XTENSTORE: Fast Shielded In-memory Key-Value Store on a Hybrid x86-FPGA System

Hyunyoung Oh[†], Dongil Hwang[†], Maja Malenko[‡], Myunghyun Cho[†],
Hyungon Moon[§*], Marcel Baunach[‡] and Yunheung Paek[†*]
[†]ECE and ISRC, Seoul National University, [‡]Graz University of Technology,
[§]Ulsan National Institute of Science and Technology (UNIST)

***Abstract*—We propose XTENSTORE, a system that extends the existing SGX-based secure in-memory key-value store with an external hardware accelerator in order to ensure comparable security guarantees with lower performance degradation. The accelerator is implemented on a commodity FPGA card that is readily connected with the x86 CPU via PCIe interconnect to form a hybrid x86-FPGA system. In comparison to the prior SGX-based work, XTENSTORE improves the throughput by 4–33×, and exhibits considerably shorter tail latency (>23x, 99th-percentile).**

***Index Terms*—SGX, Key-Value Store, FPGA**

I. INTRODUCTION

An in-memory *Key-Value Store* (KVS) is used as a temporal data store in many applications by providing a set of concise interfaces to store and retrieve key-value pairs. This in-memory KVS is now indisputably an integral part of modern cloud services. Almost all popular cloud services are known to use in-memory KVSs, such as Memcached at Twitter/Facebook/Dropbox and Redis at GitHub.

Despite their significance, popular KVSs do not offer confidentiality or integrity guarantees strong enough to protect sensitive data against cloud platform providers. When a service runs its KVS on a public cloud, the KVS has no choice but to trust the platform provider to prevent corruption or eavesdropping of its contents. In other words, the provider is technically entrusted with access privileges to KVS contents, even if written contracts strictly disallow the privileges from being ever exercised. This exposure to a potential security threat from malignant platform providers is inevitable because a KVS runs under the supervision of the privileged software layer (e.g., OS, hypervisor) on the cloud platform. A cloud application could encrypt the key and data before storing them to mitigate such a threat. However, as the privileged software has access to the cryptographic key, it can decrypt the KVS contents whenever needed.

This limitation of the contract-based security guarantees and the increasing significance of KVSs call for a new mechanism to shield KVSs with Intel SGX [1], the x86 architecture extension for *Trusted Execution Environment* (TEE). However, SGX also has a physical limitation that its protected memory, called *Enclave Page Cache* (EPC). The EPC is too small to accommodate the entire KVS, which inevitably incurs frequent page swapping, significantly increasing the average memory access latency. To overcome this size limitation, ShieldStore [2] proposed a variation of Memcached, which is tailored for SGX in order to minimize the average latency by directly managing the unprotected memory to store key-value pairs, rather than relying on the system's built-in page swapping. Although their direct management improves the throughput of a shielded KVS, we have observed that their solution still suffers from the inherent performance limitations of a pure software mechanism running on the CPU. This performance overhead increases as more key-value pairs are protected because protecting a larger volume of data requires a deeper Merkle tree, increasing the amount of Message Authentication Code (MAC) computing for each external memory access.

The purpose of our work is to develop an extension of shielded KVS, called XTENSTORE, that tackles the performance problem of existing software-only solutions while providing comparable security guarantees. For this purpose, we have built XTENSTORE on a *hybrid x86-FPGA* system which is nowadays present in many commercial cloud platforms where providers offer their tenants FPGA instances for hardware acceleration of diverse computational tasks. In our target system, FPGA is connected to an x86 CPU via PCIe interconnect, and works as an external accelerator by leveraging its parallel computing facilities for KVS operations and on-package memory for caching sensitive data. Like existing solutions, XTENSTORE also relies on SGX as a TEE to securely process and store key information for a shielded KVS. But unlike them, XTENSTORE executes time-consuming, parallel KVS tasks on FPGA for improving the overall performance. In addition, to maintain the same level of security as existing solutions, we augment FPGA with security elements and extend the trust of SGX to FPGA by converting an ordinary FPGA into a TEE that shields the entire KVS, allied with SGX on the host CPU.

All in all, XTENSTORE aims to ensure the security of shielded KVS while taking advantage of FPGA to overcome the size and performance limitation of SGX. The main body of KVS resides in an FPGA accelerator to enjoy larger space and higher performance. SGX on the CPU side initializes the KVS components and manages sessions between the KVS on FPGA and clients on the remote site. To maximize the performance of XTENSTORE, we design the two-tiered memory architecture in which the (1st tier) FPGA on-package memory stores a small portion of information for efficiently processing both KVS and MAC operations while the (2nd-tier) host memory stores the remaining most of KVS data. We demonstrate the efficacy of

[*]Corresponding authors

978-3-9819263-6-1/DATE22/©2022 EDAA

XTENSTORE in an experimental setting where KVS services run on an SGX-enabled Intel i7-6700 CPU and a Xilinx Zynq-7000 series FPGA board. Our experimental results show that XTENSTORE exhibits 4–33× higher throughput compared to ShieldStore. In addition, XTENSTORE exhibits considerably shorter tail latency, more than 23× in 99th-percentile.

II. BACKGROUND AND MOTIVATION

Cloud apps usually maintain a large portion of their data in in-memory KVSs, enabling clients to store (*PUT*) and retrieve (*GET*) values associated with keys simply and efficiently. With the goal of efficient lookup operations, hash tables are common practice. The importance of KVS as a core software component running on an untrusted cloud platform motivated researchers to develop SGX-protected KVSs. SGX provides a TEE, called *enclave*, along with EPC. Since EPC is currently limited to 128MB, an enclave demanding additional storage to hold more KVS data must evict some pages to the untrusted main memory, involving costly operations for page granularity encryption and MAC generation. ShieldStore tries to address this performance drawback by placing a large portion of the hash table outside EPC with manual protection performing encryption and MAC for each KVS operation. However, we have discovered that despite several optimizations, ShieldStore still suffers from considerable performance overhead for encryption and MAC, while maintaining its Merkle tree for sensitive data encrypted inside the untrusted memory. We view such overhead as inherent to any shielded KVS relying only on SGX with limited computing resources. Our discovery motivated us to seek a solution that tackles the performance problem by capitalizing on salient features of FPGAs. One such feature is a tightly coupled memory, called *High Bandwidth Memory* (HBM), that state-of-the-art FPGAs provide within the same package [3] to transfer data at maximum speed, yet safely from physical attacks. Another is an array of multiple FPGA modules that can be dedicated to accelerate shielded KVS operations by performing frequent cryptographic functions in parallel. One more feature is the physically isolated environment of FPGA that enables the transformation of an ordinary FPGA into a TEE by installing security primitives, as described in [4]. Once a TEE is built inside FPGA, it becomes a trusted ally of the SGX enclave to protect XTENSTORE against untrusted privileged software.

III. DESIGN AND IMPLEMENTATION

XTENSTORE on the FPGA TEE extends an in-memory KVS of an SGX enclave. Figure 1 shows a high-level overview of the x86-FPGA system that runs XTENSTORE. A client can employ XTENSTORE as its KVS just like ShieldStore or other shielded KVSs. The client establishes a secure channel with the enclave by following the standard protocol [1] including attestation and key generation ❶. XTENSTORE then generates a session key for KVS transactions and provides the client with the key ❷. Using the session key, the client interacts directly with XTENSTORE through a dedicated network interface that most FPGAs have ❸. Optionally, the enclave can also place KVS transactions to

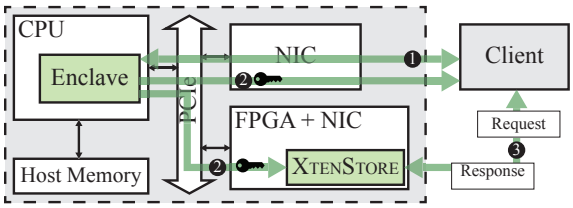


Fig. 1: An overview of XTENSTORE.

XTENSTORE through PCIe interface, either to bridge the client and XTENSTORE, or to become one of XTENSTORE's clients.

Threat Model. We consider a common SGX threat model [5] where all software entities in the cloud are assumed to be potentially malicious. An SGX-enabled CPU and the FPGA card are the only components that XTENSTORE trusts. Like other SGX-based solutions, we do not prevent denial-of-service attacks caused by compromised system software. We consider that our design is resilient to memory-based side channels [4], but other forms of physical side channels such as power monitors, are out of scope.

Challenges. XTENSTORE is a KVS whose security is guaranteed by the FPGA TEE as well as the SGX-enabled host CPU. Consequently, unlike prior FPGA-accelerated KVSs [6], [7] whose design challenge is solely on efficiently utilizing FPGA as a workhorse, XTENSTORE has an additional challenge: how efficiently to transform the workhorse into a TEE providing the needed security primitives, such as isolated execution, secure storage, and remote attestation, which are essential to guarantee confidentiality, integrity, and freshness of KVS data. Efficiency is of primary priority in our design because such guarantees entail increased data transfer and expensive cryptographic operations for encryption and MAC. More specifically, Merkle tree traversing for freshness verification significantly increases the amount of data transfer between the host memory and FPGA, potentially exhausting the already scarce PCIe bandwidth.

Two-tiered Hash Table Architecture. XTENSTORE minimizes the cost of freshness verification by implementing a *two-tiered hash table*. The 1st tier table on the FPGA memory keeps the key, pointer, and secret. The key-value pairs are stored in the 2nd tier table in the host memory with cryptographic protection (i.e., AES-CTR encryption and hash MAC). The corresponding host memory address is stored as a pointer in the 1st tier table. The per-key secret is combined with the encrypted value and IV/CTR when generating MAC. This two-tiered table is designed to meet the two requirements arising from the limited size of FPGA memory and long latency of host memory access.

First, the key-value pairs must reside in the host memory because the FPGA memory is too small. In a typical setting, the host memory can be about 16 times larger than the FPGA memory (in our system, 4GB vs. 64GB), and the FPGA memory alone is not large enough to contain a real-world in-memory KVS. Thus in our design, all key-value data are stored by default in the host memory. As an exceptional case, XTENSTORE puts a small-sized (e.g., 16B) value into the FPGA memory to

maximize performance by eliminating memory accesses to the host. In general cases with larger (> 16B) data stored in the host memory, it minimizes the number of host memory accesses by keeping the pointers to the corresponding data, enabling it to PUT/GET the data just with single access by referencing the per-pair pointer. If FPGA cannot hold as many pointers as needed due to the large KVS size, it can accommodate all key-value pairs by adopting the chaining hash table design.

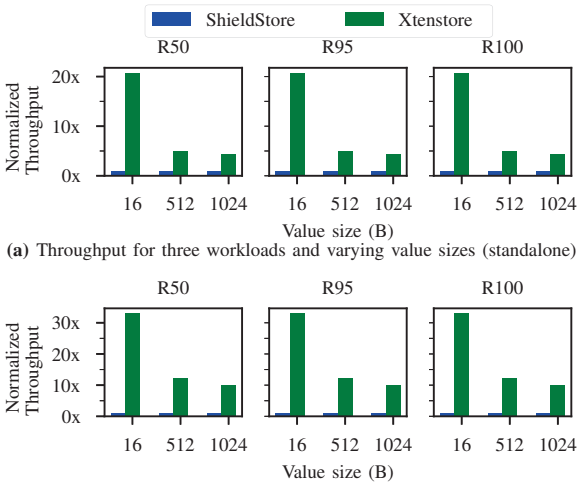
Second, data stored in the host memory must be cryptographically protected. To meet this requirement, XTENSTORE generates and verifies MAC independently using the individual secret instead of chaining all dependent MACs in a Merkle tree as done in prior work [2]. The secret is randomly generated and kept in the FPGA memory at every PUT operation to protect the freshness. This design allows XTENSTORE to prevent attackers from computing a valid MAC out of any chosen key-value pair they want to inject without using a Merkle tree, which entails high memory delays. The only way to subvert the freshness guarantee (i.e., to pass the MAC verification with a corrupted key-value pair) is obtaining the secret, which is unlikely since our design of FPGA TEE ensures that the secret never leaves FPGA or its on-package memory.

Concurrent Table Lookup and Cryptography. When processing a PUT to add or update a key-value pair, XTENSTORE encrypts the value, computes the MAC from the encrypted value combining with the other data (secret, IV/CTR), and stores each data into a proper location. In this process, the computational time of encryption and MAC adds to the already high PUT latency increased to access the host memory from FPGA. XTENSTORE resolves this performance problem by simultaneously carrying out table lookup and cryptographic operations. This optimization was possible because a cryptographic operation is independent of the output of table lookup, and vice versa. This mutual independence is thanks to our novel design of XTENSTORE that uses a new secret for every PUT transaction and hires the two-tired hash table where the 1st tier table is not encrypted and XTENSTORE does not have to verify its authenticity. We have endeavored our FPGA TEE to exert its full potential power of parallel computation by simultaneously performing these two types of time-consuming operations.

IV. EVALUATION

We use the publicly available *uniform* workload [8] to evaluate the effectiveness of XTENSTORE with the fixed key size of 16B, varying value sizes (16–1024B), initial KVS sizes (4–32GB) and request distributions (50% read, 95% read, 100% read). All our evaluations were conducted on Intel Core i7-6700 with 64GB of main memory and running Ubuntu 16.04 with Linux 4.15.18 (64-bit). XTENSTORE is loaded and operating on the Xilinx ZC706 evaluation board with XC7Z045 FFG900-2 and 4GB DDR3 RAM which is connected to the SGX-enabled CPU via PCIe. We measured the throughput by sending 100k requests to the key-value pairs preloaded.

Throughput in Standalone Mode. We measured the number of operations that XTENSTORE and ShieldStore execute every second in the standalone mode where they receive the requests from another process running on the same machine. Figure 2a shows that XTENSTORE attains 628–8995kops/s, thus greatly outperforming ShieldStore by 4–20× where KVS size is 4GB. It is noticeable that the best speedup (i.e., 14–20×) is achieved when the size of values is small enough to fit entirely into the FPGA memory. We also evaluated how KVS size affects the throughput of XTENSTORE, which reveals us that XTENSTORE can accommodate a large number of key-value pairs (8, 16, 32GB) and maintain the constant throughput.



(a) Throughput for three workloads and varying value sizes (standalone)

(b) Throughput for three workloads and varying value sizes (networked)

Fig. 2: Throughput of XTENSTORE and ShieldStore running in standalone and networked mode

Throughput in Networked Mode. The efficacy of XTENSTORE was tested in the network mode where XTENSTORE directly interacts with clients via the network interface as described in Figure 1. Prior work [6] asserts that the proximity to the network interface helps KVS with the network-capable accelerator to exhibit higher throughput. To see that this assertion is valid for XTENSTORE, we measured throughput by sending transactions to XTENSTORE using an emulated network interface, and compared the throughput results against those of networked ShieldStore. We fixed KVS size to 4GB for the evaluation. As shown in Figure 2b, XTENSTORE achieves relatively better speedup (10–33×) compared to ShieldStore in the network mode than in the standalone mode.

Latency. Table I shows latency comparison between XTENSTORE and ShieldStore. The latency here corresponds to the difference between the time a request is sent and the time the response has arrived. Among the two modes described above, we used the standalone mode in this experiment to rule out the effect of network delay. We initialized KVSs with 1GB of key-value pairs and value sizes with 512B and 1024B. The results reveals that XTENSTORE exhibits a better latency characteristics than ShieldStore, especially for the 95th-percentile and 99th-percentile latencies. These tail latency results listed in Table I suggest that 95% of the requests will experience similar latencies

TABLE I: Transaction latencies: XTENSTORE vs. ShieldStore (μs).

Val size, Workload		ShieldStore				XTENSTORE			
		min	avg	95th	99th	min	avg	95th	99th
512B	R50	0.33	4.39	5.77	140.54	2.94	4.90	5.77	5.98
	R90	0.33	4.56	6.12	146.70	2.95	5.63	5.90	6.02
	R100	0.31	4.64	4.32	162.59	4.90	5.71	5.92	6.03
1024B	R50	0.34	6.61	14.68	212.36	2.94	5.19	6.37	6.55
	R90	0.35	6.28	11.17	209.47	2.95	6.16	6.43	6.58
	R100	0.35	6.53	11.67	211.75	4.94	6.26	6.45	6.59

TABLE II: FPGA utilization after adopting XTENSTORE

XTENSTORE Modules	FPGA Resource		
	LUTs	FFs	RAMB36
Hash Table Cores	40415	49397	327.5
Host Memory Interface	16277	13918	0
Crypto Cores	118943	59158	0
FIFOs	199	0	79.5
Network/DMA/PCI IPs	26538	28631	0
Sum	202372	151104	407
Utilization %	92.58%	34.56%	76.68%

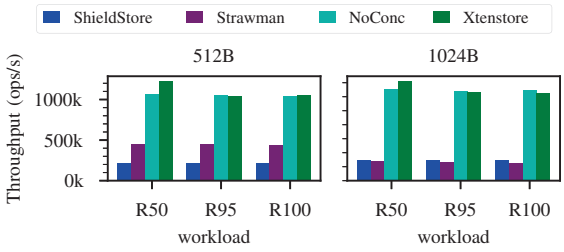


Fig. 3: Impact of design decisions. Strawman is a version with Merkle tree and NoConc is a version with concurrent execution disabled.

when served by either ShieldStore or XTENSTORE while the rest experiences longer latencies with ShieldStore. As XTENSTORE shows 23–32× lower 99th-percentile latency. It would be a much more desirable component for a cloud service in which each component has to satisfy certain Service-Level Objective (SLO) which usually includes the tail latency.

Design Choices. We compared XTENSTORE with two variants that only implement a subset of our design to evaluate the impact of our two key design decisions. The first one, *Strawman*, is identical to XTENSTORE except that it uses a Merkle tree for freshness like ShieldStore. Given KVS size of 4GB, Figure 3 shows that Strawman attains higher throughput only for relatively small values (512B). For larger values (1024B), it records poor results even compared to ShieldStore, albeit its performance benefits from FPGA. In contrast, XTENSTORE gains 2.2–5.3× higher throughput than Strawman, attesting that our design choice of replacing Merkle tree with the two-tiered hash table plays a key role in the performance gain. The second is, *NoConc*, that does not execute cryptographic functions concurrently. Comparing NoConc with XTENSTORE would show us how much performance has been improved by hiring FPGA to exploit parallelism embedded in KVS transactions. Not surprisingly, the workload with many PUTs (e.g., R50) does benefit from concurrent execution, achieving about 1.15× speedup.

Hardware Cost. Table II presents the number of logic cells that XTENSTORE uses to fulfill the maximum performance. The number shows that XTENSTORE consumes a majority amount of logic cells for Crypto Cores. This is owing to our FPGA design that prefers to use as many dedicated modules as possible to minimize the delay of cryptographic operations.

V. CONCLUSION

The purpose of our work is to employ FPGA not only as a workhorse, as done in prior work, but also at the same, as a TEE that allies with SGX to protect in-memory KVS against untrusted cloud providers. One of our technical contributions is a two-tiered memory architecture tailored for hybrid x86-FPGA. FPGA modules are parallelized to minimize the latency of KVS operations. The evaluation of XTENSTORE on a commodity FPGA demonstrates dramatic speedups (4–33× for the throughput) in comparison with ShieldStore. Moreover, XTENSTORE exhibits 23–32× shorter 99th-percentile latency, which makes XTENSTORE more attractive for cloud services.

ACKNOWLEDGMENT

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(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]), (No.2020-0-00325,Traceability Assuarance Technology Development for Full Lifecycle Data Safety of Cloud Edge), (No.2021-0-00724, RISC-V based Secure CPU Architecture Design for Embedded System Malware Detection and Response), (No.2021-0-01817, Development of Next-Generation Computing Techniques for Hyper-Composable Datacenters), and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2020R1A2B5B03095204), (No. NRF-2021R1F1A1050311), and the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2021. The EDA tool was supported by the IC Design Education Center(IDEC), Korea.

REFERENCES

[1] "Intel® software guard extensions programming reference," 2014.
[2] T. Kim *et al.*, "Shieldstore: Shielded in-memory key-value storage with sgx," in *EuroSys*, 2019.
[3] Xilinx, "Virtex ultrascale+ hbm fpga: A revolutionary increase in memory performance," 2017.
[4] H. Oh *et al.*, "Trustore: Side-channel resistant storage for sgx using intel hybrid cpu-fpga," in *CCS*, 2020, pp. 1903–1918.
[5] A. Baumann *et al.*, "Shielding applications from an untrusted cloud with haven," in *OSDI*, 2014, pp. 267–283.
[6] B. Li *et al.*, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *SOSP*, 2017, pp. 137–152.
[7] Y. Qiu *et al.*, "Full-kv: Flexible and ultra-low-latency in-memory key-value store system design on cpu-fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1828–1444, 2020.
[8] "ShieldStore," https://github.com/cocoppang/ShieldStore, accessed: 2021-08-03.