350. https://doi.org/10.1145/3319535.3354241

[57] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. 2017. Enabling Cross-ISA Offloading for COTS Binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'17, Niagara Falls, NY, USA, June 19-23, 2017*, Tanzeem Choudhury, Steven Y. Ko, Andrew Campbell, and Deepak Ganesan (Eds.). ACM, 319–331. https://doi.org/10.1145/3081333.3081337

[58] Xiaoguang Wang, Carlos Bilbao, and Binoy Ravindran. 2022. Transparent, cross-isa enclave offloading. In *The 5th Workshop on System Software for Trusted Execution (SysTEX 2022), Co-located with ASPLOS 2022, March 1, 2022, Lausanne, Switzerland*.

[59] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A Framework for Software Diversification with ISA Heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*, Manuel Egele and Leyla Bilge (Eds.). USENIX Association, 427–442. https://www.usenix.org/conference/raid2020/presentation/wang-xiaoguang

[60] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. 2019. SGXJail: Defeating Enclave Malware via Confinement. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. USENIX Association, 353–366. https://www.usenix.org/conference/raid2019/presentation/weiser

[61] Wikipedia. 2022. Trusted execution environment. *https://en.wikipedia.org/wiki/Trusted_execution_environment* (2022).

[62] Wikipedia. Accessed: 2024-02-14. MSI Protocol. https://en.wikipedia.org/wiki/MSI_protocol.

[63] wikipedia.org. 2025. Coherent Accelerator Processor Interface. *https://en.wikipedia.org/wiki/Coherent_Accelerator_Processor_Interface* (2025).

[64] wikipedia.org. 2025. *Non-uniform memory access.* https://en.wikipedia.org/wiki/Non-uniform_memory_access

[65] wikipedia.org. Accessed: 2025-05-30. Hardware random number generator. *hhttps://en.wikipedia.org/wiki/Hardware_random_number_generator* (Accessed: 2025-05-30).

[66] wikipedia.org. Accessed: 2025-05-30. Secure element. *https://en.wikipedia.org/wiki/Secure_element* (Accessed: 2025-05-30).

[67] XHercules. 2020. SGX-Darknet: SGX compatible ML library. https://github.com/anonymous-xh/sgx-dnet.

[68] XHercules. 2022. Plinius: Secure ML model training with Intel SGX and PM for fault tolerance. https://github.com/anonymous-xh/plinius.

[69] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 658–674.

[70] Shixuan Zhao, Mengyuan Li, Yinqian Zhangyz, and Zhiqiang Lin. 2022. vsgx: Virtualizing sgx enclaves on amd sev. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 321–336.

[71] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. ARMlock: Hardware-Based Fault Isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 558–569.