Fast Fully Oblivious Compaction and Shuffling*

Sajin Sasy
University of Waterloo
Waterloo, ON, Canada
ssasy@uwaterloo.ca

Aaron Johnson
U.S. Naval Research Laboratory
Washington, D.C., U.S.A.
aaron.m.johnson@nrl.navy.mil

Ian Goldberg
University of Waterloo
Waterloo, ON, Canada
iang@uwaterloo.ca

ABSTRACT

Several privacy-preserving analytics frameworks have been proposed that use trusted execution environments (TEEs) like Intel SGX. Such frameworks often use compaction and shuffling as core primitives. However, due to advances in TEE side-channel attacks, these primitives, and the applications that use them, should be *fully oblivious*; that is, perform instruction sequences and memory accesses that do not depend on the secret inputs. Such obliviousness would eliminate the threat of leaking private information through memory or timing side channels, but achieving it naively can result in a significant performance cost.

In this work, we present fast, fully oblivious algorithms for compaction and shuffling. We implement and evaluate our designs to show that they are practical and outperform the state of the art. Our oblivious compaction algorithm, ORCompact, is always faster than the best alternative and can yield up to a 5× performance improvement. For oblivious shuffling, we provide two novel algorithms: ORShuffle and BORPStream. ORShuffle outperforms prior fully oblivious shuffles in all experiments, and it provides the largest speed increases—up to 1.8×—when shuffling a large number of small items. BORPStream outperforms all other algorithms when shuffling a large number of large items, with a speedup of up to 1.4× in such cases. It can obtain even larger performance improvements in application settings where the items to shuffle arrive incrementally over time, obtaining a speedup of as much as 4.2×. We additionally give parallel versions of all of our algorithms, prove that they have low parallel step complexity, and experimentally show a 5–6× speedup on an 8-core processor.

Finally, ours is the first work with the explicit goal of ensuring full obliviousness of complex functionalities down to the implementation level. To this end, we design Fully Oblivious Assembly Verifier (FOAV), a tool that verifies the binary has no secret-dependent conditional branches.

CCS CONCEPTS

• **Security and privacy → Domain-specific security and privacy architectures**.

KEYWORDS

oblivious algorithms, trusted execution environments, SGX

*An extended version of this paper is available. [49]

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560603

ACM Reference Format:
Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560603

1 INTRODUCTION

The recent advances and availability of trusted execution environments (TEEs) have provided an opportunity to deploy privacy-preserving data analytics with high utility. As long as the TEEs are correctly designed and implemented, they can guarantee to a remote party that a given set of instructions is correctly and privately executed by an *enclave* (secure container) on the local server.

For instance, Bittau et al. [9] design the Prochlo system to support large-scale monitoring of software activities such as telemetry, error reporting, or demographic profiling. Prochlo uses TEEs (namely Intel SGX [2]) to provide strong privacy guarantees to user data. Similarly, the Private Sampling-based Query Framework (PSQF) [50] uses TEEs to provide confidentiality and integrity guarantees to user data while performing differentially private queries based on sampling [7].

Both of these frameworks make key use of *shuffling*; that is, putting data items in a uniformly random order. A major component of Prochlo is the Stash Shuffle algorithm, which is designed to overcome performance limitations of SGX enclaves. Prochlo shuffles data to logically separate incoming data items from the identities of their contributors before the analysis is performed. In PSQF, shuffling is used for random sampling. The privacy guarantees of these shuffles rest heavily on their *obliviousness*; that is, the extent to which their observable actions, such as memory accesses, are independent of the data items and the random order they are put in. However, the shuffling algorithms used in these systems are *not* in fact oblivious with respect to state-of-the-art TEE adversaries [11, 30, 32, 38, 39]. Such adversaries can observe memory accesses in supposedly "protected" memory and at high granularity (e.g., within pages), which additionally allows them to observe conditional code branching behaviour.

Therefore, in this work, we design and evaluate novel oblivious shuffling algorithms that are *fully oblivious* in that all memory accesses and code branching are independent of the inputs and output order. While it is always possible to make an existing algorithm fully oblivious (e.g., by replacing memory accesses with linear scans and executing all possible code branches), doing so can come at a high performance cost. We present the Oblivious Recursive Shuffling algorithm, ORShuffle, which experimentally outperforms the classic (and still state-of-the-art, as explained by Asharov et al. [3, §1]) method for oblivious shuffling using random labels and a bitonic sorting network (Bitonic Shuffle). ORShuffle achieves the

greatest speedup when shuffling many small items. We also present the BORPStream algorithm, which provides experimental performance competitive to Bitonic Shuffle with notable performance improvements when shuffling many large items. BORPStream is based on the Bucket Oblivious Random Permutation (BORP) of Asharov et al. [3], but includes significant modifications to BORP in order to obtain obliviousness and efficiency. These changes further enable significant speed increases in the *incremental data* setting, where data items arrive sequentially over time. An example application in this setting is Prochlo, as the data are submitted by clients in an uncoordinated way, and to provide privacy enough items must accumulate before shuffling and analysis.

A key component of our shuffling algorithms is a novel oblivious algorithm for *compaction*; that is, the problem of permuting items in an array so that a "marked" subset of them appears at the beginning. Oblivious compaction is a generally useful tool for oblivious algorithms, with one common use being to filter out "dummy" items inserted earlier to mask conditional memory writes (e.g., in an ORAM [5] or an oblivious database query [29]). Several oblivious compaction algorithms have been given recently [5, 6, 18, 34], but these algorithms have high constants, and practical improvements have not been observed since Goodrich [22]. Our oblivious recursive compaction algorithm, ORCompact, improves performance in practice, which we demonstrate experimentally. Moreover, ORCompact is *order-preserving*; that is, it maintains the relative order among marked items. This property, also provided by Goodrich compaction, is useful when compaction is used as a subroutine and later steps require that marked items stay in the same order [29].

It is also important that our algorithms be parallelizable because many TEE systems, including Prochlo and PSQF, are designed to process large amounts of data. We therefore give efficient parallel versions of our shuffling and compaction algorithms, and we implement and test them. We note that our parallel compaction algorithm is the first practical such algorithm to be explicitly described, and a straightforward parallelization of Goodrich compaction adds significant overhead (see §7).

We summarize our contributions as follows:
1) We design and analyze the novel fully oblivious compaction algorithm ORCompact (§3). We analytically show it is efficient and give an efficient parallel version. Experiments show good practical performance, for example compacting 2^{23} items of 8 bytes each in under 200 ms. They also show a 2–5.2× performance improvement over the state of the art, Goodrich's tight compaction [22].
2) We design and analyze the fully oblivious shuffling algorithm ORShuffle (§4). ORShuffle internally uses ORCompact. Our analyses show that it is practically efficient and parallelizes well. Experimentally, it consistently outperforms prior fully oblivious shuffles, achieving the most improvement when shuffling a large number of small items. For example, ORShuffle shuffles 2^{24} 8-byte items in 9.56±0.05 s, while Bitonic Shuffle takes 16.78±0.03 s.
3) We design and analyze BORPStream. BORPStream internally uses both ORShuffle and ORCompact. Our experiments show that naively adapting BORP results in poor performance but that BORPStream yields competitive performance overall and significant improvements when shuffling a large number of large items. For example, BORPStream shuffles 2^{20} 4 KiB items in 371.9±0.8 s, while Bitonic Shuffle takes 523.9±0.2 s. Moreover, it can partially shuffle data items individually as they arrive, reducing the time needed to produce a shuffle after the last item is received. For example, BORPStream finishes a shuffle of 2^{20} 4 KiB items in just 124.7±0.1 s after the last item's arrival.
4) We design Fully Oblivious Assembly Verifier (FOAV), a tool that helps us ensure that the binaries produced after compilation are indeed fully oblivious (§6). FOAV tracks all the conditional jumps in the final binary and verifies that the operands of such jumps have been marked safe by our code instrumentation. All of our code is available at https://crysp.uwaterloo.ca/software/obliv/.

2 BACKGROUND

2.1 Trusted Execution Environments (TEEs)

Since their inception with Intel TXT [25] more than a decade ago, TEEs have undergone several iterations of refinements. Recent advances in hardware-aided TEEs such as Intel SGX [2], AMD SEV [28], and their open-source sibling Keystone [31] can provably execute programs securely in hardware-protected containers called *enclaves* that provide strong *confidentiality* and *integrity* guarantees, even if its host is malicious. Effectively they provide a small Trusted Computing Base (TCB) on otherwise malicious servers. For ease of exposition, we limit the discussion of how TEEs provide these guarantees by just detailing how it is achieved in Intel SGX. Intel SGX is the most mature TEE among the choices available today, and we use it to implement and benchmark our algorithms. However, the techniques used by other TEEs are similar to those of SGX, making our work applicable to them as well.

In Intel SGX, the enclaves provide trusted execution environments by bootstrapping security from cryptographic keys that are fused into the processor at manufacture time [2]. At boot time such processors set aside a portion of their available DRAM as Processor Reserved Memory (PRM). All accesses to memory pages in the PRM occur through Intel's Memory Encryption Engine (MEE), which uses the processor-fused keys to encrypt these pages every time they leave the system cache and return to the PRM. Similarly MEE decrypts and integrity checks all pages loaded from the PRM before they are moved to the cache for use by the SGX-supported processor; thus the contents of these pages are always protected.

The SGX threat model allows an adversary to control the software stack of an SGX-supported server, including in particular the OS. The adversary may have physical access to server but is assumed to be incapable of physically tampering with the processor chip, extracting the processor-fused keys, or snooping the state of the processor registers, which would void the purported security guarantees. Nonetheless, the adversary is formidable and can observe all the shared resources and peripherals.

Researchers have identified several attacks on SGX [11, 30, 32, 38, 39, 53, 57, 58]. Most of these attacks break confidentiality by exploiting side channels or metadata that arise from secret-dependent memory accesses or control-flow branching. These attacks motivate designing the trusted code to be data-oblivious as a defense [43, 48].

2.2 Degrees of Obliviousness

Motivated by attacks on TEEs, we identify three independent ways in which an enclave program can be oblivious to secret data.

1) **External-memory oblivious**: Memory accesses to data outside of the PRM are independent of any secret data.
2) **Protected-memory oblivious**: Memory accesses to data within the PRM are independent of any secret data. The literature has provided two finer granularities of attack surfaces on protected memory access patterns, namely:

i) **Page oblivious**: Several proposed attacks leverage the fact that the untrusted OS is still responsible for page table management of enclave programs. These attacks induce page faults on enclave programs to extract the memory locations accessed by the program. However, SGX masks the last 12 bits of a page fault address to the untrusted OS, thus limiting the immediate memory access pattern leakage to a malicious OS to page-level granularity (4 KiB) [53, 57].

ii) **Cacheline oblivious**: Advanced attacks [11, 30, 38] can extract the precise address of the (64 B) cacheline loaded during a page fault.

iii) **Subcacheline oblivious**: There have been no side-channel attacks at a finer granularity than cache level. Nonetheless it is conceivable that future research might discover attacks that can extract the exact (8 B) word loaded from a cacheline to a register in a memory fetch operation.

3) **Control-Flow oblivious**: Orthogonal to leakages from data accesses induced by the execution of the trusted program, another form of leakage arises with secret-dependent control flow branches [32, 39]. These attacks can extract the exact branch chosen in a secret-dependent control flow branch, thus leaking information about the secret. We say a program is control-flow oblivious if it has no control branches in its execution dependent on secret data.

We say that a program is *fully oblivious* if it satisfies all the above definitions of obliviousness (see the extended version of this paper [49, App.A] for a formal definition). While previous work using TEEs [9, 29, 36, 45, 59] has satisfied a few of these definitions, none has targeted all of them. By providing full obliviousness, we defend against all attacks that depend on memory or control-flow side channels [11, 30, 32, 38, 39, 53, 57, 58]. Note that we do not defend against hardware side channel attacks [12, 41, 56]. We provide more details on TEE attacks in Section 2.3.

An obstacle to obtaining full obliviousness is that even if source code is control-flow oblivious, compilation may not preserve that obliviousness. Compilers are notorious for optimizing away programming logic that was intended to provide security guarantees like constant-time code, or side-channel resilient memory access patterns [54]. To address this, we design FOAV, a tool that helps verify the control-flow obliviousness of the program *binary*. We describe FOAV in Section 6.4.

Without fully oblivious algorithms a TEE adversary, in the worst case, can distinguish the locations of real blocks in the array from those of dummies in the compaction setting, and can infer the final permutation of a shuffle. These algorithms are typically used as building blocks for applications like Prochlo to provide an anonymity set to users' data by shuffling them before analysis, or for providing strong differential privacy guarantees for data in PSQF. Leakages in the underlying oblivious shuffle hence either deanonymize users or provide significantly weaker privacy properties than claimed. Ours is the first work that designs compaction and shuffling algorithms with the explicit goal of being fully oblivious, and the first to provide a fully oblivious implementation.

2.3 TEE Attacks

The taxonomy of obliviousness we present above is informed by various recent side-channel attacks on programs running in TEEs.
Page-granular attacks: Xu et al. [58] presented controlled-channel attacks, which allow a malicious OS to infer sensitive data when the victim program has input-dependent control or data transfer. Similarly, Shinde et al. [53] demonstrate that this page fault side channel can be used to extract encryption keys from implementations of cryptographic routines. This led to defences that suppress page faults during enclave execution [52, 53]. However, a malicious OS observing side effects of the address translation process can still extract this leakage without relying on page faults explicitly [57].
Cacheline-granular attacks: PRIME+PROBE [35, 44] and other cache attacks have been demonstrated outside of the SGX setting for more than a decade now. Several works [11, 23, 38] have demonstrated that such attacks in fact translate well to the SGX setting. However, these type of attacks are typically quite noisy and lossy, from cache activities of other processes on the same system and the probing process having non-negligible probability of missing the victim's access patterns. Hence they require several measurements of the victim program executions to reliably extract cache access patterns. Additionally, they require forcing interruptions of enclave executions to precise the cache access patterns. This led to defense mechanisms such as T-SGX [52] and Déjà vu [13] that detect frequent interrupts as an indicator of the enclave program being under attack. More recently, Lee et al. demonstrated Membuster [30], an off-chip physical attack that snoops the memory addresses accessed on the memory bus, which in conjunction with novel cache manipulation tricks that force cache misses allows them to extract cache usage patterns from the victim process with minimal overheads and without inducing interrupts.
Control Flow Attacks: While several cache attacks can be used to attack the control flow of programs executing in an enclave, Lee et al. [32] demonstrated a precise technique of tracing control flow execution by leveraging the branch prediction history, which was not cleared while switching out of enclave mode. Moghimi et al. [39] demonstrated a control flow attack, which precisely counts the number of instructions executed within an enclave execution, which when combined with the coarse-grained page-level leakages can reconstruct the exact control flow taken by the enclave program.

Such side-channel attacks can be thwarted by designing the enclave program to be fully oblivious, as it eliminates these vulnerabilities at their introduction point. As more powerful attacks that amplify a TEE adversary's ability to violate the privacy guarantees of programs running in an enclave become efficient and practical, it becomes paramount to ensure that security and privacy critical components of such programs be implemented in a fully oblivious fashion. We note that obliviousness, including fully obliviousness, only protects against software side channels. However, previous work has shown microachitectural side channels like speculative execution [12, 56] or the voltage scaling interface [41]. Attacks such as these that violate the SGX, and indeed the processor, security model, are mitigated with microcode patches from Intel [26, 27]. Importantly, the kinds of memory-access side channels described above are *not* considered to violate the SGX security model—SGX

makes no claim to protect the addresses of accessed memory. Therefore, these side channels must be addressed in software, using the kinds of fully oblivious algorithms we propose.

3 RECURSIVE COMPACTION

We introduce the fully oblivious recursive compaction algorithm ORCompact. Compaction is a fundamental primitive for ORAMs and TEE systems, and we will also use it for oblivious shuffling (see §4). ORCompact takes as input an array D of n data items and a bit array M of length n. The positions of M with a 1 value indicate "marked" items, and the positions with a 0 value indicate "unmarked" items. ORCompact rearranges the items in D such that all marked items appear before all unmarked items. Furthermore, as we will show, ORCompact is order-preserving; that is, it maintains the relative order of the marked items. Note that the relative order of the unmarked items may change, however.

This algorithm has time complexity $O(n \log n)$, which is the same as the algorithm of Goodrich [22] and which is higher asymptotic complexity than the linear-time optimum [5, 18]. However, unlike the asymptotically optimal algorithms, ORCompact is practically efficient for realistic values of n, and it is faster than the Goodrich algorithm by a factor of at least 2. It is also more efficiently parallelizable. See Section 7 for more details on these other algorithms.

3.1 Algorithm

We use $D_{j..k}$ to indicate the subarray $(D_j, \ldots, D_k)$. We denote the oblivious computation of the comparison c as a boolean value with $[c]$ (e.g., $[x < y]$ has value 1 if $x < y$ and 0 otherwise). OSwap(D_i, D_j, b) refers to an oblivious swap subroutine that swaps the ith and jth values of D if $b = 1$ and leaves D unchanged if $b = 0$. It can be efficiently implemented obliviously (cf. Section 3.3). Arithmetic operations must also be implemented in an oblivious way.

The core compaction logic is contained in the subroutine OROffCompact(D, M, z) (see Figure 1). OROffCompact compacts marked items to a region starting at an offset position z, with possible wraparound. However, it can only be called on an array D with a length that is a power of two because it divides D into halves to perform pairwise OSwaps. OROffCompact is fully oblivious to its input values (though not their sizes) including in particular the offset z. Because $|D|$ is a power of two, simple bitwise operators can obliviously implement the division and modulo operations.

The full ORCompact(D, M) algorithm is given in Figure 2. Intuitively, ORCompact divides D into a right side, which is the largest suffix with size n_1 a power of two, and a left side, which is the remainder of size $n_2 = n - n_1$. It then recursively compacts the n_2 items on the left and also computes the number of selected items m on that side. Finally, it compacts the right side with an offset that is $n_2 - m$ items from the end, and the final n_2 items on the right are conditionally swapped pairwise with the n_2 items on the left. OROffCompact works similarly, although it also takes into account that there may be an offset for the desired compaction. Because it can assume its input is a power of two, items on the left side and right side can be paired for potential swaps, which is why it can produce an offset in any position obliviously.

Figure 1: OROffCompact(D, M, z): Compact items in D, as marked in M, to an offset z. D and M must have a power-of-two length.

$n \leftarrow |D|$, $m \leftarrow \sum_{i=0}^{\frac{n}{2}-1} M_i$
if $n = 2$
  OSwap$(D_0, D_1, ((1 - M_0)M_1) \oplus z)$
else if $n > 2$
  OROffCompact$\left(D_{0..\frac{n}{2}-1}, M_{0..\frac{n}{2}-1}, z \bmod \frac{n}{2}\right)$
  OROffCompact$\left(D_{\frac{n}{2}..n-1}, M_{\frac{n}{2}..n-1}, z + m \bmod \frac{n}{2}\right)$
  $s \leftarrow \left[\left((z \bmod \frac{n}{2}) + m\right) \geq \frac{n}{2}\right] \oplus \left[z \geq \frac{n}{2}\right]$
  for $i \leftarrow 0 \ldots \frac{n}{2} - 1$
    $b \leftarrow s \oplus \left[i \geq \left((z + m) \bmod \frac{n}{2}\right)\right]$
    OSwap$(D_i, D_{i+\frac{n}{2}}, b)$

Figure 2: ORCompact(D, M): Compact items in D as marked in M. D and M need not have a power-of-two length.

if $|D| = 0$ **return**
$n \leftarrow |D|$, $n_1 \leftarrow 2^{\lfloor \log_2(n) \rfloor}$, $n_2 \leftarrow n - n_1$, $m \leftarrow \sum_{i=0}^{n_2-1} M_i$
ORCompact$(D_{0..n_2-1}, M_{0..n_2-1})$
OROffCompact$(D_{n_2..n-1}, M_{n_2..n-1}, n_1 - n_2 + m)$
for $i \leftarrow 0 \ldots n_2 - 1$ **do**
  $b \leftarrow [i > m]$
  OSwap(D_i, D_{i+n_1}, b)

3.2 Correctness

Theorem 1 shows that ORCompact compacts the marked items and maintains their relative order. Its proof appears in the extended version of this paper [49, App.B].

Theorem 1. *Given D and M, let $(D'_0, \ldots, D'_{w-1})$ be the subsequence of D consisting of all w items marked in M. ORCompact(D, M) rearranges the items in D such that $(D_0, \ldots, D_{w-1}) = (D'_0, \ldots, D'_{w-1})$.*

3.3 Obliviousness

We now explain how ORCompact is fully oblivious (cf. Section 2.2), assuming an implementation that follows Figure 2 in the straightforward way. The result is fully oblivious with respect to the values of its input arrays, although not to their size or location in memory. In fact, not only are all instructions and memory accesses independent of the input array values, as required for full obliviousness, they are *deterministic* given the size and location of those arrays. See the extended version of this paper [49, App.A] for a formal statement and proof of this obliviousness.

Prior work [14, 43] has shown how to efficiently implement comparisons (e.g., $[i < m]$) and swaps (OSwap) fully oblivously. These implementations are deterministic and either use conditional instructions such as `SETB` and `CMOVZ` or Boolean arithmetic. The comparisons are oblivious to the compared values but not to the comparison operator. OSwap is oblivious to the input values and to the swap flag b. It is not oblivious to locations of the inputs.

We next argue that OROffCompact is fully oblivious. We can see that the only conditional branching (i.e., the `if`/`else` and `for`) is

based on the length n of the input arrays and not on their contents or on the offset z. Thus, the instruction sequence is deterministic given the input size. We then observe that all accesses to the inputs are deterministic given n. As previously argued, the computation of comparisons (e.g., $[x < y]$) is fully and deterministically oblivious to the values of its inputs. Similarly, the calls to OSwap are oblivious to its last argument, and we observe that the other arguments are deterministic locations given n. We can recursively assume that the calls to OROffCompact are fully and deterministically oblivious. Therefore, OROffCompact is fully oblivious, and its instruction sequence and memory accesses are deterministic given the length of its input arrays D and M.

Finally, we argue that ORCompact (Figure 2) is fully oblivious. We observe that its only conditional execution is a `for` loop (Line 5) that executes a number of times that depends deterministically on n. We can also see that its variable accesses are all a deterministic function of n. By our previous arguments, the comparisons and OROffCompact call are fully oblivious. Similarly, each OSwap call is fully oblivious to its last argument, and the other arguments are deterministic functions of n. Finally, we can recursively assume that the ORCompact call is fully oblivious. Therefore, ORCompact is fully oblivious, and its instruction sequence and memory accesses are deterministic given the input size.

3.4 Efficiency

We analyze the time and space efficiency of ORCompact as the number and size of data items grow. Theorem 2 gives the asymptotic runtime of ORCompact (see the extended version of this paper [49, App.B] for a proof). As previously noted, this runtime is not the asymptotic optimal linear time, but, as our experiments results will show, the constants are low enough that it outperforms all other oblivious algorithms for realistic input sizes.

Theorem 2. *ORCompact runs in time $O(n \log n)$.*

We next consider the number of oblivious swaps (i.e., calls to OSwap) because the data items themselves are only accessed during swaps. Each swap takes time linear in the item size, and so, for large enough items, oblivious swaps will constitute an arbitrarily large fraction of the runtime. We count the number of oblivious swaps, which admits a concrete (rather than purely asymptotic) estimate, thereby enabling a direct comparison to other swap-based algorithms. Let n be the length of the input arrays to ORCompact. Theorem 3 provides upper and lower bounds on the exact number of OSwap calls in ORCompact. Theorem 3 is proved in the extended version of this paper [49, App.B].

Theorem 3. *Let $S_1(n)$ count the OSwap calls in ORCompact. Then*

$$\left(2^{\lfloor \log_2 n \rfloor}/2\right) \lfloor \log_2 n \rfloor \le S_1(n) \le \lfloor (n/2) \log_2 n \rfloor.$$

Theorem 3 shows that ORCompact performs approximately $(n/2) \log_2 n$ OSwaps. We can use this estimate to compare performance to the tight oblivious compaction algorithm of Goodrich [22], which appears to date to be the fastest algorithm in practice [29]. This algorithm can be adapted to perform compaction fully obliviously using OSwaps (the original algorithm description assumes some private memory and does not require unmarked items to be output). However, it uses more than $(\log_2 n - 2)n$ OSwaps, which suggests a runtime of about 2× that of ORCompact when item sizes are large enough for swaps to constitute most of the runtime (for small item sizes, we see in Section 6 that ORCompact outperforms Goodrich by an even larger factor).

We note that ORCompact performs *in-place* compaction and thus need not maintain extra copies of possibly large data items. Moreover, the additional memory requirements are very small, with a small constant amount of memory needed at each of the $\lceil \log_2 n \rceil$ levels of recursion. We also observe that the memory-access patterns are highly regular (linear scans) and generally with high locality (especially at the deeper levels of recursion), which yields good practical performance due to caching and prefetching.

3.5 Parallelization

We describe how ORCompact can be parallelized by giving a version of it in the EREW PRAM model [40]. In this model, multiple processors proceed in synchronized steps, and they access shared memory with exclusive read and write (i.e., only one processor can read or write a given memory location in a given step). OROCPar, given in Figure 3, performs parallel oblivious compaction to an offset when n is a power of two, and ORCPar, given in Figure 4, performs parallel oblivious compaction for any n. We indicate parallel computations with a **do in parallel** block, in which each line can be computed in parallel, and a **parallel for** block, in which each loop iteration can be computed in parallel. This presentation omits explicit assignment to processors. For our algorithms, we consider the *work* (i.e., total computations across all processors) and the number of parallel *steps* (i.e., longest sequence of operations that any single processor might execute).

ORCPar uses the subroutine PrefixSumPar to compute the array S containing the sums of prefixes of the boolean array M. S is used by both ORCPar and OROCPar to compute offsets in constant time. Hillis and Steele [24] give an algorithm to compute the prefix sums obliviously in $\lceil \log_2 n \rceil$ steps and $O(n \log n)$ total work. Note that ORCPar takes S as an extra argument, which should initially be empty, and that S is produced with a length of $n + 1$, where each element is the sum of the preceding positions in M (with $S_0 = 0$). S can thereby be used recursively to compute subsequence sums.

We observe that the parallelization does not significantly affect the work. As with ORCompact, the work in ORCPar is $O(n \log n)$. In addition, the total number of the OSwap calls remains the same as in ORCompact (quantified in Theorem 3). At the same time, parallelization significantly improves step complexity, with $O(\log n)$ parallel steps and $\lceil \log_2 n \rceil$ steps involving OSwap calls.

4 RECURSIVE SHUFFLING

We now present the fast fully oblivious recursive shuffling algorithm ORShuffle, which uses ORCompact as a key subroutine. It takes as input an array D of n items and rearranges the items of D so that they appear in a uniformly random order.

This algorithm has time complexity $O(n \log^2 n)$. This complexity is the same as some efficient sorting networks [8], which can themselves be used for shuffling by attaching random labels to items before sorting based on those labels. However, ORShuffle represents an efficiency improvement because it does not need to attach labels and move them around with the data. The complexity

Figure 3: OROCPar(D, S, z): Parallel compaction of D to offset z. S is prefix sums of marked items. D must be a power-of-two length.

1: $n \leftarrow |D|$, $m \leftarrow S_{\frac{n}{2}} - S_0$
2: **if** $n = 2$ **then**
3: OSwap$(D_0, D_1, ((1 - (S_1 - S_0))(S_2 - S_1)) \oplus z)$
4: **else if** $n > 2$ **then**
5: **do in parallel**
6: OROCPar$\left(D_{0..\frac{n}{2}-1}, S_{0..\frac{n}{2}}, z \bmod \frac{n}{2}\right)$
7: OROCPar$\left(D_{\frac{n}{2}..n-1}, S_{\frac{n}{2}..n}, z + m \bmod \frac{n}{2}\right)$
8: $s \leftarrow [\left((z \bmod \frac{n}{2}) + m\right) \geq \frac{n}{2}] \oplus [z \geq \frac{n}{2}]$
9: **parallel for** $i \leftarrow 0 \ldots \frac{n}{2} - 1$
10: $b \leftarrow s \oplus [i \geq (z + m) \bmod \frac{n}{2}]$
11: OSwap$(D_i, D_{i+\frac{n}{2}}, b)$

Figure 4: ORCPar(D, M, S): Parallel compaction of D as marked in M. D and M need not be power-of-two lengths. Pass empty S in initial call.

1: **if** $|D| = 0$ **then return**
2: **if** $|S| = 0$ **then** $S \leftarrow$ PrefixSumPar(M)
3: $n \leftarrow |D|$, $n_1 \leftarrow 2^{\lfloor \log_2(n) \rfloor}$, $n_2 \leftarrow n - n_1$
4: **do in parallel**
5: ORCPar$(D_{0..n_2-1}, \emptyset, S_{0..n_2})$
6: OROCPar$(D_{n_2..n-1}, S_{n_2..n}, n_1 - n_2 + S_{n_2} \bmod n_1)$
7: **parallel for** $i \leftarrow 0 \ldots n_2 - 1$
8: $b \leftarrow [i > S_{n_2}]$
9: OSwap(D_i, D_{i+n_1}, b)

is also higher than the $O(n \log n)$ complexity achieved by other approaches [5]. The constants are low, though, and we will show that the performance of ORShuffle exceeds that of other approaches.

4.1 Algorithm

ORShuffle follows the high-level approach of the PerfectORP algorithm of Asharov et al. [4, Algorithm 6.8]. In that algorithm, the array of items to shuffle is split into two halves, each is recursively shuffled, and then the items in each half are randomly interleaved with an Intersperse algorithm. Intersperse effectively performs a "reverse" compaction, sometimes called "expansion" [22]. We can therefore use ORCompact directly by reversing these steps (alternatively, running ORCompact in reverse itself would yield an expansion algorithm). That is, we randomly separate the items into two halves using ORCompact and then recursively shuffle each half. The result is the fully oblivious shuffle algorithm ORShuffle (see the extended version of this paper [49, App.A] for a formal statement and proof of obliviousness). Unlike PerfectORP, ORShuffle is a practically efficient algorithm because of its use of ORCompact in place of Intersperse (see §7.1).

The ORShuffle algorithm is given in Figure 5. It uses the subroutine MarkHalf(n) (Figure 6) to mark a random half of the n positions, which is also essentially given by Asharov et al. [4, Algorithm 6.1]. ORShuffle rearranges the items in D into a uniformly random order. It is fully oblivious to the values in D, and it is also fully oblivious to its random bits and therefore to the permutation it applies. ORShuffle is not oblivious to the size or location of D. Its correctness and obliviousness follow from the correctness and obliviousness of ORCompact, using arguments similar to those of Asharov et al. [5] regarding PerfectORP.

Figure 5: ORShuffle(D): Put items in D in uniformly random order.

1: $n \leftarrow |D|$
2: **if** $n = 2$ **then**
3: Generate $b \in \{0, 1\}$ uniformly at random.
4: OSwap(D_0, D_1, b)
5: **else if** $n > 2$ **then**
6: $M \leftarrow$ MarkHalf(n)
7: ORCompact(D, M)
8: ORShuffle$(D_{0..\lceil n/2 \rceil - 1})$
9: ORShuffle$(D_{\lceil n/2 \rceil..n-1})$

Figure 6: MarkHalf(n): Mark random half of boolean array of length n.

1: Create boolean array M of length n.
2: $\ell \leftarrow \lceil n/2 \rceil$
3: **for** $i \leftarrow 0 \ldots n - 1$ **do**
4: Generate $r \in [0, 1)$ uniformly at random.
5: $M_i \leftarrow [r < \ell/(n - i)]$
6: $\ell \leftarrow \ell - M_i$
7: **return** M

4.2 Efficiency

The asymptotic runtime of ORShuffle is given in Theorem 4. A proof of it appears in the extended version of this paper [49, App.B].

Theorem 4. *ORShuffle runs in time $O(n \log^2 n)$.*

While this runtime is not asymptotically optimal (oblivious shuffling algorithms exist with runtime $O(n \log n)$), the constants make it practically efficient. Theorem 5 gives an exact count of the number of OSwap calls. These oblivious swaps are relatively expensive operations, and, as with ORCompact, the data items themselves are only accessed via these swaps, and so the swaps dominate the runtime for large enough items. Theorem 5 is proved in the extended version of this paper [49, App.B].

Theorem 5. *Let $S_2(n)$ count the number of OSwap calls performed by ORShuffle(D), with $|D| = n$. For $n = 2^k, k \in \mathbb{N}$, $S_2(n) = (n/4)(\log_2 n + 1) \log_2 n$. For all n, $S_2(n) < (n/4)(\log_2 n + 1) \log_2 n + n/(6 \ln 2)$, and $S_2(n) \geq \left(2^{\lfloor \log_2 n \rfloor}/4\right)(\lfloor \log_2 n \rfloor + 1)\lfloor \log_2 n \rfloor$.*

This shows that ORShuffle performs roughly $(n/4)((\log_2 n) + 1) \log_2 n$ oblivious swaps. We can compare this to the oblivious swaps performed using a sorting network. A sorting network performs pairwise compare-and-swap operations in a fixed sequence and thus can be used for oblivious shuffling by attaching random

Figure 7: ORSPar(D): In parallel, put items in D in uniformly random order.

```
1: n ← |D|
2: if n = 2 then
3:    Generate b ∈ {0,1} uniformly at random.
4:    OSwap(D_0, D_1, b)
5: else if n > 2 then
6:    M ← MarkPar(⌈n/2⌉, n)
7:    ORCPar(D, M, ∅)
8:    do in parallel
9:       ORSPar(D_{0..⌈n/2⌉−1})
10:      ORSPar(D_{⌈n/2⌉..n−1})
```

labels to items. Using the bitonic sorting network [8] is the common choice for existing TEE systems that require oblivious sorting [1, 20, 29, 59]. The odd-even mergesort network [8] is slightly smaller but experimentally performs worse due to poor memory locality. The bitonic sorting network also requires $(n/4)((\log_2 n) + 1)\log_2 n$ oblivious swaps. Therefore, for large data items, we can expect that ORShuffle will perform similarly to Bitonic Shuffle. However, for smaller items, ORShuffle can yield an efficiency improvement because random labels need not be added and included in the oblivious swaps. Indeed, for word-sized items, random labels can approximately double the cost of the swaps.

The memory used by ORShuffle is little more than the memory already occupied by the data items. The data items are swapped in place. At most an additional n bits are created to mark items for compaction, and a small constant amount of memory is needed for each of the $\lceil \log_2 n \rceil$ levels of recursion. We also note that the algorithm can benefit significantly from memory caching and prefetching due to its regular memory-access patterns and good locality.

4.3 Parallelization

We give a parallel version of ORShuffle in Figure 7, again in the EREW PRAM model. This algorithm, ORSPar, is nearly identical to the sequential version except that it calls parallel versions of the subroutines, namely MarkPar and ORCPar, and makes its recursive calls in parallel.

MarkPar, given in Figure 8, marks a random m of n locations. This algorithm is a variant of the algorithm given by Sanders et al. [47, Algorithm P]. The differences from the original algorithm are that it is fully specified (i.e., no unspecified "local sampler") and outputs a full boolean array of length n (i.e., not only the marked index values). These changes are needed to guarantee full obliviousness with respect to m and to the random bits.

MarkPar uses HyperGeomPar for hypergeometric sampling. HyperGeomPar(n, d, m) samples the number of successes among d random draws without replacement from a set of n items that contain m total successes. Following Sanders et al. [47], we perform hypergeometric sampling with an algorithm of Stadlober [55]. We use an oblivious implementation that uses the HRUE^t variant, computes $\ln(x!)$ with a Stirling approximation, omits the fast-acceptance optimization, and repeats the rejection sampling a constant number of times to obtain a given failure probability.

Figure 8: MarkPar(m, n): In parallel, mark a random m of n positions.

```
1: if n = 1 then
2:    return m
3: else if n > 1
4:    n_1 ← ⌈n/2⌉, n_2 ← n − n_1
5:    k ← HyperGeomPar(n, n_1, m)
6:    do in parallel
7:       M_1 ← MarkPar(k, n_1)
8:       M_2 ← MarkPar(m − k, n_2)
9:    return M_1||M_2
```

The work of ORSPar is similar to that of ORShuffle. The only difference in the operations executed is from calling ORCPar instead of ORCompact and MarkPar instead of MarkHalf. As discussed in Section 3, ORCPar has the same $O(n \log n)$ complexity as ORCompact and the same number of OSwap calls. MarkPar has the same $O(n)$ complexity as MarkHalf given a fixed failure probability. Therefore, ORSPar has the same $O(n \log^2 n)$ asymptotic runtime as ORShuffle. Moreover, the number of OSwap calls is the same in both.

Parallelization does improve the step complexity of shuffling. ORSPar takes $O(\log^2 n)$ parallel steps, and it has $(\lceil \log_2 n \rceil + 1)\lceil \log_2(n) \rceil/2$ parallel steps that include an OSwap call. These results show that ORSPar is appropriate for large-scale private data analysis, where many processors might be available for fast processing.

5 BUCKET OBLIVIOUS RANDOM PERMUTATION (BORP)

We next introduce the BORPStream shuffling algorithm, which is inspired by the Bucket Oblivious Random Permutation (BORP) of Asharov et al. [3]; BORPStream leverages entirely different machinery than BORP to efficiently shuffle while attaining full obliviousness, and reuses just the underlying butterfly routing network (BRN) used by BORP. BORP is a promising alternative to ORShuffle because of its small constants, and indeed its authors present it as a faster alternative to Bitonic Shuffle. However, it is given in a client-server setting where the client has private memory. Oblivious sorting or compaction are suggested as ways to avoid needing private memory, but using them increases the complexity in n, and, as our experiments will show, this approach yields poor shuffling performance (despite using the new and faster ORCompact).

At a high level, Asharov et al. [3] randomly permute items by passing them through a well-parameterised BRN. First the algorithm partitions the n input items across $\mathsf{B} = \frac{2n}{\mathsf{Z}}$ buckets, with each bucket containing Z elements. The constant Z dictates the correctness guarantee of BORP, and the authors recommend Z=512. These initial-layer buckets contain exactly half real items and half dummy items. Real items get assigned a random destination from among the B output buckets, while the dummies are assigned $\perp$ as their destination. The butterfly routing network of depth $\log_2 \mathsf{B}$ is then used to route all the real items to their destination buckets obliviously. In order to do so, the algorithm performs a *MergeSplit* operation on the $\mathsf{B}/2$ pairs of buckets that are 2^i apart in layer i. The MergeSplit operation distributes the real items of both its input

Figure 9: MSN: **MergeSplitNode class**

```
1:  Internal State Variables: c, buffer[], out_streams[]
2:  Initialize(f, s, b, streams):
3:      initializeBuffer(buffer, s, b)
4:      c ← 0 // c is the current eviction stream ∈ [0, f − 1]
5:      out_streams ← streams // MSNs this node connects to; or
        final buckets if this is a last layer MSN
6:  ProcessItem(p):
7:      evict_stream ← extractStream(p) // For dummy items ex-
        tractStream returns ⊥.
8:      for i ← 0 . . . s − 1 do
9:          sf_1 ← [extractStream(buffer[i])=c]
10:         sf_2 ← [isReal(p) ∧ evict_stream≠ c]
11:         swap_flag ← sf_1 ∨ sf_2
12:         OSwap(p, buffer[i], swap_flag)
13:     if self.isLastLayerNode then
14:         Append(out_streams[c], p))
15:     else
16:         out_streams[c].ProcessItem(p)
17:     c ← (c + 1) mod f
```

buckets into two output buckets based on the ith bit of their destination label, and pads each output bucket with enough dummy items to make exactly Z in each output bucket. These output buckets are then locally shuffled by the client to randomize the order of items before returning them to the server. Once the routing is completed, the dummy items from each output bucket are discarded to obtain the randomly permuted real items.

Executing BORP naively within a TEE would break its security guarantees. During the MergeSplit operation or the final discard of dummies, an adversary could use observations of the memory access patterns to distinguish real and dummy items. The straightforward solution is to replace the private-memory computations with oblivious variants (largely by changing MergeSplit to use ORCompact). We call this algorithm BORPCompact, which we present in Section 5.1. It is unfortunately not competitive with the state of the art, as we will see in Section 6. We therefore design BORPStream, which operates in a very different way: by pushing items through the BORP BRN one by one. Not only does this yield much better efficiency overall, it allows much of the work to happen while items are being input in a *streaming* manner.

5.1 BORPCompact

BORPCompact is almost identical to the original BORP. The difference is that the MergeSplit operation is instantiated with our ORCompact(D, M). To produce the correct bit array M used by ORCompact, the algorithm makes two linear passes over D (the concatentated two input buckets going into the ORCompact operation, $|D| = 2Z$). In the first linear pass it obliviously counts the number of real items going into each of the output buckets. In the second linear pass it marks real items correctly in M, while additionally also marking the right number of dummies by using the counts from the first pass, such that the total number of marked

Figure 10: BORPStream(n, b, λ) where n is the number of elements to shuffle, b is the size of each element, and λ is the failure probability parameter.

```
1:  Initialize(n, b, λ):
2:      f, d, s ← BORP_Optimizer(n, b, λ)
3:      B ← f^d // Number of buckets at start/end of BRN
4:      m ← f^(d−1) // Number of MSNs in each layer of BRN
5:      c ← 0 // Number of data elements processed so far
6:      D′ ← [] // Output array to be populated
7:      BRN ← InitializeBRN(f, d, s, b) // InitializeBRN sets up the
        BRN of MSNs defined by the parameters f, d, s, and b, returning
        a 2D array of MSN nodes, BRN: MSN[d][f^(d−1)]
8:
9:  BORPStream_Phase1(p): // for each arriving packet p
10:     entry_node ← ⌊c/B⌋
11:     Generate label ∈ [0, B − 1] uniformly at random
12:     p′ ← ⟨label,p⟩
13:     BRN[0][entry_node].ProcessItem(p′)
14:     BRN[0][entry_node].ProcessItem(dummy)
15:     c ← c + 1
16:     if c = n then
17:         BORPStream_Phase2(BRN)
18:
19: BORPStream_Phase2(BRN):
20:     FlushBuffers(BRN)
21:     for i ∈ [0, B − 1] do
22:         M,r ← MarkReal(B[i]) // M is a bit array; r ← ∑ M_i; B
        is the array of B buckets at the end of the BRN.
23:         ORCompact(B[i], M)
24:         ORShuffle(B[i]_{0..r−1})
25:         Append(D′, B[i]_{0..r−1})
26: return D′
```

items will be exactly Z. Once all the items have been processed through the BRN as described above, the final output buckets are compacted to just the real items in each of them by leveraging ORCompact again. Furthermore, the real items in each bucket are then obliviously shuffled using ORShuffle, to break any correlations in relative ordering of items within an output bucket. Finally all these items are concatenated together to produce the required output array of randomly permuted items. The correctness argument for this variant is identical to that of BORP [3], but our variant is fully oblivious and can by design be safely run in a TEE.

5.2 BORPStream

BORPStream converts BORP into an algorithm processing "streaming" data that arrives incrementally. At a high level, instead of processing items layer by layer at a bucket-level granularity, we stream each input item as it arrives through the entire butterfly routing network. Each of the discrete MergeSplit operations are replaced with a stateful processing node we call a *MergeSplitNode (MSN)*, described in Figure 9. Every MSN in a layer of the network is connected directly to the appropriate MSN in the next layer that their logical output bucket would have been connected to, effectively

building a butterfly routing network of purely MSNs, followed by a layer of B output buckets. To uphold obliviousness in this streamed model, items have to be processed in a deterministic fashion; i.e., routing decisions cannot be based on the destination of the item being routed. Therefore, all MSNs follow a strict round robin schedule for routing data across all of its f output streams.

The schedule is clearly at odds with the requirement of routing items to their correct destination. To ensure correctness, each MSN is imbued with a local buffer of capacity s items. When an MSN receives an item to process it scans across its local buffer and swaps the incoming item with one destined for the current eviction stream using oblivious swaps (Figure 9, lines 7–12). Figure 10 details BORPStream in its entirety. BORPStream can be split into two phases: i) Phase 1 processes input data packets as they arrive, assigning each element a random destination label $\in [0, \mathsf{B}-1]$ and then routing it through the BRN (Lines 9–15); ii) Phase 2 starts once all input data have been received and processed through the BRN.

In scenarios where data items are acquired over time, this two-phase division takes advantage of time otherwise spent waiting for all items to arrive. Several real world use-cases provide such scenarios, including application telemetry, software profiling, or error reporting. Typically such data accumulate over time as they are generated by user activity, and shuffling is performed only after enough users have provided inputs to yield a sufficiently large anonymity set. Later in Section 6.2 we will illustrate the concrete performance improvements obtained by this division by measuring the time taken by Phase 2, which reflects the turnaround time for the shuffle once all data items have arrived.

Given the number of items n to shuffle, the size of each item b, and the desired correctness parameter λ, we design and leverage an optimizer that returns the optimal choice of parameters (f: the fan of the network, d: the depth of the network, and s: the internal buffer size of each MSN) for the underlying BRN. The failure probability of the returned set of parameters will be smaller than $2^{-\lambda}$. In our experiments we set $\lambda = 80$. Once the optimal parameters are known, the enclave initializes and sets up the BRN by initializing a grid of MSNs and connecting them appropriately. For every real item, we also propagate a dummy item through the network. The dummy elements are required in order to provide sufficient slack in the internal buffers of the MSN, lest they overflow and trigger a failure. Once all the real and dummy items have been processed through the network, there may still be real items in the MSN local buffers. We therefore additionally flush all the local buffers, one layer at a time. To maintain obliviousness, during the flush operation, each MSN performs $\mathsf{s}\cdot\mathsf{f}$ evictions, and there are $\mathsf{f}^{\mathsf{d}-1}$ MSNs in each of the d layers of the BRN, thus resulting in $\mathsf{s}\cdot\mathsf{d}$ additional items in each of the $\mathsf{B} = \mathsf{f}^{\mathsf{d}}$ final buckets. We then use ORCompact on the buckets to reduce the buckets down to just the real items. Finally, we shuffle just the real items of each bucket using ORShuffle to reorder them, as real items would otherwise arrive in the final bucket in the order that they entered the network.

5.3 Correctness and Obliviousness

The use of MSNs for BORPStream changes its correctness argument significantly from the original BORP. We are no longer concerned about the possibility of individual bucket overflows through the network, but instead our correctness argument rests on the union bound of the probability of buffer overflows of each of the MSNs' internal buffers, P. We empirically evaluate P via a failure probability calculator that performs a Markov chain analysis; see the extended version of this paper [49, App.C] for details.

The original BORP algorithm is fully oblivious from the server's perspective, since the algorithm retrieves and stores buckets in a deterministic fashion, but it requires unobservable client memory. For BORPStream, obliviousness rests on the ProcessItem algorithm (Figure 9) of MSN being oblivious. For every incoming item, MSNs perform the same set of operations (Figure 9, Lines 7–17). An adversary can only view the memory access patterns for these OSwap operations, which are deterministic and secret independent. The final operations of Phase 2 of BORPStream are ORCompact and ORShuffle, and their obliviousness has already been established. Note that revealing the number of real items in a bucket after the ORCompact step does not violate obliviousness [3]. The number of real items in these buckets is an artifact of the distribution of random labels assigned, but it does not reveal any information about which real item ends up in a given bucket. See the extended version of this paper [49, App.A] for a formal statement and proof of the obliviousness of BORPStream.

5.4 Efficiency and Parallelization

The efficiency of BORPStream depends on some key parameter choices for the BRN: the fan f, the depth d, and the buffer size s. We use a numerical optimizer to set these as a function of the desired failure probability $2^{-\lambda}$. By setting $\mathsf{f} = O(1)$, $\mathsf{d} = O(\log n)$, and $\mathsf{s} = O(\lambda + \log n)$, we can obtain an asymptotic upper bound on the runtime of $O(n \log^2 n)$ (see the extended version of this paper [49, App.B]).

Parallelization for BORPStream is not as immediate due to the two phases. However, both phases in isolation lend themselves to parallelization in a straightforward fashion. Phase 1 can be parallelized by viewing the entire network as a pipeline of OSwap operations. For Phase 1, the number of OSwaps without parallelization is $2 \cdot n \cdot \mathsf{s} \cdot \mathsf{d}$, since each of the $2n$ items (n real and n dummy) have to perform an OSwap comparison against each item of the s sized internal buffer of the d MSNs it passes through. Pipelining this brings down the number of OSWAP steps to $2 \cdot n + \mathsf{s} \cdot \mathsf{d}$. While the speedup from parallelization here is just a small factor $\mathsf{s} \cdot \mathsf{d}$, we additionally note that since Phase 1 only handles incoming data items, and since we envision such packets to arrive haphazardly, the notion of parallelism only arises in the case that items arrive in rapid succession (or even simultaneously), in which case we get the aforementioned step complexity reduction.

Phase 2 is innately more parallelizable; the FlushBuffers component of Phase 2 requires $\mathsf{s} \cdot \mathsf{f}^{\mathsf{d}} \cdot (1 + \mathsf{s} \cdot \frac{\mathsf{d}\cdot(\mathsf{d}-1)}{2})$ OSwaps, which are trivially parallelizable across the $\mathsf{f}^{\mathsf{d}-1}$ MSNs at each layer of the BRN, and can be pipelined along the depth of the network resulting in a parallel step complexity of $\mathsf{d} \cdot \mathsf{f} \cdot \mathsf{s} + \mathsf{s}(\mathsf{d}-1)$. For the remainder of Phase 2 all the $\mathsf{B} = \mathsf{f}^{\mathsf{d}}$ buckets can be individually computed upon in parallel. Each ORCompact takes $\frac{V}{2} \log V$ OSwaps where $V = \frac{2n}{\mathsf{B}} + \mathsf{s} \cdot \mathsf{d}$ is the number of items in each bucket at the end of Phase 1, and each ORShuffle takes about $\frac{\mathsf{r}}{4} \log^2 \mathsf{r}$ OSwaps, where $\mathsf{r} = \frac{n}{\mathsf{B}}$ is the expected number of real items in each bucket.

In addition, ORCPar can be leveraged to parallelize the compaction within each bucket, and the final counts of real items in each bucket can be established after the PrefixSumPar function is invoked within ORCPar each to ensure that writes are done to the correct positions in the output array in parallel. This can be done by executing PrefixSumPar on an array of B size, populated with the sum of the outputs of PrefixSumPar of each individual bucket; i.e., the number of real items in each bucket. Additionally, the recursive shuffles of real items in each of the B buckets are also completely parallelizable, and even parallelizable within a bucket by leveraging ORSPar. The final parallel step complexity of the compaction and shuffle then boils down to $O(\log V + \max(\log \mathsf{B}, \log^2 \mathsf{r}))$.

The max term arises from the fact that depending on the underlying BRN configuration, the dominant term could either be B or $\log^2$ r, from running (in parallel) PrefixSumPar on the number of real items in each bucket and ORSPar within each bucket. It is safe to execute these operations in parallel because the locations of where to write the real items in each bucket need to be known only by the end of the shuffle itself.

6 IMPLEMENTATION

We implement and benchmark our algorithms on a server-grade Intel Xeon E3-1270 running Ubuntu 20.04 with four physical cores, 64 GB of DDR4 RAM, and support for Intel SGX. The fundamental building block of our algorithms are oblivious swaps (OSwaps). Given two data elements and a `swap_flag`, OSwap swaps them if `swap_flag` is true, while not leaking any side-channel information about the contents or whether the swap was performed. Similar to prior work [43, 46, 48], we develop these building blocks by leveraging the x86 `CMOV` instruction. In our implementation we develop a library of templated inline assembly functions that perform OSwap on buffers. These functions are templated on the size of the buffer so that we can measure performance across varying problem sizes.

Although the amount of PRM is limited, Intel SGX does allow applications to use PRM beyond that limit via virtual memory. Using more memory than the PRM limit does however incur additional paging overheads of bringing the encrypted memory pages into the PRM, since the MEE performs an integrity and freshness verification on these pages via the Enclave Page Cache Map (EPCM) [15], which is a portion of the PRM that is inaccessible to user space and set aside to ensure SGX's isolation guarantees.

6.1 Compaction Algorithms

In Figure 11a, we compare the performance of our ORCompact algorithm against Goodrich's compaction algorithm [22] in terms of the number of OSwaps performed. As expected from Theorem 3, our ORCompact algorithm performs about half the number of OSwaps as Goodrich's compaction algorithm. Furthermore from Figure 11b, we see that the performance gap can grow much larger than 2× for larger choices of n with a block size of 8 B. For instance, at the largest value of $n = 2^{23.33}$ that we see in Figure 11b, ORCompact takes 243.1±0.3 ms, while Goodrich compaction takes 1263±0.3 ms; i.e., a 5.2× speedup over the state of the art. The larger performance gap is because Goodrich compaction must update a distance array of size n in each of its $\log n$ rounds, while ORCompact performs a single linear-time computation to track the number of marked items in every subarray of the given problem.

This additional cost becomes less significant as the block size b increases, however. In Figure 11c, we plot how the computation time varies for both of these compaction algorithms as b increases using a fixed $n = 10^7$. It shows that the timing gap persists even across larger block sizes. As the block size increases, however, the timing cost is dominated by OSwaps on those larger blocks, and so the performance gap becomes closer to 2×.

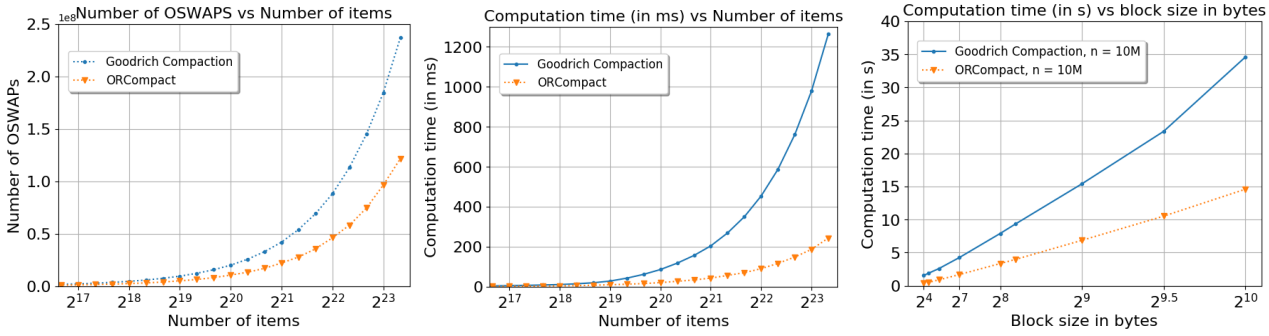
6.2 Shuffling Algorithms

We implement and evaluate the performance of the three fully oblivious shuffling algorithms presented in previous sections: ORShuffle, BORPCompact, and BORPStream. We also implement and evaluate fully oblivious versions of shuffling via OddEvenMerge sort and Bitonic sort [8]. Figure 12a shows the performance results for various problem sizes n, with a fixed block size of b = 8.

As mentioned in Section 4, shuffles based on sorting networks, such as Bitonic Shuffle, require attaching additional random tags to the data elements. In Figure 12a, we see the concrete overheads of these approaches when compared to our ORShuffle algorithm. For instance, with $n = 2^{24}$ from Figure 12a we see that ORShuffle takes 9.556±0.005 s, while Bitonic Shuffle takes 16.78±0.03 s, the performance gap widening with increasing problem size. We note that the stddevs for all our algorithm implementations are small, by virtue of their oblivious design.

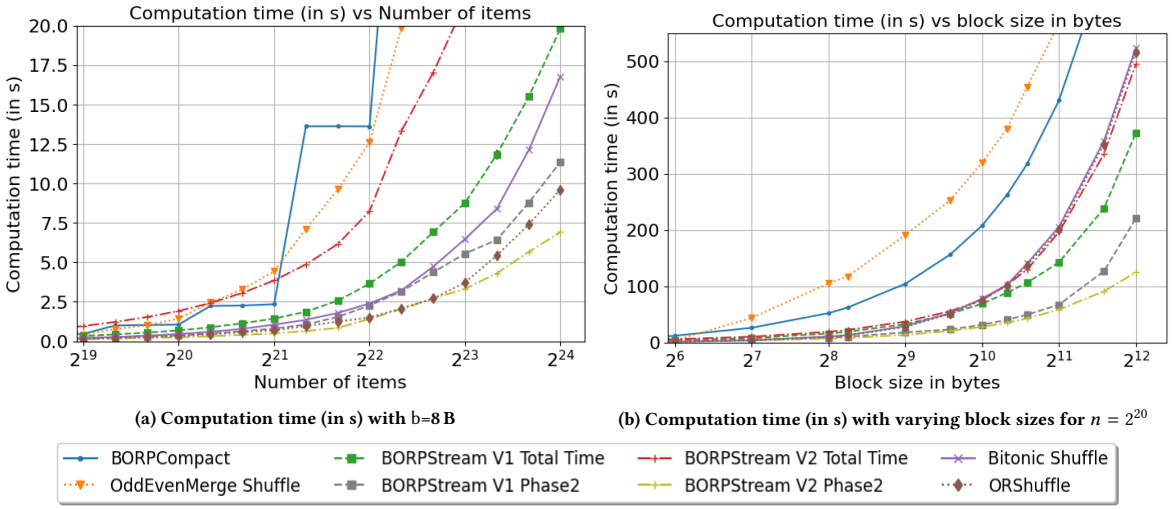
Figure 12a has four lines for BORPStream. The "V1" variants correspond to the timings (total and just for Phase 2) when the BORPStream optimizer is tuned to reduce the total time taken. However, as we alluded in Section 5.2, in the context where the data to be shuffled arrives intermittently, one could tune the optimizer to reduce the Phase 2 time instead ("V2"), and this is in fact the setting where BORPStream shines. For instance, with $n = 2^{24}$ when tuned to minimize total time (optimizer parameters f=4, d=1, s=48), we see that it takes a total time of 19.37±0.03 s, of which 11.14 ±0.03s is spent on Phase 2, implying a per-packet cost of approximately 0.49 μs as packets arrive. When tuned to minimize the time taken in Phase 2 (optimizer parameters f=4, d=5, s=49), BORPStream takes a total time of 42.42 ±0.05s, however only requires 6.784±0.002 s for Phase 2, implying a per packet cost of approximately 2.1 μs. While this is larger than that from the V1 variant, we note that this cost is still quite small in practice. In this example, if the server is likely to have 2.1 μs of otherwise idle time on average after each user submits their data, it would behoove it to use the V2-optimized BORPStream algorithm, so that once the final data arrives, the shuffle can be completed more quickly in order to reduce the latency of the overall protocol.

Finally, Figure 12b illustrates the overheads of the various oblivious shuffling algorithms with increasing block sizes, while maintaining a fixed problem size of $n = 2^{20}$. The advantage of BORPStream overall is clear here; for instance, for $n = 2^{20}$ 4 KiB blocks, ORShuffle and Bitonic Shuffle cost 515.2 ±0.3s and 523.9±0.3 s respectively, while BORPStream V1 (optimizer parameters f=4, d=3, s=47) and V2 (optimizer parameters f=4, d=4, s=47) cost 371.8 ±0.8 s and 494.7 ±0.8 s respectively, a 1.4× improvement over the state of the art. Furthermore, BORPStream V2's Phase 2 cost is 124.7±0.1 s



(a) Number of OSwaps with b=8 B (b) Computation time (in ms) with b=8 B (c) Computation time (in s) against block size

Figure 11: Comparison of Goodrich compaction and ORCompact on number of OSwaps and computation time over varying problem sizes. The elbow for Goodrich compaction in Figure 11c corresponds to it incurring paging overheads from exceeding the PRM limit.



(a) Computation time (in s) with b=8 B (b) Computation time (in s) with varying block sizes for $n = 2^{20}$

Figure 12: Comparison of shuffling algorithms in computation time (in s) over varying problem sizes. Error bars are plotted in the graph, but are too small to be seen.

which is a 4.2× improvement over the state of the art and ORShuffle. We observe performance benefits for BORPStream V1 even though it requires about 3× more OSwaps than Bitonic Shuffle; this is because BORPStream has significantly better memory locality than Bitonic Shuffle, as all the memory required for the MSNs in the first phase of BORPStream even at such large problem sizes can fit within the 90 MB PRM limit, avoiding expensive PRM paging overheads for the majority of the algorithm. The second half of BORPStream is also locality efficient as it handles items one bucket at a time. BORPStream V2 has the same memory advantages as mentioned earlier, although it incurs more than 4× the number of OSwaps in total than Bitonic Shuffle. However in Phase 2 alone, BORPStream V2 only incurs about half the number of OSwaps as Bitonic Shuffle does in total, and these OSwaps again have better locality than that of Bitonic Shuffle as the OSwaps are all contained within a single bucket, while in Bitonic Shuffle they are distributed over all the items in the entire problem. Additionally, in our evaluated range of block sizes the performance gap clearly widens as the block sizes get larger. While comparing just the Phase 2 cost of BORPStream with that of the entire ORShuffle/Bitonic Shuffle may seem unfair, note that the latter shuffles can only be performed once all the data elements are available. Therefore, ignoring the per-item timing costs of BORPStream during Phase 1 may be a fair comparison for scenarios in which data items trickle in over a relatively long time period.

To summarize, our results show that ORShuffle consistently outperforms the fastest existing fully oblivious shuffle, Bitonic Shuffle, for all problem sizes. For small block sizes, ORShuffle provides a 1.8× improvement, and BORPStream's two-phase division provides a 2.5× improvement over Bitonic Shuffle once all the packets have arrived. As block sizes get larger, ORShuffle gives only minor performance improvements over Bitonic Shuffle, but BORPStream

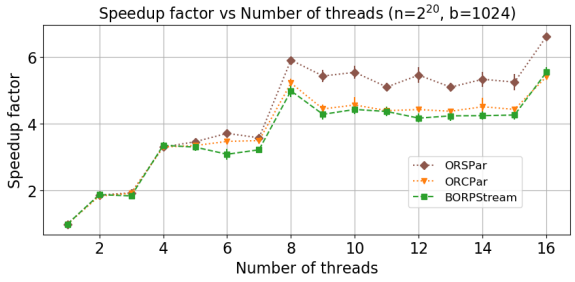


Figure 13: Speedup factor for implementations of ORSPar, ORCPar, and parallel BORPStream

provides a 1.4× speedup, and with the two-phase division it provides a 4.2× improvement once all the packets have arrived. Despite recent work on practical oblivious shuffling, which has targeted varying levels of partial obliviousness [3, 9, 42], this work demonstrates the first practical improvements in *fully* oblivious shuffling over the classic Bitonic Shuffle.

6.3 Parallel Implementation

We also implement and evaluate the parallel versions of our proposed algorithms on an 8-core Intel Xeon Gold 6334. In Fig 13 we present the speedup factor for these algorithms with increasing numbers of threads that execute the algorithm in parallel. We observe that all our algorithms get 5-6× speedup when run with 8 threads, beyond which the effect saturates as there are only 8 physical cores on our processor. While processors with SGX support are currently limited to ones with few physical cores, we envision that in the future GPU-style high core count TEEs will be available, and will benefit significantly from our algorithms.

6.4 Fully Oblivious Assembly Verifier (FOAV)

In order to ensure that the binary executed by the processor is control-flow oblivious, we instrument our C/C++ source files to insert assembly comments that state if a value in a register is "safe" before any (possibly) conditional branch inducing instructions, using `__asm__` macros. We then designed a tool that analyzes the output assembly produced by the compiler, to verify control-flow obliviousness. The tool tracks all the conditional jump instructions in the final assembly, and the corresponding *flag-manipulating instruction* (FMI) (such as `test`, `cmp`, `dec`, etc.) for each of these conditional jumps, and parses the lines above them to check if the operands of these FMIs have been marked as safe. Compiler optimizations make automating this process challenging since the optimizations may move operands to different registers before performing any FMI on them, which causes our registers marked safe by the comments to mismatch with the ones used by the FMI. To resolve this, our tool parses the assembly lines above each of the conditional jump, tracking both the register operands of the FMI, and the registers that were already marked explicitly safe from source comments, to identify other registers that may be deemed safe as well. This close analysis is performed up until it hits the start of a function or a jump instruction, and if it still cannot resolve a conditional branch as safe it marks it for closer human inspection.

Our tool automatically marks 638 out of the 660 conditional branches in our code as safe, and we manually verified the obliviousness of the remaining 22 to ensure that they are all safe as well. A majority of them arise from our use of `try-catch` blocks that ensure that the enclave was able to allocate dynamic memory for its execution, and hence these too are in fact safe conditional branches, as they do not depend on secret data.

7 RELATED WORK

7.1 Oblivious Compaction

Goodrich [22] defines compaction as outputting only the m marked elements among n total elements. Variants for *tight* and *loose* compaction are given where the output array is of size exactly m or of size $O(m)$. Goodrich gives an $O(n \log n)$ order-preserving tight compaction algorithm. It can be implemented in a way that outputs all n elements, starting with the m marked items in their original relative order (but possibly reordering the unmarked ones), using as few as $(\log_2 n - 2)n$ oblivious swaps. Note that this number is roughly a factor two more swaps than in ORCompact. Goodrich compaction can also be parallelized with $\log_2 n$ rounds at the costs of i) using a second array to move items into in parallel (otherwise items would be read and written in the same step), and ii) performing compaction on both the marked and unmarked items separately and then merging the result (otherwise the unmarked items would be overwritten). This parallel form requires approximately double the work and quadruple the memory cost of the sequential one, in contrast to ORCompact, for which the total parallel and sequential costs are about the same.

Asharov et al. [5] give a linear-time tight compaction algorithm. Dittmer and Ostrovsky [18], however, show that the constant for this compaction algorithm is more than 2^{228}. They give a different $O(n)$ tight compaction algorithm that runs in time at most $9405.73n$, which, for realistic n, is much larger than the additional $\log n$ and constant factor in Goodrich compaction. Similarly, Mitchell and Zimmerman [37] give an $O(n \log \log n)$ algorithm for compaction, but the analysis by Falk and Ostrovsky [21] indicates that the algorithm requires $4n \log \log n + 14n$ OSwaps (with a 2^{-40} failure probability), which is more than in ORCompact unless $n > 2^{78}$.

We also note that fully oblivious compaction is suitable for other (non-TEE) secure computing technologies, such as secure multiparty computation (MPC) and fully homomorphic encryption. For example, Blanton and Aguiar [10] use Goodrich compaction to perform MPC set operations. They could instead use ORCompact and reduce the expensive comparisons by a factor of two.

7.2 Oblivious Shuffling

Melbourne Shuffle [42] is a data-oblivious shuffle designed for cloud storage with minimal client storage overheads. Its optimized (recursive) variant requires $O(\sqrt[4]{n})$ client or private memory to store and permute items, and this private memory needs to be obscure to the server. To instantiate Melbourne Shuffle in a TEE, one would have to convert the computation that happens within this client memory to be fully oblivious. This computation however is non-trivial to convert into a fully oblivious variant; it requires obliviously distributing $\sqrt[4]{n}$ items across $\sqrt[4]{n}$ logical buckets (that reside on the server), as well as padding them up to $p \cdot \sqrt[4]{n}$ items for each of those

buckets, where the constant p dictates the failure probability of the algorithm. The algorithm fails if the shuffle permutation requires moving more than $p \cdot \sqrt[c]{n}$ items to any of the destination buckets. The algorithm induces $O((c-1)(n^{\frac{c-1}{c}}))$ such client-side operations over $n^{1/c}$ items. One could convert this into a fully oblivious variant in quadratic time over the $n^{1/c}$ items it operates on, which leads to an algorithm of $O((c-1)(n^{\frac{c+1}{c}}))$ complexity, asymptotically worse than the ones we proposed. Additionally, there are constants missing here that further blow up the cost of implementing this in practice within a TEE.

Stash Shuffle, the shuffling algorithm in Prochlo [9], was designed to be efficient for the TEE setting. The shuffle is similar to a Melbourne shuffle without the recursion, but it also uses a stash of size $O(\sqrt{n})$ to hold items that cannot be distributed in a given round to their intended output buckets. Stash Shuffle is designed to reduce the number of times a single item has to be processed by the shuffler, which in classical oblivious shuffling algorithms is $\log^2 n$; their work reduces this down to a constant factor. Their construction succeeds in their desired efficiency goal only in the setting where access patterns within PRM are safe from side channels, but as we detail in Section 2.3, this is no longer a realistic assumption.

Sorting networks with random labels are a natural approach for oblivious shuffling [3]. They are a common tool for oblivious sorting inside TEE systems [1, 20, 29, 59]. The bitonic network [8] is preferred because it performs better in practice [19], even though the odd-even mergesort network [8] is slightly smaller. Probabilistic sorting networks [33] can achieve smaller size at the cost of failing to sort correctly with certain probability. They have not been evaluated for oblivious use in TEEs. Random sorting networks appear to be asymptotically efficient [51], but their practical performance has not been demonstrated [16].

7.3 Systems Using TEEs and Oblivious Algorithms

Several privacy-preserving systems have been proposed that make use of TEEs and oblivious algorithms.

The Oblix system [36] provides secure search of encrypted data using a TEE. Oblix uses "doubly oblivious" algorithms, in which memory accesses are oblivious to both protected memory and external memory. The definition of double obliviousness does not include the control flow, in contrast to full obliviousness. The authors note that the Oblix implementation does not guarantee obliviousness after compilation, an issue we address with FOAV.

Opaque [59] is a TEE-based platform for secure data analytics. It performs SQL-type queries on tables, such as filter and join. Opaque assumes the existence of some memory where accesses are not observable to the adversary, and it does not provide control-flow obliviousness. Opaque performs compaction to filter unwanted records and to remove dummy records created during oblivious processing, and it could benefit from our oblivious compaction algorithm ORCompact.

Prochlo [9] provides privacy-preserving telemetry, for example to collect information on how a web browser is used. The Prochlo architecture includes shuffling as a major component to help anonymize user data. Prochlo introduces the aforementioned Stash Shuffle algorithm, which is external-memory oblivious. To obtain security against memory and timing side channels, Prochlo could instead use our ORShuffle algorithm. Alternatively, Prochlo could replace the algorithms it uses to process data in protected memory with oblivious versions. These algorithms accomplish shuffling and compaction tasks, and so our algorithms could be used.

Krastnikov et al. [29] give oblivious algorithms for database joins. A key operation uses a reverse compaction operation (called "distribution"), for which they need order preservation. They use Goodrich compaction in reverse, and ORCompact in reverse could be used instead to improve performance.

Ohrimenko et al. [43] propose to use TEEs for privacy-preserving machine learning. They give several machine-learning algorithms in which all memory accesses are data-oblivious. Several of their algorithms use shuffling as a key component, including to randomize the training samples for SVMs and neural networks and to randomize initial clusters for k-means clustering. These algorithms could use ORShuffle or BORPStream to improve performance.

Scramble-then-Compute (StC) [17] makes certain non-oblivious algorithms oblivious by preceding them with an oblivious shuffle. StC can eliminate memory side channels within TEEs for otherwise non-oblivious algorithms for problems including compaction, sorting, and selection. StC is described and evaluated using the Melbourne shuffle [42], which only provides external-memory obliviousness. ORShuffle or BORPStream could be used instead in StC to provide fully oblivious algorithms for the same problems.

8 CONCLUSION

TEE systems are gaining traction as a candidate to enable large-scale applications and data analytics that maintain user privacy. In parallel, however, we observe ever-growing side-channel attacks against these systems. In this work we demonstrate fully oblivious algorithm designs and implementations for the fundamental primitives of tight compaction and shuffling, which serve as building blocks for myriad applications that have been proposed in the TEE setting. Our algorithms outperform the state of the art in concrete timing benchmarks as illustrated by our experimental results. We additionally verify that our implementations indeed achieve the most elusive degree of obliviousness, control-flow obliviousness, by instrumenting an assembly checker tool to this end. The more recent side-channel attacks that precisely extract cache granular memory access patterns or fine-grained control flow information of programs executing in a TEE illustrate how much more powerful such attack vectors are than previously understood. It is therefore critical that we design applications that run in TEEs to be fully oblivious lest they be susceptible to these side channels.

ACKNOWLEDGMENTS

This work was supported by the Office of Naval Research. We thank the Ontario Graduate Scholarships program, NSERC (CRDPJ-534381), and the Royal Bank of Canada for supporting this work. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

REFERENCES

[1] A K M Mubashwir Alam, Sagar Sharma, and Keke Chen. 2021. SGX-MR: Regulating Dataflows for Protecting Access Patterns of Data- Intensive SGX Applications.

Proceedings on Privacy Enhancing Technologies 2021, 1 (2021).

[2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. https://software.intel.com/content/www/us/en/develop/articles/innovative-technology-for-cpu-based-attestation-and-sealing.html. Accessed December 2021.

[3] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket oblivious sort: An extremely simple oblivious sort. In *Symposium on Simplicity in Algorithms*.

[4] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2018. OptORAMa: Optimal Oblivious RAM. Cryptology ePrint Archive, Report 2018/892.

[5] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, Elaine Shi, and Yuval Ishai. 2020. OptORAMa: Optimal Oblivious RAM. In *Advances in Cryptology (EUROCRYPT)*.

[6] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. 2020. Oblivious Parallel Tight Compaction. In *Conference on Information-Theoretic Cryptography (ITC)*.

[7] Borja Balle, Gilles Barthe, and Marco Gaboardi. 2018. Privacy Amplification by Subsampling: Tight Analyses via Couplings and Divergences. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[8] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*.

[9] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. 2017. Prochlo: Strong Privacy for Analytics in the Crowd. In *Symposium on Operating Systems Principles (SOSP)*.

[10] Marina Blanton and Everaldo Aguiar. 2016. Private and oblivious set and multiset operations. *International Journal of Information Security* 15, 5 (2016).

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. https://doi.org/10.1109/EuroSP.2019.00020

[13] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*.

[14] Scott D Constable and Steve Chapin. 2018. *libOblivious: A C++ Library for Oblivious Data Structures and Algorithms*. Technical Report 184. Syracuse University Electrical Engineering and Computer Science.

[15] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained*. Technical Report 2016/086. IACR ePrint.

[16] Artur Czumaj. 2015. Random permutations using switching networks. In *ACM Symposium on Theory of Computing (STOC)*.

[17] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. 2017. Privacy-Preserving Computation with Trusted Computing via Scramble-then-Compute. *Proceedings on Priviacy Enhancing Technologies* 2017, 3 (2017).

[18] Sam Dittmer and Rafail Ostrovsky. 2020. Oblivious tight compaction in O(n) time with smaller constant. In *Conference on Security and Cryptography for Networks (SCN)*.

[19] Kris Vestergaard Ebbesen. 2015. *On the Practicality of Data-oblivious Sorting*. Ph. D. Dissertation. Aarhus Universitet, Datalogisk Institut.

[20] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proceedings of the VLDB Endowment* 13, 2 (2019).

[21] Brett Hemenway Falk and Rafail Ostrovsky. 2021. Secure Merge with O (n log log n) Secure Operations. In *Information-Theoretic Cryptography (ITC 2021)*.

[22] Michael T Goodrich. 2011. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[23] J. Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. *European Workshop on Systems Security (EuroSec)* (2017).

[24] W Daniel Hillis and Guy L Steele Jr. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986).

[25] Intel. 2012. Intel Trusted Execution Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html. Accessed December 2021.

[26] Intel. 2018. Q3 2018 Speculative Execution Side Channel Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html. Accessed December 2021.

[27] Intel. 2019. Intel Processors Voltage Settings Modification Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html. Accessed December 2021.

[28] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf. Accessed December 2021.

[29] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Proceedings of the VLDB Endowment* 13, 12 (2020).

[30] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX Security Symposium*.

[31] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting TEEs. In *European Conference on Computer Systems (EuroSys)*.

[32] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*.

[33] Tom Leighton, Yuan Ma, and Torsten Suel. 1997. On probabilistic networks for selection, merging, and sorting. *Theory of Computing Systems* 30, 6 (1997).

[34] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. 2019. Can we overcome the n log n barrier for oblivious sorting?. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[35] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy (S&P)*.

[36] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *IEEE Symposium on Security and Privacy (S&P)*.

[37] John C Mitchell and Joe Zimmerman. 2014. Data-oblivious data structures. In *Symposium on Theoretical Aspects of Computer Science (STACS)*.

[38] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Cryptographic Hardware and Embedded Systems (CHES)*.

[39] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *USENIX Security Symposium*.

[40] Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized algorithms*. Cambridge University Press.

[41] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE Symposium on Security and Privacy (S&P)*.

[42] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. 2014. The Melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming (ICALP)*.

[43] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*.

[44] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *RSA Conference (CT-RSA)*.

[45] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. 2020. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *USENIX Security Symposium*.

[46] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium*.

[47] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. 2018. Efficient parallel random sampling—vectorized, cache-efficient, and online. *ACM Transactions on Mathematical Software (TOMS)* 44, 3 (2018).

[48] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Network and Distributed System Security Symposium (NDSS)*.

[49] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. https://crysp.uwaterloo.ca/software/obliv/.

[50] Sajin Sasy and Olga Ohrimenko. 2019. Oblivious Sampling Algorithms for Private Data Analysis. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[51] Mingwei Shih. 2019. *Securing Intel SGX against side-channel attacks via load-time synthesis*. Ph. D. Dissertation. Georgia Institute of Technology.

[52] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium (NDSS)*.

[53] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*.

[54] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What you get is what you C: Controlling side effects in mainstream C compilers. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.

[55] Ernst Stadlober. 1990. The ratio of uniforms approach for generating discrete random variates. *Journal of computational and applied mathematics* 31, 1 (1990).

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

[57] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*.

[58] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (S&P)*.

[59] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.