TrustOre: Side-Channel Resistant Storage for SGX using Intel Hybrid CPU-FPGA

Hyunyoung Oh
ECE & ISRC, Seoul National University

Adil Ahmad
Purdue University

Seonghyun Park
ECE, Seoul National University

Byoungyoung Lee*
ECE, Seoul National University

Yunheung Paek*
ECE & ISRC, Seoul National University

ABSTRACT

Intel SGX is a security solution promising strong and practical security guarantees for trusted computing. However, recent reports demonstrated that such security guarantees of SGX are broken due to access pattern based side-channel attacks, including page fault, cache, branch prediction, and speculative execution. In order to stop these side-channel attackers, Oblivious RAM (ORAM) has gained strong attention from the security community as it provides cryptographically proven protection against access pattern based side-channels. While several proposed systems have successfully applied ORAM to thwart side-channels, those are severely limited in performance and its scalability due to notorious performance issues of ORAM. This paper presents TrustOre, addressing these issues that arise when using ORAM with Intel SGX. TrustOre leverages an external device, FPGA, to implement a trusted storage service within a completed isolated environment secure from side-channel attacks. TrustOre tackles several challenges in achieving such a goal: extending trust from SGX to FPGA without imposing architectural changes, providing a verifiably-secure connection between SGX applications and FPGA, and seamlessly supporting various access operations from SGX applications to FPGA. We implemented TrustOre on the commodity Intel Hybrid CPU-FPGA architecture. Then we evaluated with three state-of-the-art ORAM-based SGX applications, ZeroTrace, Obliviate, and Obfuscuro, as well as an end-to-end key-value store application. According to our evaluation, TrustOre-based applications outperforms ORAM-based original applications ranging from 10× to 43×, while also showing far better scalability than ORAM-based ones. We emphasize that since TrustOre can be deployed as a simple plug-in to SGX machine's PCIe slot, it is readily used to thwart side-channel attacks in SGX, arguably one of the most cryptic and critical security holes today.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; *Security services*; Security protocols.

KEYWORDS

access pattern based side-channel; secure storage; hybrid CPU-FPGA; Intel SGX

ACM Reference Format:
Hyunyoung Oh, Adil Ahmad, Seonghyun Park, Byoungyoung Lee, and Yunheung Paek. 2020. TrustOre: Side-Channel Resistant Storage for SGX using Intel Hybrid CPU-FPGA. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3372297.3417265

1 INTRODUCTION

Intel Software Guard eXtension (SGX) is a processor extension that offers strong and practical security guarantees by excluding privileged system software and other unprivileged software from the trusted computing base (TCB). This processor extension provides software applications with shielded execution environments, called an *enclave*, where security-sensitive code and data can safely run in an environment isolated from the rest of systems. Using SGX, even privileged software components such as OSes and hypervisors are not allowed to directly access the enclave's memory.

However, recent reports have demonstrated that Intel SGX is vulnerable to various memory-based side-channel attacks (e.g. page-fault-based attacks [82], cache-based attacks [11], branch prediction [42], ForeShadow [73], RIDL [74], and Fallout [49]). We note these side-channel attacks pose real threats as demonstrated by recent studies — using these attacks, it has been shown that adversaries can completely nullify the confidentiality guarantee of Intel SGX through leaking sensitive information from enclaves.

In order to prevent these side-channel attacks against SGX, recent studies [3, 4, 60] have proposed using Oblivious RAM (ORAM) to protect access patterns leaked while accessing memory. Under the concept of ORAM [23], which provides cryptographically proven security resistant against access pattern based attacks, each data object is appended with multiple dummy objects and continuously shuffled after each access. Using ORAM, several protection systems have been proposed. ZeroTrace [60] ensures secure accesses to data structures, Obliviate [3] provides secure file systems, and Obfuscuro [4] guarantees black-box based secure program execution. While these applications have further security benefits and protection scopes, the key common feature is that they are all relying on ORAM protocols to thwart side-channels within enclaves.

However, ORAM is notorious for its slow performance and scalability issues, severely hindering its adoption in real-world scenarios. The ORAM protocol requires orders of magnitude larger memory bandwidth than a normal memory access, as it has to access an

*co-corresponding author

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417265

entire tree path for each memory access — it approximately requires $O(LogN)$ memory accesses per access, where N is proportional to the size of protected data. Based on the experimental evaluations in above-mentioned papers, the overhead of ORAM is at minimum two degrees over native enclave execution. More importantly, all these papers point to scalability issues of ORAM — as the size of protected data becomes larger, the performance exponentially slows down, a fact we experimentally verify in our evaluation as well.

This paper proposes TrustOre, a system that addresses performance and scalability issues that arise while using ORAM with Intel SGX. The main idea behind TrustOre is to leverage an external device, i.e., an FPGA, to implement a trusted storage service for enclaves. This is based on the observation that the root cause of memory-based side-channel issues in enclaves is because memory-related units (such as caches, page tables and branch prediction units) are designed to be shared with untrusted software. Therefore, we design the trusted storage on an isolated FPGA environment, having its own memory-related units isolated from other entities and as a result avoiding memory-based side-channels by design.

In order to adopt FPGAs with Intel SGX, TrustOre has to tackle following challenges: (a) extending trust from enclaves to FPGA without imposing architectural changes to involved components; (b) providing a verifiably-secure connection from enclaves to FPGA without imposing massive overheads; and (c) seamlessly supporting various access operations from enclaves to the FPGA without involving huge porting efforts. To elaborate, TrustOre relies on an external FPGA device, but the FPGA is not a trusted component in the Intel SGX architecture and therefore we need an additional mechanism to extend trust to it. Moreover, an FPGA architecture itself does not provide a mechanism to affirm its authenticity to others. To address this challenge, TrustOre designs a new attestation mechanism for the CPU-FPGA architecture such that enclaves can safely verify the authenticity of FPGA instance.

We highlight that TrustOre's attestation does not impose any hardware/architectural change, and thus it can be used with *commodity-off-the-shelf* Intel SGX hardware and FPGA devices. This is a notable difference from previous solutions, which require hardware/architectural changes, rendering their solutions limited to custom architectures and thus incurring fabrication challenges or high manufacturing costs. For example, hardware-assisted ORAM designs [21, 22, 43, 45] require adding special hardware or significant extra protections (for ORAM controllers in particular) within the host CPU, as shown by recent report [4]. As another example, previous works encrypting the communication channel between CPU and memory [2, 8, 63, 76] either require special hardware components in memory (i.e., processing-in-memory) or design a custom memory board. On the contrary, TrustOre is built on an Intel hybrid CPU-FPGA architecture for which its practical aspect is already evidenced in several places — for instance, Amazon EC2 already provides FPGA-builtin servers in Amazon Web Services [5].

Furthermore, TrustOre has to efficiently connect an enclave to the FPGA with security assurance. Since TrustOre outsources a memory access to the external FPGA device, the performance of an SGX application can be blocked by the extra communication delay between CPU and FPGA. More importantly, TrustOre has to assume that an attacker would be able to eavesdrop this communication which involves untrusted system components as well as untrusted memory. As such, in consideration of the *man-in-the-middle* attack model, TrustOre carefully designs its communication channel while ensuring following features: (a) all MMIO/DMA addresses remain constant and (b) all data written to these addresses is encrypted using a shared key established between the enclave and the FPGA module.

Lastly, TrustOre has to seamlessly support various memory allocation/deallocation as well as access operations from enclaves without involving huge porting efforts. To this end, TrustOre takes a *key-value store* model in designing its FPGA on-chip memory storage, and provides minimal and simple interfaces for enclaves to utilize TrustOre's key-value store. Since the key-value store model can be easily used to represent many different data resources and objects, TrustOre's storage model can easily fit the needs of different enclave applications. As a concrete case-study, we first implemented TrustOre based on a commodity hybrid CPU-FPGA architecture. Then, we develop extra APIs to fairly evaluate existing schemes such as ZeroTrace, Obliviate, Obfuscuro, while replacing their ORAM operations with TrustOre's trusted storage service. We also applied TrustOre for an end-to-end key-value store, ShieldStore [39].

According to our evaluation, TrustOre-based applications (which replaced ORAM with TrustOre) significantly outperforms ORAM-based original applications for all cases. TrustOre-based ZeroTrace accesses data structures with various block sizes 49× faster than ORAM-based ZeroTrace (i.e., an original ZeroTrace) on average. TrustOre-based Obliviate accesses the files of size 1GB about 10× faster compared to ORAM-based Obliviate. TrustOre-based Obfuscuro is faster than ORAM-based Obfuscuro, ranging from 2× to 43×. TrustOre-based ShieldStore is faster than ORAM-based ShieldStore about 188×. Particularly focusing on scalability , TrustOre-based applications showed far better scalability compared to its original ORAM version. While ORAM-based applications exponentially slow down as increasing the data size, TrustOre-based applications slow down smoothly or maintain the constant throughput irrespective to the data size (more details are in §7).

Given the trusted memory storage provided TrustOre, various interesting applications for an enclave can be built with side-channel security assurance. We believe TrustOre is an attractive solution particularly because it is ready-to-deploy and easy-to-deploy — by simply plugging-in the FPGA card to the available PCI-E slot, TrustOre can start serving the trusted memory storage at the gigabytes scale[1] without any hardware/architecture changes.

In summary, this paper makes the following main contributions:

- **Design: Extending Trust to FPGA.** To the best of our knowledge, TrustOre is the first system extending the trust of Intel SGX to FPGA. Unlike other related work [75] extending trust to an external device, TrustOre does not impose architectural/hardware changes, and is therefore readily deployable on *commodity-off-the-shelf* Intel SGX machines.
- **Design: Trusted Storage on FPGA.** TrustOre designs its trusted storage on FPGA, which tackles unique design requirements. TrustOre provides the secure connection between an

[1]Intel currently provides FPGA with 16GB packed DRAM in Intel Stratix 10 MX and Xilinx does 8GB in Virtex UltraScale+ HBM VCU128-ES1.

SGX application and the FPGA, and seamlessly supports the requirements of various enclave applications by maintaining a key-value store model on the FPGA memory, achieving efficient and scalable performance.

- **Case Study: Practical End-to-End System.** We implemented TrustOre on a commodity Intel hybrid CPU-FPGA architecture. Then, in order to stake our claim as a faster, and more scalable storage system than ORAM for SGX environments, we used TrustOre to re-implement three ORAM-based protection systems, including ZeroTrace [60], Obliviate [3] and Obfuscuro [4], and compared performance aspects between ORAM-based and TrustOre-based ones. Our results indicate that TrustOre-based schemes outperforms ORAM-based schemes by almost two degrees, demonstrating its scalability for real-world workloads.

2 BACKGROUND

2.1 Intel SGX

Intel SGX [48] is an extension to the x86 Instruction Set Architecture (ISA) available to processors starting from the Skylake architecture. These instructions allow a user-level process to allocate a trusted memory region called an enclave, which is only accessible to the enclave itself, and not to other user or system components, e.g., other processes, OS, hypervisor and BIOS. This memory region is allocated from a reserved memory on the DRAM called the Enclave Page Cache (EPC), which is initialized at the booting time. The EPC is currently limited to 128 MB, but this memory limitation can be alleviated by using page swap-in/out mechanism.

Side-channel Issues. Reports [72, 82] have shown that attackers can observe page-granular memory interactions being performed by SGX application using page faults or stealthily observing the access/dirty bit within enclave page tables. Using this information, researchers have demonstrated how to leak various information such as inferring rendered JPEG images and spell-checked words from enclaves. Furthermore, researchers [11, 61] have shown how to leak entire cryptographic keys from enclaves running mbedTLS [44] through prime+probe attacks [52] on various caches (from L1 to LLC) which are shared by enclave and non-enclave entities. Finally, branch prediction [19, 42] allows privileged entities to observe the branching history of the enclave thereby providing them fine-grained insights into the control-flow allowing for similar attacks to the ones mentioned above. More recently, various speculative execution attacks [13, 73] have been shown to affect SGX enclaves.

2.2 FPGA

A Field-Programmable Gate Array (FPGA) is a specially designed re-configurable integrated circuit. At a high-level, FPGAs consist of two components, the *infra primitive* and *fabric*. The infra primitive is a set of non-configurable hardware circuits. The fabric is a configurable circuit which is initialized by the infra primitive to realize a logic designed by the developer. With this two-staged circuit design, FPGAs offer developers a design flexibility as well as various security guarantees, as explained in the coming paragraphs.

Bitstream Preparation. The core of an FPGA module is a bitstream, which is the logic intended by the developer. This bitstream is compiled using the FPGA manufacturer's compiler-tools and is appended with a manufacturer-specific bootloader to form a raw

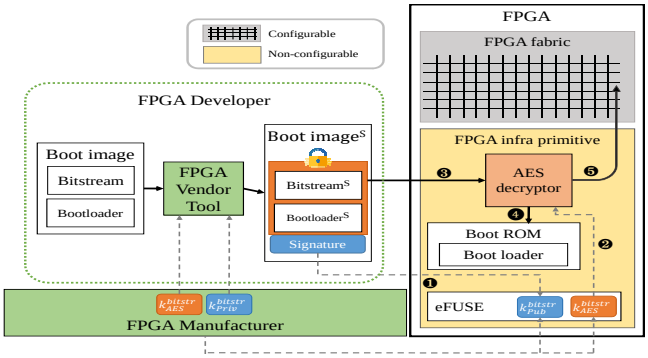


Figure 1: A generic secure boot architecture of FPGA

boot image. To protect the the FPGA design from copying or reverse engineering, the boot image is encrypted. Authentication additionally provides assurance that the boot image is genuine and created by an authorized developer, i.e., authentication verifies both data integrity and authenticity of the boot image. For such encryption and signing, it is required to generate and securely manage two kinds of keys: an RSA public/private keys for signing (i.e., $k^{\text{bitstr}}_{\text{Pub}}$ and $k^{\text{bitstr}}_{\text{Priv}}$) and an AES encryption key (i.e., $k^{\text{bitstr}}_{\text{AES}}$). To be more specific, each component of the boot image (i.e., boot loader and bitstream) is encrypted using $k^{\text{bitstr}}_{\text{AES}}$ and then signed using $k^{\text{bitstr}}_{\text{Priv}}$, in turn (bootloader^S and bitstream^S). So, $k^{\text{bitstr}}_{\text{AES}}$ and $k^{\text{bitstr}}_{\text{Priv}}$ should be protected from any others except an authorized user. Since FPGAs have one-time programmable and tamper-resistant storage for the keys as will be explained later, we assume the FPGA manufacturer bakes the key during manufacturing. Once all of the above steps are complete, bootloader^S and bitstream^S are now ready to be released.

Bitstream Loading. The on-chip FPGA infra primitive loads signed boot image using a *secure* boot process. The implementation details of this process depend on FPGA manufacturers, such as Secure Device Manager (SDM) [69] of Intel and Processing System (PS) of Xilinx [59]. Figure 1 provides a general outlook on the secure boot process in a hardware-agnostic manner. The boot ROM in the infra primitive loads the signed boot image from the host system. Then, it authenticates each component of the image using the RSA public key (i.e.,$k^{\text{bitstr}}_{\text{Pub}}$) (❶), and decrypts correctly authenticated ones using the AES key (i.e.,$k^{\text{bitstr}}_{\text{AES}}$) (❷,❸). After this, the bootloader is launched in the infra primitive (❹). Lastly, the bootloader loads the bitstream to the FPGA fabric (❺), and hands over the execution to the loaded bitstream.

Secure Boot Assumptions. FPGA manufacturers support a secure boot process of bitstreams to ensure authenticity and confidentiality. This security feature is designed to hold under the following assumption on key management and FPGA configuration. First, FPGA manufacturers (such as Intel and Xilinx) are assumed to be trusted and responsible to securely generate cryptographic keys (i.e., $k^{\text{bitstr}}_{\text{Pub}}$ and $k^{\text{bitstr}}_{\text{Priv}}$ for RSA authentication and $k^{\text{bitstr}}_{\text{AES}}$ for AES encryption) such that an adversary cannot obtain security sensitive keys including $k^{\text{bitstr}}_{\text{Priv}}$ and $k^{\text{bitstr}}_{\text{AES}}$. $k^{\text{bitstr}}_{\text{Pub}}$ and $k^{\text{bitstr}}_{\text{AES}}$ can be programmed to either one-time programmable eFUSE or multiple re-writable battery-backed RAM, which can flexibly facilitate the secure key setup. Second, all security sensitive code and data (including boot ROM, AES decryptor, and $k^{\text{bitstr}}_{\text{Pub}}$ and $k^{\text{bitstr}}_{\text{AES}}$) are assumed to be tamper-resistant by their design. The boot ROM functions as a *root-of-trust*

for ensuring confidentiality and authenticity of bitstream and therefore should not be tampered-with by system components.

It is worth to note that whereas Intel SGX supports remote attestation of an enclave application, current FPGA architectures do not employ such remote attestation feature. FPGA's secure boot only guarantees the authenticity and confidentiality of the bitstream during boot time. The reason why FPGAs do not have the remote attestation feature is arguably related to the fact that FPGAs are designed as a general purpose hardware components, e.g., prototyping new hardware logics or facilitating highly parallelizable computation. This is different from Intel SGX, which is designed as a security enhancing feature for trusted remote computation.

Connection with Host CPU. After the booting, the FPGA begins operation as programmed in the bitstream. When the bitstream is running, the FPGA is connected to the host CPU through the IO bus (QPI [31] or PCIe [80], which are often used in CPU-FPGA hybrid architectures). The FPGA device driver provided by the manufacturer extends host CPU memory space to include FPGA memory using memory mapped IO, such that the CPU can access FPGA address space in the same way as regular memory. Inside the FPGA, the IO bus signals from the host CPU are converted to its internal bus and transmitted to the programmed hardware modules or on-chip FPGA memory directly. The sender waits for an acknowledgment before initiating the next transmission.

3 THREAT MODEL

Enclave Assumptions. We assume that a user wants to run an application safely using a trusted SGX enclave on a remote machine. The application itself is public and known to the attacker and therefore, the code sections of the application are not security-critical. However, the data provided to the enclave by the user (e.g., a cryptographic key) is critical and has to be protected from side-channel leakage prevalent in Intel SGX. In the context of our work, we assume a single party, i.e., the enclave developer is the same entity who will bootstrap our service onto the FPGA device. However, the developer can run multiple enclaves which can concurrently access the FPGA device for some service.

Hardware Assumptions. Our hardware trusted base consists of the CPU chip and FPGA chip, which we assume are tamper-proof and correctly implemented. We assume that the attacker cannot directly extract secrets or corrupt state within the packages, as reverse engineering these packages is highly challenging. We note that, similar to commodity-off-the-shelf CPUs, current FPGAs have a 3D-stacked layer with 14 nanometer fabrication nodes[2], thereby introducing similar reverse-engineering challenges. Power [40], thermal [55] side-channels for FPGA and CPU packages are outside the scope of this paper.

Attacker's Capabilities. The attacker controls all privileged software components including BIOS, OS, VMM, and device drivers. Furthermore, they control all other external devices (e.g., storage, networking etc.) on the system except for the FPGA device. Therefore, while the attacker cannot directly access memory owned by the target application, they can still launch side-channel attacks such as page tables and cache attacks. Furthermore, all non-EPC

[2]Intel i7-9700 CPU has 14 nanometer fabrication nodes, while Xilinx UltraScale+ FPGA has 16 and Intel Hyperflex FPGA has 14 nanometer.

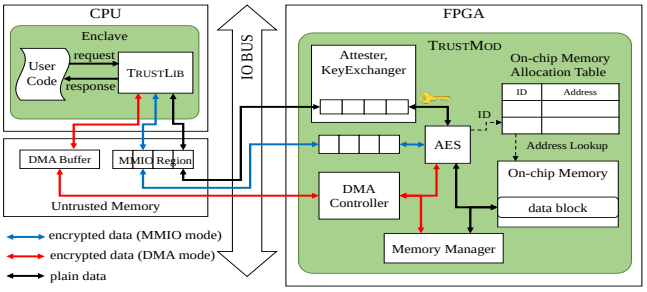


Figure 2: Design overview of TrustOre.

host memory (such as DMA/MMIO buffers) is assumed to be completely controlled by the attacker. Although the attacker cannot directly extract the data from the FPGA device memory protected by TrustOre's access control mechanism (see §4.2, §4.3), we assume that the attacker can launch side-channel attacks against the FPGA device memory (similar to DRAM-based side-channel attacks [54]). We further assume that the attacker has physical access to all hardware components such as FPGAs. Thus, the attacker can launch physical side-channel attacks, which snoop various buses on host—the host memory bus, the PCIe bus, and other exposed IO buses.

We note that TrustOre does not mitigate transient execution attacks (such as ForeShadow, RIDL and Fallout) which target the enclave program running on the CPU. Instead, TrustOre prevents side-channel attacks of the enclave's sensitive data by offloading the data to the FPGA. Attackers can still launch side-channel attacks against the enclave program, but the data is protected and the access pattern is also protected by TrustOre. Specifically, we design the FPGA modules and interfaces which are secure from various side-channel attacks (as will be shown in Table 1) and provide the side-channel resistant storage for sensitive data evacuated from the CPU. In FPGA, attacks like ForeShadow cannot be launched because no caches and prediction logics are inside the FPGA (see §4.2).

4 DESIGN

TrustOre is a trusted storage system for enclave applications implemented on an authenticated and attested external FPGA device. Figure 2 depicts a design overview of TrustOre. In general, TrustOre consists of two major components — TrustLib and TrustMod.

TrustLib is an in-enclave library required for establishing and maintaining the communication channel between the enclave program and the trusted storage service.

TrustMod is the core component implementing the trusted storage, which is a module loaded to the FPGA device.

In the coming subsections, we begin by describing how TrustOre extends trust from an SGX enclave (containing TrustLib) to TrustMod (§4.1). Since TrustMod is an external component (i.e., not located inside the enclave), we design a new remote attestation mechanism for FPGA along with a cryptographic scheme for securing communication between these entities. Next, we describe how TrustOre implements the trusted storage within TrustMod (§4.2). In general, TrustMod constructs a key-value store using the FPGA on-chip memory, providing a storage service capable of servicing multiple enclaves while enforcing access control ensuring the security of enclave data stored on the FPGA device. Lastly, we describe the two communication mechanisms (i.e., MMIO and DMA)

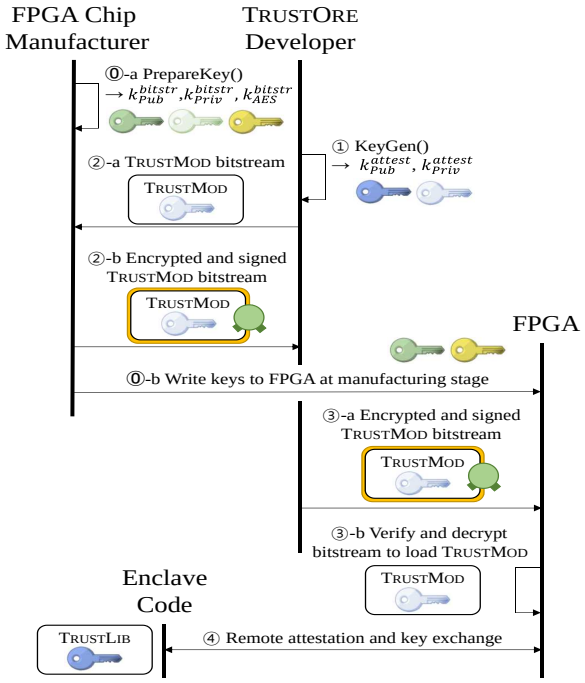


Figure 3: Key management for secure loading TrustMod

supported by TrustOre in order to efficiently connect the aforementioned components (§4.3).

4.1 Loading and Attesting TrustMod

One of the key components of TrustOre, i.e., TrustMod is implemented on an external FPGA device. However, TrustMod has to be loaded by the untrusted OS (using the device driver) and is therefore susceptible to the attacker's malicious behavior. Since external devices are not part of the protection scope of SGX, conventional remote attestation provided by Intel [48] is insufficient to attest the correct loading of TrustMod. Furthermore, state-of-the-art FPGA architectures (such as those provided by Intel and Xilinx) lack an attestation feature as mentioned in §2.2. As a result, we develop new techniques for extending trust to TrustMod in order to verify the correctness of loaded instance.

Loading using Device Driver. TrustOre provisions an existing security feature of the FPGA platform to securely load TrustMod to the FPGA. More specifically, TrustOre leverages an FPGA's secure boot process (see §2.2) to ensure the confidentiality and authenticity of TrustMod (i.e., a bitstream layout of TrustMod) to be loaded to the FPGA fabric. The confidentiality of TrustMod is important since it contains a pre-installed private key, which is used to attest TrustMod itself (as we explain shortly). On the other hand, authentication is important because an unauthenticated module can directly leak the trusted data of an enclave and should therefore not be utilized.

However, consider a naive secure boot of TrustMod, natively supported by existing FPGAs and mentioned in §2.2. In particular, a bitstream layout of TrustMod is initially prepared by the TrustOre developer (let the TrustOre developer be us in this paper). Then, this bitstream is delivered to the FPGA manufacturer, who returns an encrypted and signed bitstream pair which is loaded onto the FPGA device. Even though the FPGA chip manufacturer correctly signs and encrypts the bitstreams, this step only guarantees the confidentiality of TrustMod. The reason is that the attacker can just as easily procure signed bitstreams from the FPGA manufacturer and overwrite TrustMod at runtime. As a result, there is a need for a further authentication or attestation scheme to augment the protections of secure boot.

Attestation through TrustLib. TrustLib is responsible for attesting that the correct module of TrustMod is loaded onto the FPGA device that the enclave is copying its private data to. To achieve this, TrustOre places an RSA private key within the bitstream of TrustMod and uses it as a security anchor for runtime attestation of the FPGA.

To be more specific, we generate a unique attestation key pair (i.e., $k_{\text{Pub}}^{\text{attest}}$ and $k_{\text{Priv}}^{\text{attest}}$) beforehand. These pairs are generated from scratch each time a module is compiled ensuring that different devices have different attestation key pairs. Afterwards, TrustOre appends the attestation private key ($k_{\text{Priv}}^{\text{attest}}$) to TrustMod (②-a) and the attestation public key ($k_{\text{Pub}}^{\text{attest}}$) is provided to TrustLib (④) as shown in Figure 3. As TrustMod's compiled bitstream is handed over to the FPGA manufacturer for signing, $k_{\text{Priv}}^{\text{attest}}$ is stored encrypted within the bitstream (②-b, ②-c). One shortcoming of this scheme is that the FPGA chip manufacturer may perform circuit-level reversing to decipher the attestation key stored in TrustMod. However, in our threat model, the chip manufacturer is trusted and therefore this problem is out-of-scope. It can be solved by applying obfuscation techniques proposed previously [37, 46, 79].

As soon as TrustMod's bitstream is loaded to the FPGA (③), TrustLib attests it using its attestation private key (④), similar to other known attestation schemes [6, 16]. One notable difference is that TrustMod does not need to provide a runtime measurement, since the FPGA's secure boot already ensures the integrity of the loaded bitstream. Moreover, in order to ensure the freshness of the attestation message, TrustLib generates a random nonce for each attestation and sends it to TrustMod as illustrated in Figure 4 (❶). Then TrustMod creates a signature of the received nonce with $k_{\text{Priv}}^{\text{attest}}$, and returns the signature back to the enclave which can be verified by an enclave using $k_{\text{Pub}}^{\text{attest}}$ (❷). This prevents malicious entities from intercepting and/or replaying communication. Therefore, if an adversary rewrites the FPGA bitstream to impersonate TrustMod, this would be detected by our runtime attestation as long as $k_{\text{Priv}}^{\text{attest}}$ from the encrypted bitstream is secure.

In summary, the attestation process works as follows — TrustLib writes a random nonce to the fixed MMIO address assigned to the attestation transmission channel. TrustMod receives the request of the nonce and starts the signing operation. After signing using $k_{\text{Priv}}^{\text{attest}}$, TrustMod transmits the resultant back to the enclave. The enclave attests that it belongs to a correct module using $k_{\text{Pub}}^{\text{attest}}$ which should only work as long as the module has the correct $k_{\text{Priv}}^{\text{attest}}$.

4.2 Bootstrapping a Storage Model

After TrustMod has been loaded onto the FPGA, it begins the process of building a trusted storage within the internal device memory. In order to accommodate various enclave applications, TrustOre designs a flexible storage model which is compatible with various use-cases — including the general data structures (similar to ZeroTrace [60]), various files (similar to Obliviate [3]) and general memory blocks (similar to Obfuscuro [4]). Furthermore,

we support an access control mechanism with respect to an enclave application so that an instance of TrustMod can be concurrently used by multiple different enclave applications.

4.2.1 Memory Addressing & Tracking. TrustOre needs a scheme to address the device memory for further storage semantics and needs to track the device memory which will be allocated to different enclaves.

Device Memory Addressing. TrustMod accesses the underlying FPGA memory directly — it does not cache/fetch the memory content and does not utilize virtual addresses either. This is the notable difference from enclave applications, where its page tables, cache units, and branch prediction-units (BPUs) are accessible to untrusted components and therefore vulnerable to side-channels.

On-Chip Memory Allocation Table (OCMAT). TrustMod tracks all memory allocated to various enclaves using the On-Chip Memory Allocation Table (OCMAT). Using OCMAT, all storage operations (requested by TrustLib as we will describe later in §4.3.3) are carried out based on `ID` (i.e., a key in a key-value store). Each row of OCMAT records following information: 1) `ID`, an identifier specifying a data object. `ID` is internally maintained and assigned by TrustMod; 2) EID, an identifier specifying the owning enclave. TrustMod ensures an enclave cannot access data objects owned by other enclaves; 3) On-chip address, a physical FPGA on-chip address storing the data object corresponding to `ID`; Since TrustMod's storage is built upon FPGA on-chip memory (which has linear physical memory space), this address assists TrustMod to locate the data using `ID`; and 4) Size, the memory size allocated for each `ID`.

4.2.2 Request Servicing. Each enclave is assigned a unique identifier during initialization and their subsequent requests are placed in an internal FIFO queue by TrustMod, for servicing on a first-come-first-serve basis. In case the internal FIFO queue becomes full, TrustMod stalls the bus from accepting more requests by delaying the acknowledgment signals.

Access Control. TrustMod assigns a unique Enclave-ID (EID) to each connected enclave. EID is derived from the cryptographic shared key linked to the enclave and TrustMod stores this information internally. Afterwards, each subsequent request from the enclave (identified by encrypted communication on a fixed MMIO/DMA region), is attributed to this EID. The EID ensures that an enclave can only access its own memory and a request to a memory region it does not own will be promptly caught by TrustMod and dropped.

Servicing Memory Allocation/Deallocation Requests. When allocating a new data object, TrustMod first searches for an available `ID`, which will be dedicated for this object. Then, TrustMod searches for an available FPGA on-chip memory large enough to hold the requested object size. Finally, all this information is stored as a new row in OCMAT, while also appending the EID of the requesting enclave. When deallocating the object (specified by its `ID`), TrustMod removes the matched row in OCMAT only if TrustLib is the owner of the object. This effectively makes the object corresponding FPGA on-chip memory space available. It is worth noting that TrustMod denies all requests to map, unmap or access FPGA memory from all entities except the enclave which has established secure channel with TrustMod (see §4.3).

Servicing Read/Write Requests. To perform write/read request for an object, TrustMod first asserts whether the request has been initiated by the owner of the enclave (using the allocated enclave-ID) and then locates the FPGA on-chip memory address stored in OCMAT. Next, for a write operation, TrustMod writes the provided data to the FPGA on-chip memory sequentially. For a read TrustMod transfers data sequentially starting from the FPGA on-chip memory to TrustLib. We note that TrustMod does not have hardware cache units in accessing on-chip memory within the FPGA, so TrustOre is secure against cache side-channel attacks plaguing in the traditional computing architectures.

To mitigate any potential side-channel issues in accessing FPGA on-chip memory (e.g., DRAM-based side-channel attacks utilizing the time difference between row hits and row conflicts [54]), TrustMod ensures that the access time of FPGA on-chip memory is always the same from outside. To be specific, TrustMod ensures that the access time always takes the worst-case cycle (which is obtained through empirical experiments) by stalling the response. Although this worst-case based mitigation may not sound the best design choice from the performance aspect, this in fact only incurs 0.7% overhead according to our evaluation — the 64-Byte array access takes 9639.6*ns*, 67*ns* (0.7%) more than the one before the mitigation. This is because performance bottlenecks of TrustOre are mostly in either packet construction or IO data transmission, and thus always-worst DRAM access time does not contribute much on the overall performance overhead. In the next section (§4.3), we will explain how the above mentioned data is actually transmitted between TrustLib and TrustMod.

4.2.3 Memory Resource Management. TrustMod has to handle fragmentation of device memory which can occur since different enclaves will have different memory demands which have to be accommodated altogether.

On-Chip Memory Compaction. In order to avoid memory fragmentation and subsequent failure of memory allocation request, TrustMod runs a memory compaction algorithm [27] at regular intervals, which moves fragmented data objects to available space so as to cluster data objects in the on-chip memory space. More specifically, TrustMod compacts the on-chip memory if it has processed a specific number of deallocation requests (a configurable parameter), and accordingly updates OCMAT. Although the attacker could figure out through timing channels whether a memory compaction event is taking place or not, the event itself leaks no meaningful information to the attacker. It only shows the device does not have a physically contiguous chunk of memory as requested.

In the unlikely case that a new allocation fails even after performing memory compaction, TrustMod can swap-out data objects to secondary storage (i.e., system memory or disk). In order to avoid leaking information, TrustMod would require Oblivious RAM (ORAM) primitives to ensure secure loading/unloading of data. However, recent FPGA on-chip memory has reached several GBs and we believe that this will not occur frequently. The design of such a scheme is out-of-scope of this paper.

4.3 Connecting TrustLib and TrustMod

This subsection first describes how TrustOre creates a secure channel between its two major components (§4.3.1). Then, we elaborate of the I/O communication techniques supported (§4.3.2) and

lastly, we explain the programming interfaces supported for such communication (§4.3.3).

4.3.1 Secure Channel Establishment. To facilitate secure communication between TrustLib and TrustMod, we use the Diffie-Hellman [58] key exchange. This allows TrustLib and TrustMod to share a secret key, which can be used to encrypt and decrypt all later communication. We use `rdseed` instruction [32] within TrustLib to obtain true random numbers from CPU and include FPGA-based true random number generator proposed in [53] within TrustMod, to make extremely hard for the adversary to influence or guess random numbers on both CPU and FPGA. Alongwith the Diffie-Hellman, TrustMod also sends an attestation nonce (using $k^{\text{attest}}_{\text{Priv}}$, as per the scheme described in Figure 4) which is used by TrustLib to ensure that it is communicating with a correct instance of TrustMod.

While generating the secret values required for DH, TrustLib and TrustMod utilizes constant-time implementations to avoid leaking information. More specifically, TrustLib and TrustMod always run a multiplier by a fixed number of times, which may include dummy runs (i.e., multiplying with 1). To make even the execution time by adding dummies is a widely used concept in the work mitigating timing side-channel attack [2]. We note that the computation cycles of TrustOre's multiplier and divider depend on the width of data (not the value of it), and the width is constant.

After the above setup, only TrustLib and TrustMod can derive the shared random key, $g^{ab}\ mod\ p$, based on their own secret value, a and b, respectively. This shared random key is used to encrypt and decrypt all following communication. Similar to the attestation process, the key exchanging procedure can be started as needed during program execution through the fixed MMIO address assigned to the key exchanging transmission channel. We use AES-128 (with Galois/Counter Mode) as the authenticated encryption algorithm because its side-channel resistant implementation is easily accessible with the SGX library using AES-NI.

Freshness and Integrity. In AES-GCM, a 96-bit counter, *Initialization Vector* (IV) is initially set through our key exchange mechanism and synchronized between the enclave and TrustMod by monotonically increasing for every AES operation. AES-GCM also provides an authentication tag that is calculated by combining the hashes of the messages and the counter value. Since the counter value acts as a timestamp, data freshness as well as the integrity is guaranteed, which can prevent an adversary to corrupting or replaying captured data. Therefore, TrustOre avoids the need for Merkle Tree typically demanding considerable overhead to manage.

4.3.2 Supported I/O Communication Methods. TrustOre supports two communication channels between TrustLib and TrustMod, Memory Mapped IO (MMIO) and Direct Memory Access (DMA).

Memory Mapped IO (MMIO). An MMIO channel is established using the device driver. We assume this driver has been installed on the untrusted operating system, and it registers the FPGA to the Linux device file system (`devfs`) such that the access to the FPGA device memory can be performed through MMIO. Since all communication through this channel is encrypted, the device driver can at most initiate a *denial-of-service* attack (such as unmapping or modifying the PCIe configuration during runtime) which is also beyond the scope of SGX. To setup MMIO access for an enclave application, TrustLib asks the device driver to map TrustMod's memory to TrustLib's non-EPC memory region which can directly be accessed without an enclave exit.

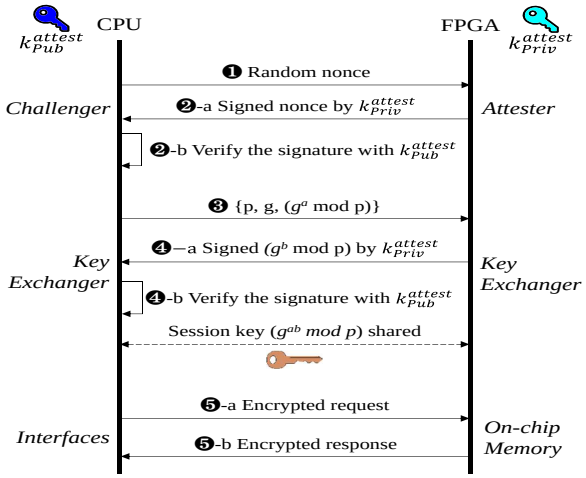


Figure 4: Secure channel establishment between CPU and FPGA

Direct Memory Access (DMA). In addition to MMIO, TrustOre also supports efficient transfer of large data using Direct Memory Access (DMA). The DMA mode of memory transfer involves two main components — (a) a MMIO region for passing requests, and (b) a DMA buffer to transmit actual memory related to the requests, both setup by the device driver. Requests, through MMIO request buffer, are passed in plaintext but are only used by the device to fetch the actual encrypted commands sent by TrustLib on the DMA buffer. Although there are setup costs in sending commands to the DMA controller and processing DMA interrupts, the DMA mode allows burst transmissions on the IO bus, supporting significantly improved throughput for a big chunk data packets (as we show in Figure 5a and Figure 5b).

Preventing Side-channels on MMIO/DMA Regions. Although each request/response transmitted through MMIO/DMA is encrypted, TrustOre could still suffer from side-channel inference as messages are passed. To protect from these side-channels, TrustOre ensures that all data accesses — attestation, key exchange, data packet transfer and DMA request are performed through the dedicated and fixed MMIO addresses. Furthermore, TrustOre reads/writes a data packet within an MMIO region at the granularity of 16 bytes that is synchronized to the block size of AES. To access a packet larger than 16 bytes, TrustOre repeatedly writes or reads the 16 bytes region while repeatedly concatenating each packet. Therefore, the enclave accesses the same MMIO location regardless of execution context and in the exact same way, thereby mitigating other side-channels. The concrete empirical analysis is provided in the Appendix C. TrustOre also restricts that the data block size is always the same at runtime (e.g., ORAM-based applications often used the static data block size by default, such as 4KB [3, 67]). As a result, TrustOre always reads or writes the same size of data irrespective of the request, and that size is determined during the initialization.

4.3.3 Supported Programming Interfaces. After setting up the communication channels between TrustLib and TrustMod, TrustOre is now ready to accept commands from the enclave program. In the

following we first explain the singular interface function TrustLib can invoke on TrustMod to perform various different operations. Then we describe how TrustOre defines the communication format of the interface function.

Storage Interfaces. TrustMod supports the following interfaces that TrustLib can request to use various storage operations. The interfaces are labeled as: `alloc`, `dealloc` and `access`.

```
OUT ID alloc();
OUT STATUS dealloc(IN ID id);
OUT STATUS access(IN type, IN id, IN SIZE_T offset,
    IN void* dat_in, OUT void* dat_out, IN SIZE_T size)
```

In the above prototype of the interface functions, `IN` and `OUT` are a type qualifier: `IN` denotes an argument that TrustLib is responsible to provide a value; and `OUT` denotes a return or an argument that TrustMod is responsible to return or fill up in response to the request. `ID` denotes an identifier of a data object, and `STATUS` indicates whether the invocation was successful or not.

`alloc` allocates a data object within TrustMod's on-chip memory, and returns an `ID` which uniquely identifies the corresponding data object. `dealloc` makes the previously allocated data object available within TrustMod's on-chip memory. It is worth noting here that `alloc` and `dealloc` functions are meant to be only called at the start and end of the program's execution respectively. Therefore, calls to these functions are not sensitive and do not need to be protected as such.

In `access`, TrustMod takes the offset as a parameter to determine the offset within the data object specified by `ID`. Then, depending on `type` (i.e., the first parameter which can be either `read` or `write`), TrustMod performs the followings. In the case of `read`, TrustMod copies the source data in a range starting from (base + offset) to (base + offset + size) to the destination buffer specified by the data argument, where base is the FPGA on-chip memory address specified in OCMAT. In the case of `write`, TrustMod updates the FPGA's internal memory with the data provided by TrustLib. It is worth noting here that `access` equalizes the lengths of data packets in either case of `write` or `read` by appending dummies. A single interface is utilized to serve both `read` and `write` requests to ensure that attackers cannot distinguish between these requests, in order to provide protections at-par with Oblivious RAM (ORAM).

Communication Packet Format. For each interface command invoked by TrustLib, a *request* packet is first sent from TrustLib to TrustMod, and then a *response* packet is returned. TrustOre illustrates the format of this packet in Figure 8 (in the Appendix A). Following the illustrated format, TrustOre constructs a packet according to `IN` and `OUT` type qualifiers — the request packet sets fields with `IN` while the response packet does with `OUT`.

We also tried to reduce the packet header information to support multiple enclaves. To be more specific, we do not include an ID of each enclave in the transaction header since this would require more bits in the header. Instead, as we assign a unique MMIO address for each TrustLib instance (i.e., each enclave instance), it is possible to easily support multiple TrustLib through transporting transactions to different address value.

5 SECURITY ANALYSIS

We summarize the security properties achieved by TrustOre in Table 1. In the following paragraphs, we provide details about critical side-channel attacks against our system and discuss the remaining attacks (mentioned in the table and previously tackled in the design section) in Appendix B.

Side-channel attacks against Enclave (TrustLib). Since enclave memory is prone to side-channel information leakages, TrustOre carefully provisions TrustLib to mitigate information leakage through all known memory channels including page table, cache and branch-prediction. In particular, TrustLib crafts all `read` and `write` requests in the same *branch-free* manner (using *oblivious* wrapper [56]) to ensure that the attacker cannot distinguish between a read and write request. It is worth mentioning here that `alloc` and `dealloc` are distinguishable but since they are meant to be called at the start and end of enclave execution respectively, they do not possess or leak sensitive information.

Side-channel attacks against FPGA (TrustMod). As previously mentioned, our storage model (on TrustMod) has no page tables or caches. Also, TrustMod guarantees that each read/write request returns the same amount of data (fixed during initialization) irrespective of what the program has requested. Therefore, TrustMod is free from memory-based side-channel attacks.

Furthermore, each request to TrustMod could leak some timing information (i.e., the processing time of a request is dependent to the processing time of previous requests). As we illustrated in §4.2, TrustOre ensures that regardless of TrustOre's operational contexts, an allocation request takes the worst-case time and read/write requests are constant time. Therefore, the only information leaked is the number of previous read/write operations performed by TrustMod. It is worth noting that this information is not secured by other schemes (e.g., ORAM-based storage) either and is out-of-scope of our work.

Lastly, TrustOre does not allow any other logic to run concurrently on the FPGA running TrustMod and therefore this is a non-viable attack channel.

6 IMPLEMENTATION

In the following, we explain how TrustOre implements its two main components, TrustMod and TrustLib.

TrustMod. We implemented all components of TrustMod in Verilog-HDL, and converted those to the bitstream by Xilinx's Vivado 2015.1 tool. Then we generated and managed secret keys for the secure loading and the attestation as explained in §4.1. TrustMod is loaded into the XC7Z045 FFG900-2 Xilinx Zynq-7000 series FPGA [81] which is connected to the host CPU via PCIe, and runs at 150MHz. FPGA chip XC7Z045 contains 2.2 `MB` on-chip block RAM and two 1 `GB` on-board DRAMs. We implemented TrustOre to utilize on-board DRAMs if the data size is over 2.2 `MB`. Our logic for TrustMod has 3,846 lines of code.

Our current implementation of TrustMod supports the following storage structures:

(1) **Trusted Data Array.** We implement a trusted data array (similar to ZeroTrace [60]) which is allocated using our `alloc` (mentioned in §4.3.3). The trusted data array has functionality of read/write on indices using our `access` calls and deallocation is performed using `dealloc` call.
(2) **Trusted File System.** We also develop a trusted file system (similar to Obliviate [3]). We extend our APIs (namely `trus_open`, `trus_close`, `trus_read`, `trus_write`, `trus_fsync`)

Table 1: Security analysis of TrustOre. More details are provided in Appendix B.

TrustOre Event/Component	Potential attacks	Defenses
Loading TrustMod to the FPGA	Reverse-engineering the bitstream; Unloading/exchanging the module	Secure boot and remote attestation (§4.1)
Communication between TrustLib and TrustMod	Eavesdropping/tampering; Side-channels for MMIO/DMA region	Cryptographic schemes (§4.3.1); Oblivious manner accesses (§4.2.2, §4.3.2)
TrustLib	Side-channels on EPC memory [41]	No input-specific data access; Branch-free implementation (§4.3.3)
TrustMod	Side-channels on FPGA memory; Concurrent attack logic on FPGA [84]	No virtual memory or caches; Constant-time access (§4.2.2); No other logic runs with TrustMod (§4.1)
FPGA device driver	Eavesdropping/tampering	Cryptographically-secured protocols (§4.3.1)

in order to support POSIX file system APIs. While some of these calls are trivial to understand, we mention a few that pose some trick to them.

`trus_open()`. Since we need to allocate a new region within the FPGA on-chip memory, `alloc` is called as part of `trus_open`. When `O_WRONLY` or `O_RDWR` is specified as an argument, we allocate a pre-defined size, similar to what Obliviate does. If the program attempts to write over the boundary of the file, we re-run the function with the `O_APPEND` flag and TrustMod doubles the current file size in the trusted storage.

`trus_close()`. The process of closing a file involves `dealloc` releasing the FPGA on-chip memory resources and writing-back all the stored data to the original file. This does not leak any information since it is a sequential write-back of the entire contents onto the main memory. Also, TrustOre uses a library for data sealing provided by SGX to ensure confidentiality of files.

TrustLib. We implemented this component using the Intel SGX SDK. Each API listed above is implemented as a wrapper function calling the primitive interfaces (i.e., `alloc`, `dealloc`, `access`) as subroutines. Two software modules, *Challenger* and *Key Exchanger* illustrated in Figure 4 for attesting TrustMod and exchanging the session key are also implemented together in TrustLib. The open source FPGA driver [10] was loaded into Linux kernel to register the FPGA through MMIO and enable the DMA transfer. We do not need to modify the driver but only adds 66 lines of code to the non-enclave code for delivering each pointer of MMIO and DMA buffer to the application enclave. In an enclave, TrustLib consists of 212 lines of code where the primitive interfaces introduce 155 lines of code among them. As a part of TrustMod, hardware modules relating to cryptographic operations (i.e., *Attester*, *Key Exchanger* and *AES*) consists of 1,412 lines of code.

7 PERFORMANCE EVALUATION

In order to show the superior performance achieved by TrustOre against existing ORAM-based schemes employed to defend against side-channel leakage, we perform a case-study comparing each of Obliviate [3], ZeroTrace [60] and Obfuscuro [4].

Experimental Setup. All our experiments were performed on Intel (R) Core (TM) i7-6700 CPU @ 3.40GHz (Skylake with 8 `MB` cache) with 32GB of RAM (128 `MB` for EPC) and running Ubuntu 16.04 with Linux 4.4.0.31 (64-bit). We used the official Intel SGX SDK and Intel SGX drivers for Linux for all experiments. To obtain experimental values, each of the same experiments was repeated at least 100 times. In addition, power saving mode was turned off and CPU frequency was set to the maximum value in Linux in order to minimize variation between experiments. For all the time measurements, `clock()` function was used in common.

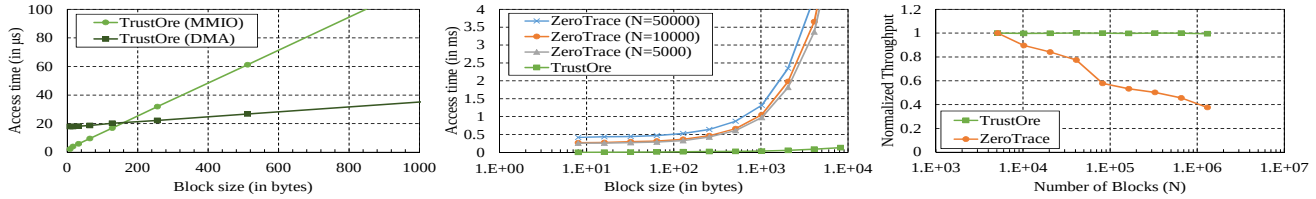
7.1 Experimental Results

In this subsection, we show the performance achieved by individual storage structures supported by TrustOre.

Performance of Data Array. We compare the latency and throughput achieved by TrustOre while accessing a data array as compared to ZeroTrace. Figure 5b shows the latency when accessing an index within an oblivious array of various sizes using ZeroTrace's ORAM-based access and TrustOre's FPGA-based access. Furthermore, we provide results by varying the size of N, i.e., 5000, 10000, and 50000. The value of N corresponds to the size of the oblivious array for ZeroTrace. Since ZeroTrace uses ORAM operations, it has to create a fixed size tree (and a fixed size array as a result). TrustOre does not have such a restriction as long as the size does not exceed the total memory available on the FPGA device. Based on the results, TrustOre accesses data 49× faster than ZeroTrace on average across different block sizes. In the case of 64B block, which is a general cache-line size and thus referenced as a default configuration when evaluating ORAM-based systems' performance [21, 22, 57], TrustOre shows 37× faster access speed. However, as the block size grows (greater than 10^3), the amount of memory accessed for an ORAM operation exponentially increases, and as a result the performance degrades considerably. For example, for block size of 10^5, the difference between ZeroTrace and TrustOre is almost 75×.

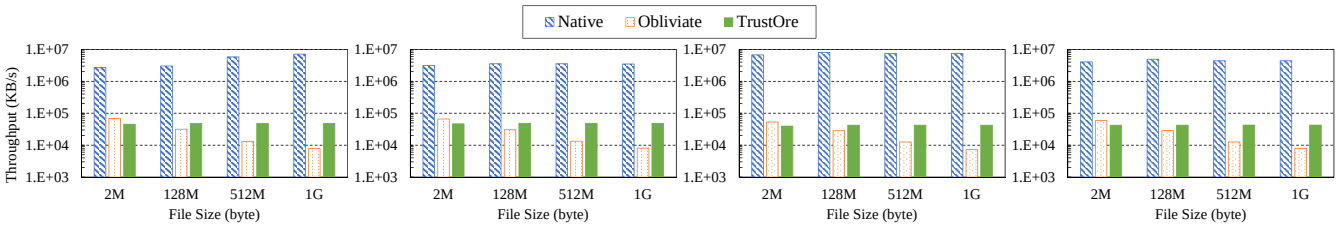
To better compare the scalability difference between TrustOre and ZeroTrace, we measure the throughput with respect to the number of blocks (N) as shown in Figure 5c. In the figure, the throughput of ZeroTrace decreases as N increases, while TrustOre shows constant throughput. This is due to that the ORAM tree that ZeroTrace has to access increases considerably as the value of N increases. It is worth recalling that ORAM trees grow exponentially as the blocks are increased in order to maintain the security properties. For further details on this matter, we refer to the original works related to ORAM [68, 78].

To measure the performance of TrustOre compared to the state-of-the-art hardware-based ORAMs [21, 22], we compared the access time for 64B block with TrustOre's one and the performance number reported in [21, 22]. Compared to the native access time for 64B block (i.e., without any protection mechanism), TrustOre decreases the throughput by 102× while [21] and [22] decrease 22× and 4×, respectively. While this performance result may not be in favor of TrustOre, we note that TrustOre does not require any hardware/architectural changes which would be challenging to deploy for practical use-cases. We further note that ZeroTrace decreases by 3774×, so we argue that TrustOre provides the comparable performance leveraging the readily available hybrid CPU-FPGA architecture.



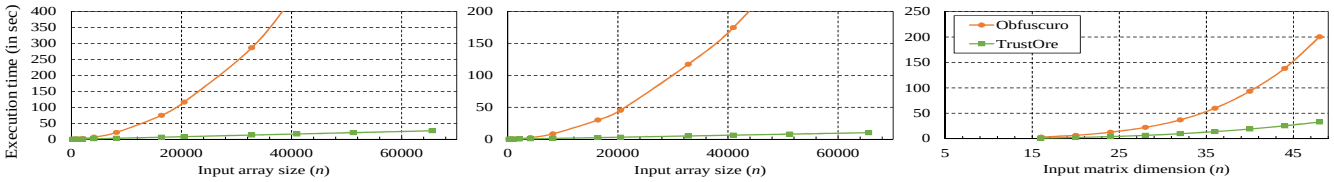
(a) Performance of MMIO and DMA mode **(b) Performance on various block sizes** **(c) Throughput on various block numbers (N)**

Figure 5: Performance comparison of oblivious data array access



(a) Sequential reads **(b) Sequential writes** **(c) Random reads** **(d) Random writes**

Figure 6: Performance comparison of oblivious file access



(a) findmax **(b) sum** **(c) matmul**

Figure 7: Performance comparison of oblivious execution with various input data size

Performance of File System. We evaluate the performance of TrustOre's file system as compared to the native file system (accessing the disk) and Obliviate [3]. Figure 6 shows the results of sequential and random read/write (not including open/close) comparing the three file systems using Iozone [51], a widely used file system benchmark. TrustOre performs 102× slower on average than the native file system, while Obliviate 221× slower than the native one. When the test file size is 2 MB, the performance of Obliviate and TrustOre is comparable. This is because the TrustOre's IO transmission overhead through PCIe is still higher than the operation of the Obliviate's ORAM controller which deals with the 2 MB data set. However, TrustOre maintains the constant throughput, 44 MB/s on average, regardless of the file size, while Obliviate exhibits severe throughput degradation as the file size increased. As the size of the data set grows, the memory accessed by Obliviate's ORAM controller increases significantly due to the increased size of the ORAM tree. On the other hand, TrustOre performs only a number of operations depending on the requested data size. In the test for a 1GB file, TrustOre is approximately 10× faster than Obliviate. We expect this trend to explode further as the file size increases but due to implementation limitations, we were unable to get results from larger than 1GB files for Obliviate.

Other Systems (Obfuscuro). Our storage structures (e.g., trusted data array) can also be indirectly applied to other systems. For example, Obfuscuro [4] proposes an obfuscation scheme for Intel SGX while relying on trusted in-enclave storage using ORAM. We compared our trusted storage against Obfuscuro's ORAM-based storage by reusing their framework but changing all ORAM access operations in their source code (publicly available [1]), with our trusted data array implementation.

Table 2: Execution time comparison of nbench between ZeroTrace and TrustOre

nbench	Native SGX (us)	ZeroTrace -based (sec)	TrustOre -based (sec)	Speed -up
Num sort	548.186	229.947	1.777	129.38
String sort	18670.301	17526.434	140.267	124.95
Bit operation	0.001	2434.542	12.666	192.21
Fp. emulation	1556.493	6.795	0.091	74.30
Fourier	7.543	0.065	0.002	31.43
Assignment	297.796	121.658	1.071	113.59
Idea	56.430	800.842	4.501	177.94
Huffman	149.011	109.081	0.568	191.92
Neural net	7954.184	10839.355	94.621	114.56
Lu decomp.	280.143	430.785	2.215	194.53
GeoMean				120.16

We tested three programs which perform matrix multiplication (matmul), integer summation (sum), and finding the max in an array (findmax). These programs were provided to us with the source code of Obfuscuro. To test the scalability of TrustOre, we vary the input data size n for each of the aforementioned program. As shown in Figure 7, the execution time of Obfuscuro exponentially increases while that of TrustOre increases smoothly. For example, TrustOre executes `findmax` 2.88× faster when the input size is 2K and 43× faster when the input size is increased to 64K. It demonstrates that TrustOre solves the scalability problem of Obfuscuro which has to rely on costly ORAM operations.

7.2 Other Benchmarks

Nbench. We evaluate the performance of TrustOre for real world workload by using nbench [47]. To run nbench to SGX, each test

program of nbench is manually modified and split into two parts: non-enclave and enclave part. Non-enclave part generates input data arrays, copies them to enclave arrays and then enclave runs the program without another entry/exit until finishing the program. Each memory store/load operation in enclave is modified to perform oblivious access using ZeroTrace's API calls and TrustOre's `access` API. This modification is straightforward and is intended to see how slow the performance of oblivious access would be on nbench. For the fair comparison, we set N (number of blocks) for ZeroTrace to the exact number of blocks required by each individual benchmark within the test-suite. N varies from 3000 to 20000 in our experiment. And the block size is set to the same as the size of each data object (mostly integers and floats). As shown in Table 2, although TrustOre-based shows orders of magnitude slower than the native execution because of IO delay to access FPGA per every memory accessing operation, TrustOre is 120× faster on average than ZeroTrace. As expected, TrustOre shows better performance in programs where data access occurs heavily (`Huffman`, `Lu decomp.`) rather than compute-intensive ones (`Fourier`).

MMIO Versus DMA. TrustOre supports both communication standards for CPU-FPGA communication, i.e., MMIO and DMA. In order to gauge the performance difference between these standards, we measure the latency when transferring data packets of various sizes using each of the aforementioned channels. As shown in Figure 5a, TrustOre transfers data of size greater than 128-bytes faster using DMA than MMIO. There are two main reasons why DMA is slower when the data sizes are smaller: (a) requiring to transfer the control command to the DMA controller within TrustMod in order to initiate DMA transfer exceeds the performance gain and (b) waiting the DMA interrupt passed back from OS also takes some time. So, to maximize the performance of TrustOre, TrustLib seamlessly chooses DMA transfer mode for data bigger than 128-bytes and MMIO otherwise.

End-to-End Key-Value Store. To evaluate TrustOre on the real system requring the side-channel protection, we used TrustOre for an end-to-end key-value store application, *ShieldStore* [39]. ShieldStore is a state-of-the-art in-memory key-value store designed for SGX. ShieldStore addresses the performance issues due to the SGX memory limitation through storing the data in unprotected memory, where each key-value pair is individually encrypted and integrity-protected using its secure component running inside an enclave. However, ShieldStore is vulnerable to side-channel attacks, since ShieldStore does not hide the address being accessed. Thus, its key-value store is insecure against access pattern based side-channel attacks. TrustOre offers a side-channel resistant storage for ShieldStore to address those attacks.

We implemented TrustOre-based ShieldStore, which modified ShieldStore to store the table (the hash and its corresponding data) on our trusted data array (see §6). To better understand the performance impact of TrustOre-based ShieldStore, we also implemented ZeroTrace-based ShieldStore, which provides side-channel resistant storage with an ORAM mechanism. We compare the throughput of key-value store operations (i.e., SET/GET) between ZeroTrace-based ShieldStore and TrustOre-based ShieldStore. 500 random operations were tested when both the sizes of key and value are 16-byte. As shown in Table 3, TrustOre-based ShieldStore shows 188× higher throughput against ZeroTrace-based one on average. Compared to the baseline ShieldStore, TrustOre-based ShieldStore showed 111× lower throughput, while ZeroTrace-based one showed $20.8K\times$ lower throughput. We believe such an outstanding performance improvement of TrustOre-based ShieldStore (compared to ZeroTrace-based one) is due to the fact that TrustOre only requires the same number of memory accesses comparing to the native ShieldStore, while ZeroTrace needs much more accesses to perform an ORAM mechanism.

Table 3: Throughput comparison of ShieldStore between ZeroTrace and TrustOre

	ShieldStore	ZeroTrace-based	TrustOre-based
Throughput (Ops/s)	35465.484	1.702	320.054
Slow-down	-	$20.8K\times$	111×

8 DISCUSSION

This subsection discusses more subtle, sophisticated attacks (which are not part of our threat model) against TrustOre: cold-boot attacks and part of side-channel attacks. We also discuss the security concerns about involving the FPGA manufacturer as our new trusted party.

Cold-Boot Attacks. Cold-boot attacks attempt to read un-encrypted data stored in memory by physically detaching it from the board. Similar attacks may be launched on TrustOre as well — desoldering the FPGA chip at runtime and dumping the data from the on-chip memory. However, compared to the previously known cold-boot attacks [28, 83], it would be much more challenging to dump FPGA on-chip memory. FPGA on-chip DRAM is 3D stacked with the FPGA fabric, and IO port for accessing the stacked DRAM is not directly exposed unlike a traditional DRAM [35].

Nevertheless, in order to completely thwart cold-boot attacks, TrustOre can be extended with a memory encryption mechanism similar to Intel SGX's Memory Encryption Engine [26]. We note that as TrustOre's encryption logic would be written in bitstream and eventually run at the hardware level and therefore, it should incur less overhead compared to the purely software-based ones [14, 29].

Side-Channel Attacks TrustMod introduces new operational semantics to the system, including allocation, read, write, and deallocation requests, each of which can be a source of side-channel attacks. First, these requests may leverage resources shared with other bitstream instances, as an OS may instruct the FPGA to run other bitstream than TrustMod. To mitigate these side-channels, TrustOre always wipes out all data and resources before being unloaded. Moreover, since the OS has the full control over the FPGA and thus can suddenly reprogram the FPGA before wiping all data and resources, TrustOre disables the processor's connection to the configuration ports of the FPGA during the secure boot as illustrated in [17]. Second, each request may share resources with other requests (i.e., the processing time of a request is dependent to the processing time of previous requests). As we illustrated in §4.2, TrustOre ensures that regardless of TrustOre's operational contexts, an allocation request takes the worst-case time and read/write requests take the constant time, effectively avoiding potential side-channel attacks.

The Trusted FPGA Manufacturer We acknowledge that imposing an additional trusted party (i.e., an FPGA manufacturer) may weaken the original threat model of SGX. We argue that, however,

if the FPGA manufacturer and the CPU manufacturer are the same (i.e., Intel), this would not raise much trustworthy issues. In fact, given the market dominance on CPU and FPGA, we believe both would be (or already are) manufactured by the same party, Intel. Intel is aggressively pushing towards an CPU-FPGA hybrid architecture in response to the data-intensive computing trends (e.g., Xeon-FPGA chip [33], FPGA acceleration card [34]). In this case, Intel would be manufacturing both CPU and FPGA, so the key can be installed in the FPGA by Intel (just as the SGX key in the CPU) in order to utilize the FPGA as a security extension of SGX. In this deployment scenario, it can be assumed that the server for bitstream encryption/signing is also securely integrated with the server for the SGX attestation.

9 RELATED WORK

Systems based on SGX. Haven [9] is the pioneering SGX work which developed a windows-based library operating system (LibOS) to enable easy porting of legacy applications on Intel SGX. Its counterpart, Graphene [70, 71] proposed a Linux-compatible LibOS. OpenSGX [36] provides an opensource framework for SGX development. Ryoan [30] implements a distributed sandbox, Scone [7] proposes secure containers, and SGX-Shield [62] enables ASLR on SGX enclave. Graviton [75] proposes hardware modifications to enable trusted execution on GPUs in tandem with Intel SGX. All aforementioned systems are not concerned with side-channel limitations of Intel SGX and would greatly benefit from the side-channel protections afforded by TrustOre.

Side-channel Attacks on SGX. There are four main types of side-channel attacks discovered against Intel SGX, i.e., IAGO [12], Page table [72, 82], cache [11, 24, 61] and branch prediction [19, 42]. Leaky Cauldron [77] provides an overview of memory-based side-channel attacks possible within Intel SGX. TrustOre involves a trusted FPGA in order to protect trusted data from all aforementioned side-channels.

Side-channel Mitigations for SGX. The existing side-channel defenses for Intel SGX can be divided into cryptographic [3, 4, 60] and non-cryptographic [25, 62, 64, 65] defenses. The existing cryptographic defense schemes utilize ORAM to protect the application from access pattern-based attacks. Although ORAM offers cryptographic security, it is prohibitively slow (as we experimentally show in §7) since it involves a degree of higher memory interactions than native execution. On the other hand, existing non-cryptographic defenses offer comparatively lower overheads but protect against individual side-channels and cannot be leveraged to protect against all side-channels which leak access patterns. For example, T-SGX [64] protects against page-fault attacks but cannot protect against page table and cache attacks. Similarly, Cloak [25] can protect against page-fault and cache attacks but cannot protect against page table attacks. SGX-Shield [62] can only provide probabilistic defense of side-channels through memory randomization. Compared to all existing solutions, TrustOre offers superior performance as well as proven protection against all access pattern leakage.

Other Side-channel Mitigation Schemes. There are various other software [15, 38, 56, 66, 85] and hardware [20, 43, 45] schemes to mitigate side-channels in non-SGX environments. Some of the software schemes [15, 38, 85] are not applicable within Intel SGX since they require OS support. From amongst the applicable schemes, Raccoon [56] is the most notable since it provides protections against all digital side-channels. However, Raccoon only secures annotated part of a program's data and uses oblivious copy (and ORAM) to securely access the data at a high cost to performance (i.e., 21×). Hardware techniques, such as HOP [50] and Phantom [45], also utilize ORAM to protect the data which as mentioned before is very slow as compared to TrustOre. Also, these schemes are prototyped using custom (RISC-V) processors and are therefore less deployment-friendly than TrustOre.

Secure Memory Architectures. The most related works are [2, 8], which leverage computation capabilities (i.e. encryption and decryption) of 3D-stacked memory to achieve ORAM-equivalent security guarantees. In particular, it encrypts all memory bus traffic and thus adversaries launching man-in-the-middle attacks cannot see the intention behind memory bus uses. However, these approaches are limited for the following reasons: essentially built based on new memory architecture, smart memory, which comes at high cost and provides the limited capacity. Secure DIMM [63] employs an ASIC buffer chip to offload ORAM functionality such that memory bandwidth can be reduced. Compared to TrustOre, all of these works rely either on new memory structure or a custom hardware, limiting their use-cases in real-world. TrustOre is implemented on hybrid CPU-FPGA architecture and Intel SGX, readily available to use today.

10 CONCLUSION

This paper proposed TrustOre, a system built for stopping side-channel attacks against Intel SGX. It utilizes an external device, an FPGA, to implement a trusted storage service for SGX applications. Since the FPGA is running in an isolated environment having its own dedicated memory-related units, TrustOre avoids memory-based side-channels by design. Moreover, unlike ORAM-based side-channel protection solutions, TrustOre scales well as the data size increases, demonstrating its strong practical prospects for real-world workloads. We emphasize that as TrustOre does not impose any architectural change but can be deployed as a simple plug-in to SGX machine's PCIe slot, it is readily used to thwart side-channel attacks in Intel SGX, arguably one of the most cryptic and critical security holes today.

ACKNOWLEDGMENTS

This work was partly supported by the BK21 Plus program of the Creative Research Engineer Development for IT, Seoul National University in 2020 and the EDA tool from the IC Design Education Center and the National Research Foundation of Korea grant funded by the Korea government(MSIT) (NRF-2017R1A2A1A17069478, NRF-2019R1C1C1006095) and Institute of Information & communications Technology Planning & Evaluation grant funded by MSIT (No.2018-0-00230, Development on Autonomous Trust Enhancement Technology of IoT Device and Study on Adaptive IoT Security Open Architecture based on Global Standardization [TrusThingz Project] and No.2017-0-00213, Development of Cyber Self Mutation Technologies for Proactive Cyber Defense).

REFERENCES

[1] Github - adilahmad17/obfuscuro: Commodity obfuscation for intel sgx, 2019. URL https://github.com/adilahmad17/Obfuscuro.

[2] S. Aga and S. Narayanasamy. Invisimem: Smart memory defenses for memory bus side channel. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, New York, NY, June 2017.

[3] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. Obliviate: A data oblivious file system for intel sgx. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[4] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[5] Amazon. Aws ec2 fpga development kit. https://github.com/aws/aws-fpga. [Online; Accessed 22. August 2019], 2018.

[6] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 14th Hardware and Architectural Support for Security and Privacy (HASP)*, Tel-Aviv, Israel, June 2013.

[7] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, et al. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[8] A. Awad, Y. Wang, D. Shands, and Y. Solihin. Obfusmem: A low-overhead access obfuscation for trusted memories. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, New York, NY, June 2017.

[9] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[10] K. Bhimani. Zc706 pcie driver. https://github.com/codelec/zc706_pcie. [Online; Accessed 22. August 2019], 2017.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017.

[12] S. Checkoway and H. Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[13] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018. URL http://arxiv.org/abs/1802.09085.

[14] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, Mar. 2008.

[15] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2009.

[16] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[17] M. Coughlin, A. Ismail, and E. Keller. Apps with hardware: Enabling run-time architectural customization in smart phones. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[18] M. Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. In *Federal Information Processing Standards (FIPS) Special Publications (SP)*, Nov 2007.

[19] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, Mar. 2018.

[20] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12, 2012.

[21] C. W. Fletcher, L. Ren, A. Kwon, M. v. Dijk, E. Stefanov, D. Serpanos, and S. Devadas. A low-latency, low-area hardware oblivious ram controller. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 215–222, May 2015. doi: 10.1109/FCCM.2015.58.

[22] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2015.

[23] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[24] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel sgx. In *EUROSEC*, pages 2–1, 2017.

[25] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug. 2017.

[26] S. Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.

[27] B. K. Haddon and W. M. Waite. A Compaction Procedure for Variable-Length Storage Elements. *The Computer Journal*, 10(2):162–165, 08 1967. ISSN 0010-4620. doi: 10.1093/comjnl/10.2.162. URL https://doi.org/10.1093/comjnl/10.2.162.

[28] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009. ISSN 0001-0782. doi: 10.1145/1506409.1506429. URL http://doi.acm.org/10.1145/1506409.1506429.

[29] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[30] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[31] Intel. An introduction to the intel(r) quickpath interconnect. https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html. [Online; Accessed 22. August 2019], 2009.

[32] Intel. Intel 64 and ia-32 architectures software developer's manual. https://www.intel.co.kr/content/www/kr/ko/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html. [Online; Accessed 18. August 2020], 2016.

[33] Intel. Intel(r) xeon(r) gold 6138 processor, 2018. URL https://en.wikichip.org/wiki/intel/xeon_gold/6138p.

[34] Intel. Intel(r) programmable acceleration card (pac) with intel(r) arria(r) 10 gx fpga datasheet, 2018. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ds/ds-pac-a10.pdf.

[35] Intel. Intel stratix 10 mx (dram system-in-package) device overview. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-mx-overview.pdf. [Online; Accessed 22. August 2019], 2019.

[36] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. OpenSGX: An Open Platform for SGX Research. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[37] R. Karam, T. Hoque, S. Ray, M. Tehranipoor, and S. Bhunia. Robust bitstream protection in fpga-based systems through low-overhead obfuscation. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Nov 2016. doi: 10.1109/ReConFig.2016.7857187.

[38] T. Kim, M. Peinado, and G. Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[39] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 14:1–14:15, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6281-8. doi: 10.1145/3302424.3303951. URL http://doi.acm.org/10.1145/3302424.3303951.

[40] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, Apr 2011. ISSN 2190-8516. doi: 10.1007/s13389-011-0006-y. URL https://doi.org/10.1007/s13389-011-0006-y.

[41] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa. An off-chip attack on hardware enclaves via the memory bus. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[42] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug. 2017.

[43] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGARCH Computer Architecture News*, 43(1):87–101, 2015.

[44] A. Ltd. mbed tls. https://tls.mbed.org. [Online; Accessed 22. August 2019], 2015.

[45] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[46] H. Mardani Kamali, K. Zamiri Azar, K. Gaj, H. Homayoun, and A. Sasan. Lutlock: A novel lut-based logic obfuscation for fpga-bitstream and asic-hardware protection. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 405–410, July 2018. doi: 10.1109/ISVLSI.2018.00080.

[47] U. F. Mayer. Linux/unix nbench. https://www.math.utah.edu/~mayer/linux/bmark.html. [Online; Accessed 22. August 2019], 2011.

[48] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 14th Hardware and Architectural Support for*

Security and Privacy (HASP), Tel-Aviv, Israel, June 2013.

[49] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, B. Sunar, F. Piessens, and Y. Yarom. Fallout: Reading kernel writes from user space. *CoRR*, abs/1905.12701, 2019. URL http://arxiv.org/abs/1905.12701.

[50] K. Nayak, C. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shi, and V. Goyal. Hop: Hardware makes obfuscation practical. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[51] W. D. Norcott and D. Capps. Iozone filesystem benchmark. http://www.iozone.org. [Online; Accessed 22. August 2019], 2003.

[52] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *CryptographersâĂŹ Track at the RSA Conference*, pages 1–20. Springer, 2006.

[53] A. Peetermans, V. Rozic, and I. Verbauwhede. A highly-portable true random number generator based on coherent sampling. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 218–224, 2019. URL https://github.com/KULeuven-COSIC/COSO-TRNG.

[54] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[55] J.-J. Quisquater and D. Samyde. Side Channel Cryptanalysis. In *Invited talk in SEcuritÃĺ de la Communication sur Internet (SECI 02). Tunis, Tunisia*, 9 2002. Invited talk.

[56] A. Rane, C. Lin, and M. Tiwari. Raccoon: closing digital side-channels through obfuscated execution. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[57] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious ram. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[58] E. Rescorla. Diffie-hellman key agreement method. https://tools.ietf.org/html/rfc2631. [Online; Accessed 22. August 2019], 1999.

[59] L. Sanders. Secure boot of zynq-7000 all programmable soc. https://www.xilinx.com/support/documentation/application_notes/xapp1175_zynq_secure_boot.pdf. [Online; Accessed 22. August 2019], 2015.

[60] S. Sasy, S. Gorbunov, and C. W. Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[61] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[62] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[63] A. Shafiee, R. Balasubramonian, M. Tiwari, and F. Li. Secure dimm: Moving oram primitives closer to memory. In *Proceedings of the 24th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Vienna, Austria, Feb. 2018.

[64] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[65] S. Shinde, Z. Chua, V. Narayanan, and P. Saxena. Preventing your faults from telling your secrets. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–June 2016.

[66] R. Sinha, S. Rajamani, and S. A. Seshia. A compiler and verifier for page access oblivious computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017. ACM, 2017.

[67] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[68] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[69] R. K. S. A. Ting Lu. Secure device manager for intel(r) stratix(r) 10 devices provides fpga and soc security. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01252-secure-device-manager-for-fpga-soc-security.pdf. [Online; Accessed 22. August 2019], 2018.

[70] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[71] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[72] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug. 2017.

[73] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[74] S. van Schaik, A. Milburn, S. ÃŰsterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[75] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on gpus. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, Nov. 2016.

[76] R. Wang, Y. Zhang, and J. Yang. D-oram: Path-oram delegation for low execution interference on cloud servers with untrusted memory. In *Proceedings of the 24th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Vienna, Austria, Feb. 2018.

[77] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2016.

[78] X. Wang, H. Chan, and E. Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[79] J. B. Wendt and M. Potkonjak. Hardware obfuscation using puf-based logic. In *Proceedings of the 33rd IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, USA, Nov. 2014.

[80] J. Winkles. Pci express(r) basics. http://www.cs.uml.edu/~bill/cs520/slides_15B_PCI_Express.pdf. [Online; Accessed 22. August 2019], 2006.

[81] Xilinx. Zc706 evaluation board for the zynq-7000 xc7z045 soc user guide. https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf. [Online; Accessed 22. August 2019], 2018.

[82] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[83] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX, Feb. 2017.

[84] M. Zhao and G. E. Suh. Fpga-based remote power side-channel attacks. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[85] Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

Appendix A DATA FORMAT OF TRANSACTION PACKET

One notable feature of TrustOre's packet format is that it provides two format options for the request packet, Req^{T} and Req^{G}. Req^{T} is for *time efficient* communication by decreasing the header overhead, particularly when $N \leq 8$ (N is the size of data). Req^{G} is for supporting other *general* cases where N is larger. Figure 8 illustrates two packet formats, Req^{T} and Req^{G} supported by TrustOre. Since every encryption/decryption is executed in a unit of AES block, the number of the blocks directly impacts the performance. Req^{T} is for reducing the number of the AES blocks when the data size N is relatively small.

In particular, if N is 8B, the total request packet size is to be 128-bit in the Req^{T} format, which is the same as an AES block size and thus an entire packet can be sent within a single AES block. However, if data of 8-byte size is transmitted in Req^{G} format, the total packet size becomes 208-bit and two AES blocks (the remaining 48-bits are filled with dummy) have to be transmitted. Therefore, the access latency is increased each time as compared with Req^{T}.

Appendix B FURTHER SECURITY ANALYSIS

We discuss the security properties of TrustMod components/events not thoroughly discussed previously.

Loading TrustMod As mentioned in §4.1, TrustOre uses existing hardware-based security features implanted in commodity FPGA platforms in order to guarantee the correct loading of TrustMod. To reiterate, FPGAs support a secure boot mechanism which ensures confidentiality and verifiability. Using the secure boot mechanism, TrustOre can load an encrypted module onto the FPGA. Furthermore, it can verify the module's correctness through a returned bitstream from the FPGA corresponding to the memory layout of the module. The bitstream layout of TrustMod remains consistent regardless of the underlying FPGA and can therefore be easily verified by the enclave. If the verification is successful, the enclave knows that the correct module was securely loaded on the FPGA via the untrusted system. In case the verification fails, the enclave finds out that some entity on the system is behaving maliciously and aborts. If the adversary behaves maliciously, TrustOre has no other choice but to stop executing since it constitutes a *denial-of-service* violation, neither guaranteed by Intel SGX or TrustOre.

Communication channel. TrustOre creates a secure channel between TrustMod (loaded onto the FPGA) and TrustLib (located inside the enclave) using Diffie-Hellman secret key exchange protocol [58]. TrustOre establishes a secret key between the two trusted parties which is bolstered using side-channel oblivious AES-GCM protocol [18] provided by Intel SGX SDK. All ensuing communication (i.e., allocating/deallocating or reading/writing memory) is encrypted using the shared key as it passes through the untrusted memory. All requests passed from TrustLib to TrustMod are encrypted and written directly by the enclave onto the MMIO regions or DMA buffers. At this point, OS can launch only a denial-of-service attack by remapping MMIO and DMA regions. Furthermore, each request is carefully crafted to be indistinguishable from each other (as mentioned in §4.3). To elaborate, TrustOre ensures that the size of each transaction is set at the start of the program and never changes at runtime. In the same way, all responses have a consistent format and size and are encrypted. Finally, all MMIO or DMA regions are established during initialization and remain consistent throughout the program's execution. Therefore, an attacker can only figure out the number and time of requests/responses which are not the protection scope of TrustOre.

FPGA Device Driver. The device driver is responsible for the setup of the MMIO/DMA channel between an application and the FPGA device. We conceive that the untrusted device driver can act maliciously in the following ways: (a) Choose not to setup MMIO/DMA regions, (b) Stop handling device and CPU interrupts, or (c) Modify messages as they are being transmitted. Amongst the following, (a) and (b) are *denial-of-service* violations which are out of the scope of this paper. Furthermore, (c) will easily be caught by TrustLib and TrustMod since all messages are encrypted with a shared secret key unknown to the attacker. Therefore, TrustOre is secure from tampering by the FPGA device driver.

Table 4: Pearson correlation of the accessed memory addresses

	Native SGX (1st input)	Native SGX (2nd input)	TrustOre (1st input)	TrustOre (2nd input)
Native SGX (1st input)	1	0.383801	0.035149	0.035149
Native SGX (2nd input)	0.383801	1	0.034864	0.034864
TrustOre (1st input)	0.035149	0.034864	1	1
TrustOre (2nd input)	0.035149	0.034864	1	1

Table 5: Memory read and write count comparison of nbench (Num sort) between native SGX and TrustOre

	Read Count	Write Count	Total Count
Native SGX (1st input)	616,106	212,446	828,552
Native SGX (2nd input)	616,564	212,626	829,190
TrustOre (1st input)	1,697,664	1,697,664	3,395,328
TrustOre (2nd input)	1,698,940	1,698,940	3,397,880

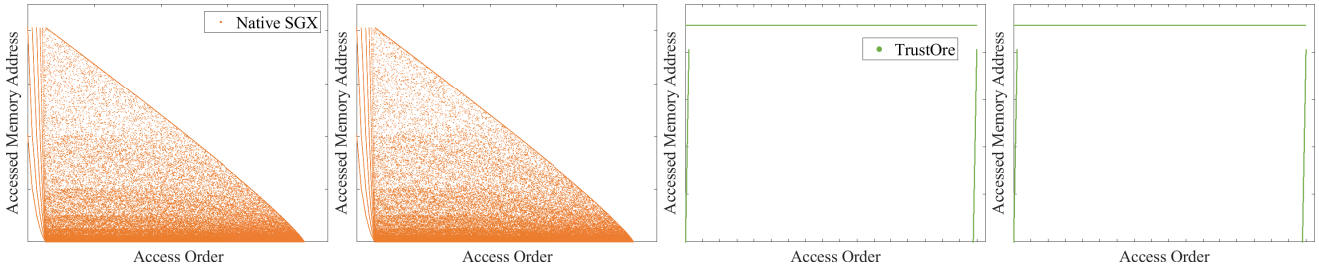
Appendix C MEMORY ACCESS PATTERN ANALYSIS

We provide the results of our empirical study on how TrustOre protects memory access patterns. To present the realistic comparison, we chose one of the nbench program, Num sort. Num sort performs the heap sort algorithm for the given data array. We run Num sort on the different two input data sets which are generated from the different random seed. To compare memory traces between the native SGX and TrustOre, we capture all the write and read accesses for the data array stored in the enclave. We also gather the traces of TrustLib including the mapped MMIO region to communicate with the FPGA.

Figure 9 shows the comparison of the captured memory traces between the tests. Using the accumulated data, we calculate the Pearson correlation value (Table 4) between each test to quantify how similar (close to 1) and different (close to 0) the traces are. As shown in Figure 9a and Figure 9b, a particular trace pattern is observed in the memory trace gathered from the program running in the native SGX. Furthermore, the traces between when inputting the different data is distinguishable, given that the Pearson correlation value is 0.383801 calculated in Table 4. The change in trace pattern according to the input data is likely to result in the leakage of data information (e.g., ordering) by differential attack. On

Direction	Bit position [start:size]		Field name	Description
	Req^T	Req^G		
Enclave to FPGA	[0:2]		OP_TYPE	Indicating the requested operation type 0x0: write request 0x1: read request 0x2: allocation request 0x3: deallocation request
	[2:10]		ID	Indicating the ID of the data that is targeted of the request (supporting range: 0~1023)
	[12:26]	[12:64]	SIZE	Indicating the requested data size in bytes (= N) 26-bit for the option Req^T (up to 64MB) 64-bit for the option Req^G (up to 2^{64} bytes)
	[38:26]	[76:64]	OFFSET	Indicating the byte offset within the object to write or read 26-bit for the option Req^T, 64-bit for the option Req^G
	-	[140:4]	-	For byte-wise alignment for data
	[64:8N]	[144:8N]	DATA_TO_WRITE	Data to write If OP_TYPE is not write request, dummies are filled. N here means the byte size of data, SIZE
FPGA to Enclave	[0]		STATUS	Indicating the status of the request 0x0: request success 0x1: request fail
	[1:10]		ID	Indicating the allocated ID of the data (supporting range: 0~1023)
	[11:5]		-	For byte-wise alignment for data
	[16:8N]		DATA_TO_READ	Read data If OP_TYPE was not read request, dummies are filled. N here means the byte size of data that is received via SIZE

Figure 8: The data format of transaction packet between TrustLib and TrustMod.



(a) Native SGX (1st input) **(b) Native SGX (2nd input)** **(c) TrustOre (1st input)** **(d) TrustOre (2nd input)**

Figure 9: Memory Access Pattern Comparison for nbench (Num sort)

the other hand, the memory trace shown in the TrustOre-based program (Figure 9c and Figure 9d) is constant during executing the heap sort function. TrustOre-based program shows the same trace even if the input data is changed as shown in Table 4. The only non-constant trace pattern that an attacker can observe are what is observed at the beginning and end of the program but it leaks no meaningful information to the attacker. The ascending trace patterns at the beginning and end are for moving data array from the enclave to the FPGA memory and vice versa, respectively.

We also count the number of write and read to show TrustOre protects the access type, too. Recall, TrustLib executes the same number of write and read at each request regardless of the actual access type by inserting dummy write or read as explained in §4.3. As a result, TrustOre-based program balances the total number of write and read as shown in Table 5, while the native SGX shows the unbalanced write/read ratio. More specifically, TrustLib executes four memory accesses, i.e., two writes and two reads sequentially at each write or read request for the data block. Therefore, total memory access count of TrustOre is about four times larger than that of the native SGX. The reason why TrustOre needs a little more than the four times larger number is the additional memory operations at the beginning and end as mentioned earlier.