Enclave-Accelerated Replay: Efficient Integrity for Server Applications

Ahmed Awad
Brad Karp
University College London (UCL)
London, United Kingdom

ABSTRACT

The limitations of the two main approaches to enforcing integrity of execution on third-party servers, enclave-based execution (e.g., Intel SGX) and log-and-replay (e.g., Orochi), constrain their applicability. Early implementations of SGX enclaves provide strong integrity, but are tightly limited in memory size (to 96 or 192 MB). The recent Ice Lake SGX implementation supports larger enclaves, but only by trading off a weaker integrity guarantee on memory. Log-and-replay offers a strong integrity guarantee without constraining an application's memory size, but instead incurs the compute cost of re-executing the verified application on an offline trusted server. In this paper, we illustrate that each of these two approaches is well suited to addressing the other's shortcomings, and that one need not sacrifice strength of memory integrity guarantee to accommodate applications with large memory footprints, nor incur the cost of re-executing an entire application offline. We propose a hybrid enclave/log-and-replay design for checking the integrity of application execution on an untrusted server. More broadly, we identify TEEs' potential as accelerators for offline verification, as TEEs can obviate re-execution of parts of the application on the offline trusted server.

ACM Reference Format:
Ahmed Awad and Brad Karp. 2022. Enclave-Accelerated Replay: Efficient Integrity for Server Applications. In *Proceedings of Proceedings of the 5th Workshop on System Software for Trusted Execution (SysTEX '22 Workshop).* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

In the era of cloud computing, many deployers of server-side applications run code as tenants on data-center-hosted virtual or physical servers whose hardware is owned and operated by a third party. They choose to do so because a cloud provider's economy of scale typically allows pricing of resources allocated to tenants (CPU, bandwidth, storage) more cheaply than a tenant could provision the equivalent dedicated resources. Tenancy incurs concern over execution integrity, however: a deployer may worry whether the whole stack of hardware and software (machine, hypervisor, OS, application code) running in the cloud *faithfully executes* the deployer's application—i.e., whether her application will yield the same results running in the cloud environment as it would when run on a hardware/software stack controlled by the deployer.

Prior work on ensuring integrity of execution on an untrusted server has fallen broadly into two categories, each with distinct material limitations:

- **enclave-based execution** of application code, as most widely deployed in Intel's Software Guard eXtensions (SGX) [6]: SGX guarantees integrity and confidentiality[1] for execution within an *enclave*, an encrypted memory region containing code and data that the CPU hardware isolates strongly from the rest of the machine's hardware and software (including from the OS and hypervisor). Early implementations of SGX provided strong memory integrity, including protection against replay, but suffered from severe constraints on the amount of memory within an enclave: the enclave page cache (EPC) was restricted to 192MB [11]. While Intel has recently introduced an SGX implementation that allows much larger enclave memory regions, this design only offers weaker integrity guarantees (by dropping protection from replay attacks on memory hardware) [12, 14]. AMD's SEV [1] makes a similar trade-off of increased memory size for a weaker integrity guarantee in its encryption of virtual machines' memory. We discuss these integrity strength/memory size trade-offs further in Section 2.3.
- **log-and-replay of execution**, exemplified by Orochi [18] and Cobra [19]: A server logs the requests and responses that it receives and sends, along with untrusted "advice" describing scheduling. The deployer then periodically ships the log to a trusted verifier box that holds a full copy of the deployed software stack. The verifier re-executes the server's computation from the log's contents to check that the application generates the same responses in the server's log. The verifier accelerates re-execution by batching requests from the log and executing them in SIMD-like fashion [18]. Acceleration notwithstanding, the verifier still incurs a significant CPU cost beyond the untrusted server's cost. There is also significant latency between execution of a request on the server and checking of that request's execution on the verifier. While Tan et al. suggest copying the log from server to verifier once a day [18], even if one streamed the log from server to verifier, the verifier's batching of requests for acceleration increases the latency of integrity checking.

We observe that while the above two approaches suffer from limitations individually, each is particularly well suited to remedy the other's: log-and-replay is free to use a machine's entire physical memory for applications with large in-core data sets, and executing a portion of an application in a size-constrained enclave provides strong integrity protection without the CPU cost of re-executing that portion of the application on an offline verifier. This complementarity raises a tantalizing possibility: can one use enclave-based execution as an "accelerator" for compute-intensive portions of

SysTEX '22 Workshop, @ASPLOS, Lausanne, Switzerland
2022.

[1] We deem confidentiality as out of scope in this work; we concern ourselves only with integrity.

an application's execution, in that such portions need not be re-executed on a verifier, yet use log-and-replay for integrity-protected execution of the remainder of the application, with freedom to use the machine's entire memory (without the severe limits of EPC size)? In this paper, we answer this question in the affirmative: we sketch the design of Enclave-Accelerated Replay (EAR), a hybrid enclave/log-and-replay approach to application integrity protection on cloud servers. While we focus herein on SGX enclaves in the interest of exploring a concrete design for a Trusted Execution Environment (TEE) that is already widely deployed, we expect this hybrid approach can extend to emerging TEEs for special-purpose compute accelerators, such as for GPUs [9, 21].

Whether EAR is viable rests largely on performance: can the domain crossings and data motion between in-enclave and extra-enclave code be made sufficiently efficient for EAR to perform well? We expect that for applications with a large total memory footprint, but that execute many memory-limited, compute-intensive sub-tasks that fit in an enclave, EAR should perform well. Such applications will reap the benefit of eliding re-execution of compute-intensive sub-tasks on an offline verifier, yet should not incur too much cost from domain crossings and data motion across enclave boundaries. We demonstrate that humble locality-aware blocking of data upon entry into an enclave can significantly outperform the best known techniques for paging EPC memory to extra-enclave memory (by up to 1.5-2x). We conclude that EAR's approach to integrity assurance for cloud applications holds promise as a means for escaping the memory limits of today's enclaves that provide the strongest integrity guarantee, while reducing the computational costs of the verifier required in log-and-replay designs.

2 BACKGROUND & PROBLEM DEFINITION

After stating the threat model for integrity verification systems for server applications, we articulate the goals for such systems, and assess how well today's two main design approaches for such systems meet those goals.

2.1 Threat Model

We assume that an adversary can compromise the OS and hypervisor running on the untrusted cloud provider's hardware. We further assume that she can compromise any hardware in the untrusted machine but the CPU, and present arbitrary network input. We trust the development environment, administrator-provided computational resources, compiler toolchain, and Intel-provided SGX infrastructure (i.e., SGX SDK, attestation infrastructure, and SGX platform services). Like prior work on execution integrity, ours leaves denial of service and hardware side channels out of scope. SGX is rife with implementation vulnerabilities [10, 20, 22, 23]. We target it as a concrete example of an enclave substrate, and expect future SGX implementations to improve, and our design to generalize to other enclave platforms [7, 13].

2.2 Goals

The following are desiderata for a system that ensures execution integrity of cloud-hosted server applications:

Soundness & Completeness. The system detects (or thwarts) all attempts by an adversary to violate the application's execution integrity (i.e., no false negatives), and does not detect violations if the cloud server *(executor)* faithfully executed the program (i.e., no false positives).

Property	SGX	Auditing	Hybrid
Soundness & Completeness	✓	✓	✓
Multi-threaded	✓	✓	✓
Efficient	At the cost of integrity	✓	✓
Timely	✓		✓
Persistence support		✓	✓
Language agnostic	✓		✓
Generality	++	+	+++

Table 1: Properties of three approaches.

Multi-threaded. The system supports multi-threaded applications.

Efficient. The system imposes only a moderate performance cost.

Low Extra-cloud Resources. As much of the computational cost of application execution and integrity checking as possible should be borne by *cloud-provided* computational resources, as they are cheaper than deployer-provisioned, dedicated resources.

Timely. The system supports timely integrity violation detection. If an executor is not faithful in its execution, then the system should reveal it within a limited period. The tightest such limitation would be to detect integrity violations before they become visible to clients.

Integrity-protected Persistence. In line with many cloud applications' needs, the system should offer mutable storage that supports validation of data integrity.

Language-agnostic. While a good fit for some web applications, interpreted languages introduce overhead that is a poor fit for compute-intensive applications. To accommodate such applications, the system must support unmanaged languages (e.g., C++, Rust, &c.).

Generality. The system should offer the above properties for all applications. Of course, the system may perform better for certain classes of applications than others.

We next consider the extent to which Intel's Software Guard eXtensions (SGX),[2] a widely deployed enclave design, and log-and-replay audit fit the above desiderata.

2.3 SGX and Its Constraints

Intel SGX is a set of architectural extensions to x86-64 CPUs that lets applications execute procedures using *enclave* memory, an encrypted and integrity-protected region of memory allocated from a pool known as the Enclave Page Cache (EPC). Intel's SGX implementations to date have made two rather different trade-offs between limitations on EPC size and strength of the integrity guarantee on enclave memory. Earlier SGX implementations placed a Memory Encryption Engine (MEE) in the CPU package, and used a tree-based MAC/counter scheme to protect the integrity of an enclave's

[2] We focus on SGX, as most cloud providers offer Intel CPUs and offered SGX by 2021.

memory contents, where the tree's root node is stored within the MEE (i.e., on-chip). The use of counters, which increment on writes, provides strong integrity protection, including against replay: even an adversary that can read and write DRAM directly (without going through the CPU) cannot replay prior valid memory contents, as the adversary cannot modify the root node within the MEE.The cost is complexity: this MEE-based approach stores all nodes but the root of the tree in DRAM, and thus imposes significant DRAM bandwidth requirements for tree reads and updates. As the EPC grows, so does the depth of the tree and the DRAM bandwidth for checking and updating the tree's contents on every memory access. Intel was compelled to limit the maximum application-accessible EPC size to 192 MB in the most recent version of MEE-based SGX hardware to bound tree depth (and the concomitant DRAM memory bandwidth).

Most recently, in the Ice Lake microarchitecure's SGX implementation, Intel has changed tack significantly by eliminating the MEE in favor of Total Memory Encryption (TME), which stores all MAC-related state in DRAM, and does not use a tree-based MAC. Because the TME-based implementation of SGX eschews checking (for reads) and updating (for writes) multiple levels of a tree-based MAC data structure, it escapes the MEE-based implementation's tight 192 MB EPC size constraint, and supports enclaves hundreds of GB in size. However, this increase in enclave memory size limit comes at the cost of weakened integrity protection on an enclave's contents: this new TME-based SGX implementation cannot protect against replay attacks on DRAM contents by attackers who can access DRAM directly (without going through the CPU) [12, 14]. This change is somewhat insidious, in that it represents a material weakening of SGX's original integrity guarantee, while Intel still refers to the TEE design as SGX, and an enclave's memory as the EPC. Memory replay attacks may well be an unacceptable integrity risk for application deployers who do not trust the owner and/or operator of cloud-based server hardware. For those deployers, MEE-based SGX's stronger integrity guarantee will be a far better fit, though it comes with a tight 192 MB EPC size limit. In this paper, we concern ourselves with deployers and applications that demand MEE-based SGX's stronger integrity.

SGX prohibits enclave procedures from invoking system calls. Instead, the *untrusted* portion of the application outside the enclave must "proxy" syscalls for enclave code. Switching between untrusted and trusted execution requires a costly hardware context switch. While enclave code may access untrusted memory, for integrity guarantees to hold, it must execute integrity-critical operations in enclave memory. Thus enclave procedures first copy arguments into trusted memory before executing.

SGX allows applications to oversubscribe the EPC by paging enclave memory to untrusted memory. An SGX kernel module services page faults on EPC memory by using privileged instructions to evict pages from the EPC to untrusted memory. The eviction process is as follows: after selecting a page for eviction, SGX blocks all enclave threads from accessing that page, and flushes all TLB entries that reference that page. Finally, MEE-based SGX encrypts the page using AES-GCM and writes it to untrusted memory. Context switches, cryptographic operations, and inter-processor synchronization make this process expensive.

While MEE-based SGX offers a strong integrity guarantee, its EPC size limit constrains application size, and the syscall restriction mandates at least an untrusted shim layer to invoke syscalls on behalf of enclave code. To achieve performance and correctness, developers who want MEE-based SGX's strong integrity guarantee must take on the onerous task of carefully partitioning their applications between enclave and non-enclave code, while ensuring that applications do not blindly trust the results of system calls or oversubscribe the EPC.

Some prior work [2, 4] resorts to pulling much of the OS into the enclave to avoid partitioning the application. Given MEE-based SGX's constrained EPC, this approach doesn't scale to applications with a large memory footprint. Some systems [16, 17] attempt to escape EPC size constraints on MEE-based SGX implementations by improving the performance of EPC paging with a trusted, in-enclave paging mechanism. These approaches still incur significant costs, however (see §4).

Finally, many server applications need mutable persistent storage. SGX's sealing primitive does not provide *temporal* integrity, and thus leaves mutable storage susceptible to rollback attacks.

2.4 Auditing and Its Costs

Auditing an application's execution integrity, by contrast, does not require partitioning the application, nor does it limit the application's memory footprint. Auditing assumes that the application is untrusted; it executes outside of any TEE. The deployer instead instruments the application to gather untrusted information about the application's control flow, thread scheduling, and other non-deterministic events, and store it in *advice* files. The server application communicates with clients via an in-enclave TLS stack. Terminating TLS in the enclave allows an in-enclave *trusted observer* to faithfully record clients' requests and responses in a *trace* [3]. The trace is an immutable, append-only log; the *trusted observer* stores the full trace on a cheap untrusted disk, and records a compact count of trace entries and their hashes in a *trusted store*.

A trusted *verifier* verifies that the executor faithfully executed the application by:

(1) fetching the contents of the trusted store, advice files, and trace,
(2) validating the trace's contents by computing the hash of each entry and comparing the computed hashes against the hashes the verifier fetched from the trusted store,
(3) re-executing the application on the requests in the trace, and finally,
(4) comparing the re-execution's output against the responses in the trace. If they match, the verifier deems the execution faithful.

The verifier takes advantage of request batching and the untrusted application's advice to eliminate operations during re-execution, achieving better performance than naively re-executing the application on client requests (5-10x times faster for PHP-based web applications [18]). However, this approach's performance is dependent on commonality in client request executions. Moreover, the verifier achieves acceleration only when it executes a batch of requests per unit of control flow, and batching makes detection of integrity violations less timely.

State-of-the-art auditing implementations [18] are not language agnostic, and target data-flow-like request-response processing

patterns. We observe that many RPC-like servers exhibit a similar processing pattern, so this approach may be widely applicable. However, at this writing, the only auditing system that supports concurrent server applications, Orochi [18], targets only *PHP* applications. Implementing auditing for an unmanaged language presents challenges unaddressed in prior work: an interpreter provides convenient interposition on an application's execution at the granularity of interpreter operations, which simplifies the implementations both of gathering advice and the verifier's SIMD-like acceleration technique. An unmanaged language, by contrast, offers no such interposition point. While tools such as Valgrind and Pin allow dynamic instrumentation of code in unmanaged languages, their performance overhead is unattractive, particularly given that developers choose unmanaged languages for performance reasons.

3 DESIGN

MEE-based SGX cannot accommodate applications with large data sets and does not offer temporal integrity for persistent storage. Log-and-replay audit incurs significant cost for re-execution on a trusted verifier box (even after batching-based acceleration), trades improved re-execution throughput for reduced timeliness of detection of integrity violations (because of batched trace processing), and has to date only been achieved for interpreted PHP.

We now sketch Enclave-Accelerated Replay (EAR), a design that remedies these lacunae. The central insight underlying EAR is that there is synergy between SGX and replay audit, which we believe is readily observable, though we do not believe this observation appears in the literature, or has served as the basis for a design to date. To aid exposition, we consider throughout the simple running example of an RPC-based image processing server application.

3.1 A Synergistic Hybrid

Table 1 summarizes the design goals that SGX and auditing satisfy. It is readily apparent that each goal unsatisfied by one is satisfied by the other. Could a hybrid of the two approaches inherit auditing's support for applications that use the untrusted server's full physical memory, and support for temporal integrity for persistent storage? And could it also leverage SGX to reduce the extra-cloud compute cost imposed by auditing's offline verifier? Such a hybrid design would entail running parts of an application in an SGX enclave and parts outside the enclave. How can we compose the integrity guarantees in each of these domains to achieve integrity for the whole application? And finally, are the overheads of crossing between those domains, both in execution and data motion, tolerable?

To reduce the execution cost of auditing's offline verifier, we must reduce the number of instructions that it must re-execute. Applying a filter to an image, for example, is a compute-intensive task that an image processing application might execute often. If the verifier did not have to re-execute this task for *any* request-response pair, it would re-execute faster. Doing so might render batches of fewer requests (or even single requests) efficient. What is needed is some means to trust *just* the execution of the filter, despite its running on the untrusted cloud server. One could then simply log the filter function's result on the untrusted server, and have the verifier use this "cached" result, rather than re-execute the filter function.

SGX conveniently does just this: it executes instructions with integrity on an untrusted machine. EAR executes image filtering as an enclave procedure, and records the inputs and output[3] to/from that procedure in a *procedure record* (PR). At audit time EAR's verifier uses results from PRs instead of re-executing the corresponding compute-intensive procedures. To do so, EAR's verifier first loads the inputs and output from the PR, then compares the inputs in the re-execution with the inputs in the PR. If they match, EAR's verifier continues executing with the "memoized" output from the record.

EAR in essence treats SGX as a *compute accelerator* for the verifier to improve its efficiency. Doing so reduces the extra-cloud resources consumed by auditing, instead shifting compute to the cloud provider, while maintaining integrity. There is a a further potential performance benefit. Recall that Orochi's verifier's performance relies on request batching, which increases the latency of verification. Eliding a function's re-execution on EAR's verifier significantly speeds the verifier even for a *single* request. So EAR should also offer more timely auditing.

PRs fit straightforwardly into an EAR untrusted cloud server's request-response log. Just as with requests and responses, EAR's trusted observer also persists PRs to the untrusted disk, and sends hashes to the trusted store.[4]

3.2 Architecture Overview

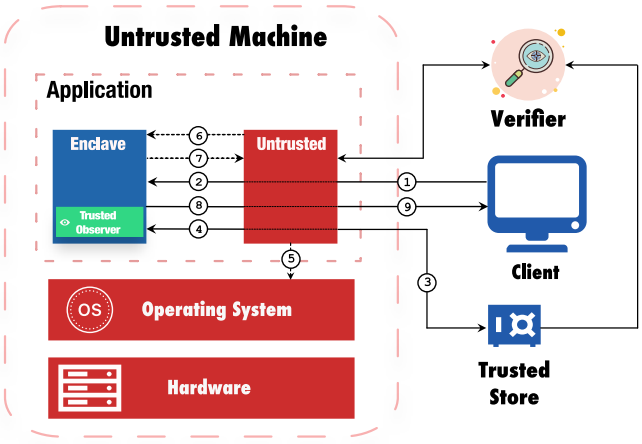

Figure 1: A high-level view of EAR's architecture. Solid arrows denote authenticated, integrity-protected communication; dashed arrows are procedure invocations.

Figure 1 depicts EAR's top-level components:

- the trusted observer, running inside the enclave, which records all inputs/outputs to/from enclave procedures, including client requests and responses,

[3] As stated in §2.3 any integrity-critical enclave procedure must copy inputs into the enclave and outputs out of the enclave.

[4] Note that the mechanism used during audit for checking the integrity of interactions between the application and the file system requires re-executing all instructions that led to a particular file modification. Therefore, EAR's verifier re-executes the operation that produced the file output.Why not just use a trusted store for the application's files? Parts of the application under audit are not trusted. For correctness, the verifier must re-execute the untrusted portions of the application regardless of whether the input came from a trusted store.

- a trusted store that holds hashes sent by the trusted observer,
- the untrusted component, which performs I/O on behalf of the trusted component and executes part of the request,
- an untrusted runtime that captures a tag representing the control flow a request experienced during its execution by the untrusted application. The runtime stores {request, tag} pairs in untrusted advice files on the untrusted server's disk.

Clients send requests to the server application over TLS-protected channels (1). The untrusted component forwards a request to the trusted observer for recording and decryption (2). The trusted observer records the request by storing a hash (3 & 4) of the request in the trusted store, then appending the request to the trace file (5). After updating its records, the trusted observer returns the plaintext request to the untrusted component. The untrusted component executes a request in a user-level thread (fiber). We adopt this concurrency model because the correctness proofs for Orochi's verifier (on which EAR's is based) assume a single thread of control executes a single request.

EAR's execution of the untrusted component of the application builds on Orochi's tracing: as the untrusted component of the application executes a request, EAR's runtime records control flow information and the output of non-deterministic functions in *advice* files. When EAR encounters an enclave procedure call in the untrusted component, it forwards the call to the trusted observer (6). The trusted observer executes the procedure and records the procedure's inputs and output in just the same way a client's request is recorded. After recording the enclave procedure's inputs and output, the trusted observer forwards the output to the untrusted component (7).

Once the untrusted component finalizes a response to a client, it passes the response to the trusted observer. The trusted observer records the response, encrypts it, then returns the encrypted output to the untrusted component (8). The untrusted component can now return the response to the client (9).

This order of operations is vital for temporal integrity: the trusted observer stores hashes in the trusted store *before* releasing the input/output to the untrusted machine. If an adversary erases entries from the untrusted disk, checking the trusted store's records will immediately reveal the truncation because the number of records in the trusted store will be greater than the number of entries on the untrusted disk.

Unlike Orochi, which targets interpreted PHP, EAR targets compiled C++ applications. EAR includes LLVM passes that statically instrument application code to collect control flow information. In the interest of efficiency, EAR captures control flow at basic-block granularity.

Our working EAR prototype compiles C++ application source into two executables: a logging binary that runs on the untrusted server, and a verifier binary that runs on a trusted machine. Implementation of the server and verifier sides of enclave acceleration is in progress; we already have a working trusted observer, building upon a previously published design [3].

4 PERFORMANCE

While our prototype does not yet let us evaluate the performance improvement EAR's verifier enjoys from SGX-obviated re-execution, there is an even more pressing, basic performance question about EAR's viability. EAR targets applications whose memory footprints cannot fit within MEE-based SGX's EPC, and executes only portions of those applications in enclaves. For EAR to be viable, it must compare favorably in application performance on the untrusted cloud server with the best known techniques for in-enclave SGX execution of *entire* applications that cannot fit in the EPC. To that end, we measure the relative performance of the sorts of control and data transfers EAR must make between extra-enclave and in-enclave code vs. the overheads of vanilla SGX paging and state-of-the-art in-enclave paging (IEP) techniques, as offered by Eleos and CosMIX.

For workloads with good reference locality but data sets that exceed the EPC's size, we expect an application using EAR or IEP to outperform one that relies on vanilla SGX paging, as the former two approaches avoid context switching to bring data into the enclave. They will also scale better than SGX paging in increasing thread count, because they avoid inter-processor synchronization, which may serialize threads in the presence of page faults. However, all three schemes must copy the data set from untrusted to trusted memory at least once. In the presence of good reference locality it is possible that both IEP and EAR will copy data into the enclave the same number of times, yielding similar performance.

For workloads with poor locality, we expect EAR to outperform both SGX paging and IEP. EAR relies on the programmer to explicitly indicate upon enclave procedure invocation which exact data to copy into the enclave. Paging, by contrast, is oblivious to the word-level access pattern of the application, and may thus evict pages that will be used soon thereafter.

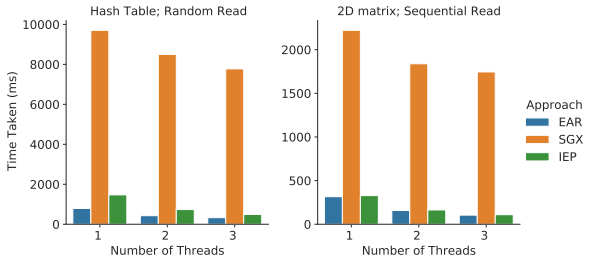

Figure 2: Time to complete a summation for values in a hash table and 2D matrix vs. number of threads for a data set that occupies 512MB (~5x EPC size). SGX's paging performance does not scale in thread count for large data sets as page fault synchronization serializes threads.

To evaluate our approach's overhead for an access pattern with good locality, we measure the time taken to compute the sum of a single 2D-array. The vanilla-SGX application copies the matrix into the enclave and then computes the sum in a single enclave call. The EAR-based application performs a row-wise copy into the enclave and summation, then computes the total sum from the vector of partial sums.

We perform the same computation on random elements of a hash table that maps 8 byte keys to 8 byte integer values to evaluate overheads for applications with access patterns that do not have good locality and with expensive to copy data structures. Note that we precompute the keys.

All experiments ran on a 6-core, SGXv1 Intel i7 8700 CPU at 3.20 GHz with 16GB of DDR4 RAM. We pinned threads to cores. To reduce SGX's context switching overhead, enclave threads do not invoke EEXIT; instead, they poll a lock-free queue. Untrusted code enqueues requests there to request service from the enclave.

The synthetic benchmark results[5] in Figure 2 confirm our hypotheses. For 32 MB data sets that fit inside the enclave (results elided for brevity), keeping the data set in untrusted memory and copying it into trusted memory before enclave execution introduces memory copy overhead.[6] Once the data set's size exceeds the EPC's size, EAR outperforms SGX and IEP because EAR leverages programmer knowledge about access patterns to avoid needless copying within full pages. Note that IEP and EAR perform similarly in the sequential access case as each copies the entire data set into the enclave once. In the random access case, EAR outperforms IEP by ~2X and SGX paging by ~10X.

5 DISCUSSION

While Ice Lake's TME-based SGX implementation offers the prospect of enclaves that are hundreds of GB in size, it only does so by weakening the integrity guarantee on enclave memory. Deployers who want integrity protection that includes replays will find the MEE-based SGX implementation a better fit. While several have proposed placing entire applications within an SGX enclave [2, 4, 5], MEE-based SGX implementations' constrained 192 MB EPC [11] cannot accommodate the many applications whose memory needs exceed that limit. And as we have illustrated experimentally, even the best known techniques for paging the EPC to non-enclave memory perform relatively poorly. We posit that enclaves are particularly well suited to serve as accelerators of log-and-replay integrity checking for an application's execution, by obviating re-execution of compute-intensive, memory-bounded portions of an application's execution. And we believe EAR's synergistic combination of enclave execution with log-and-replay audit holds promise for efficient integrity verification of native-code-compiled server applications with a large memory footprint.

To learn more about EAR's sphere of applicability, we intend to apply EAR to applications whose characteristics we expect to be a good fit. MapReduce applications [8] may fit EAR as they manipulate large data sets typically comprised of small objects amenable to processing in parallel. Tree-based key-value stores, such as Masstree [15], also process large data sets whose constitutent data units are small, and frequently perform computations on limited key ranges. Deep learning workloads often exhibit skewed access to a subset of data items, and thus may be a good fit for the EAR approach, particularly if the untrusted server's GPU or TPU provides TEE functionality, so that EAR can save the offline verifier re-execution of hardware-accelerated operations on the untrusted server.

We close by observing that if MEE-based and TME-based implementations of SGX coexisted within a single machine, they would complement each other well: the former could provide strong integrity and confidentiality for a "core" EPC of limited size, while the latter could provide confidentiality for a further "outer" very large enclave memory region. An application running under EAR on such a machine could enjoy secrecy and strong (replay-resistant) integrity guarantees over its entire memory footprint. TME would provide confidentiality for the application state in the outer enclave memory region, MEE would provide confidentiality and strong integrity for the application state in the core EPC region, and EAR would provide strong integrity for the application state in the outer enclave memory region and handle domain and data crossings between the core and outer regions.

[5] We model IEP's overhead by multiplying the number of pages SGX brings in by the time to copy one page into the enclave (without synchronization overheads). We resorted to this crude yet generous model because CoSMIX does not build in a 2021 SGX development environment.

[6] For data sets that fit inside the enclave, developers should place all data within the enclave (as SGX paging and IEP do).

REFERENCES

[1] AMD Corporation. Amd sev-snp: Strengthening vm isolation with integrity protection and more. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020.

[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, Daniel, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI 2016*.

[3] A. Awad and B. Karp. Execution integrity without implicit trust of system software. In *SysTEX 2019*.

[4] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI 2014*.

[5] C. che Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *USENIX ATC 2017*.

[6] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016, 2016.

[7] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security 2016*.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel. Telekine: Secure computing with cloud GPUs. In *NSDI 2020*.

[10] Intel Corporation. Software guidance for security advisories. https://software.intel.com/security-software-guidance/software-guidance, 2019.

[11] Intel Corporation. *10th Generation Intel® Core Processor™ Families*, 2020.

[12] Intel Corporation. *Supporting Intel SGX on Multi-Socket Platforms*, 2021.

[13] D. Lee, D. Kohlbrenner, K. Cheang, C. Rasmussen, K. Laeufer, I. Fang, A. Khosla, C.-C. Tsai, S. Seshia, D. Song, and K. Asanovic. Keystone enclave: An open-source secure enclave for RISC-V. https://keystone-enclave.org/files/keystone-risc-v-summit.pdf, 2018.

[14] Linux Kernel. Software guard extensions (sgx). https://www.kernel.org/doc/html/latest/x86/sgx.html#encryption-engines, 2022.

[15] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys 2012*.

[16] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: Exitless os services for sgx enclaves. In *EuroSys 2017*.

[17] M. Orenbach, Y. Michalevsky, C. Fetzer, and M. Silberstein. CoSMIX: A compiler-based system for secure memory instrumentation and execution in enclaves. In *USENIX ATC 2019*.

[18] C. Tan, L. Yu, J. B. Leners, and M. Walfish. The efficient server audit problem, deduplicated re-execution, and the web. In *SOSP 2017*.

[19] C. Tan, C. Zhao, S. Mu, and M. Walfish. Cobra: Making transactional key-value stores verifiably serializable. In *OSDI 2020*.

[20] 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 *USENIX Security 2018*.

[21] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on GPUs. In *OSDI 2018*.

[22] 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 *CCS 2017*.

[23] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. https://foreshadowattack.eu/foreshadow-NG.pdf, 2018.