VASA: Vector AES Instructions for Security Applications*

(Full Version)

Jean-Pierre Münch
TU Darmstadt
Darmstadt, Germany
jean-pierre.muench@posteo.de

Thomas Schneider
TU Darmstadt
Darmstadt, Germany
schneider@encrypto.cs.tu-darmstadt.de

Hossein Yalame
TU Darmstadt
Darmstadt, Germany
yalame@encrypto.cs.tu-darmstadt.de

ABSTRACT

Due to standardization, AES is today's most widely used block cipher. Its security is well-studied and hardware acceleration is available on a variety of platforms. Following the success of the Intel AES New Instructions (AES-NI), support for Vectorized AES (VAES) has been added in 2018 and already shown to be useful to accelerate many implementations of AES-based algorithms where the order of AES evaluations is fixed a priori.

In our work, we focus on using VAES to accelerate the computation in secure multi-party computation protocols and applications. For some MPC building blocks, such as OT extension, the AES operations are independent and known a priori and hence can be easily parallelized, similar to the original paper on VAES by Drucker et al. (ITNG'19). We evaluate the performance impact of using VAES in the AES-CTR implementations used in Microsoft CrypTFlow2, and the EMP-OT library which we accelerate by up to 24%.

The more complex case that we study for the first time in our paper are dependent AES calls that are not fixed yet in advance and hence cannot be parallelized manually. This is the case for garbling schemes. To get optimal efficiency from the hardware, enough independent calls need to be combined for each batch of AES executions. We identify such batches using a deferred execution technique paired with early execution to reduce non-locality issues and more static techniques using circuit depth and explicit gate independence. We present a performance and a modularity-focused technique to compute the AES operations efficiently while also immediately using the results and preparing the inputs. Using these techniques, we achieve a performance improvement via VAES of up to 244% for the ABY framework and of up to 28% for the EMP-AGMPC framework. By implementing several garbling schemes from the literature using VAES acceleration, we obtain a 171% better performance for ABY.

KEYWORDS

privacy preserving machine learning, secure multi-party computation, VAES.

1 INTRODUCTION

The primitive of choice for encryption and similar tasks is AES. It is used for communication encryption [71, 84], disk storage encryption [21, 34], and database encryption [72] among other applications. To improve the performance and resource utilization of this important primitive, the AES-NI extension to the x86 instruction set was introduced [5, 56] with common implementations computing AES-128 with ~ 1.3 cycles/byte on one core [2].

History of VAES. Further improving on this, Intel has developed support for vector AES (VAES) instructions [33] and shipped it starting with their Ice Lake microarchitecture [35]. These VAES instructions compute a single round of AES on different blocks, using multiple different round keys [33, 56]. The original paper of [33] already discussed the importance of batching data to the vector AES-NI instructions and microarchitectural properties of these instructions. The authors demonstrated how to apply VAES to several modes of operations of block ciphers such as AES-CTR, AES-CBC, AES-GCM, and AES-GCM-SIV with up to 4× performance improvements. This increased throughput of AES in standard modes of operation can yield direct performance improvements for applications such as storage encryption, disk encryption, database encryption, or secure channels (TLS) [21, 34, 72, 84, 87] and VAES is already included in the popular OpenSSL library [90]. Subsequently, Drucker and Gueron showed how to use VAES to accelerate Pseudo-Random Functions (PRFs) and Pseudo-Random Generators (PRGs) [30]. Multiple NIST Post Quantum Cryptography Project candidates use Deterministic Random Bit Generators (DRBGs) for which the implementation of Drucker and Siri achieves up to 4× performance improvement using VAES [29]. The contribution of this VAES-accelerated DRBG was evaluated for the post-quantum secure multivariate-polynomial signature scheme Rainbow [26] in [31], and for the key encapsulation mechanism BIKE [6] in [32].

Our Motivation. What is common to all these applications of VAES studied before is that the algorithm is fixed beforehand, and hence the parallelization can be done manually. For maximum throughput with VAES, the main challenge is to batch enough independent AES calls together for the AES hardware units to be constantly busy and not idle when processing blocks.

However, finding a good batching becomes much more challenging when the algorithm and hence sequence of AES operations is not fixed in advance. Some AES operations can depend on the output of others but some do not and many small memory-abstracted library invocations are expensive. This batching problem and its solutions are not unique to AES on x86-64 using VAES (which is our focus). It can be generalized to all non-trivial implementations of cryptographic primitives which includes pipelined AES implementations on ARM [7], bitsliced AES implementations [17, 63] as well as more unusual techniques like instance-vectorized hash functions. A natural area where such complex dependencies occur is Secure Multi-party Computation (MPC), especially garbled circuits [10, 40, 67, 86, 96, 98], which is why we use them for assessing the performance impact for VAES. More concretely, with

*Please cite the conference version of this paper published at 37th Annual Computer Security Applications Conference (ACSAC'21) [74].

Table 1: Summary of our performance improvements. New Batched AES-NI indicates whether the implementation received an *additional* batching AES-NI implementation. VAES indicates whether the performance improvement includes VAES.

Framework	New Batched AES-NI	VAES	Max. Total Improv.
ABY (Ref) [13, 25, 98]	✓	✓	244%
ABY (Custom) [25, 40, 41, 98]	✗	✓	171%
EMP-OT [91]	✗	✓	30%
EMP-AGMPC [91, 93]	✓	✓	24%
CrypTFlow2 [83]	✗	✓	52%

garbled circuits, typically binary circuits using primarily AND and XOR gates are evaluated with XOR gates only requiring XOR operations [67], whereas AND gates do require AES operations to be and sending ciphertexts. These garbled circuits can then be used for high performance secure two-party computation, interactive zero-knowledge proofs of arbitrary statements [44, 60, 98], and other applications.

MPC allows to securely compute a public function on private input data provided by multiple parties and hence is an interactive way for computing under encryption. Since several years, a multitude of companies, including Alibaba, Bosch, NTT, and Unbound among many others in the MPC Alliance [4], are working on MPC technology. We study the ABY framework [25] for passively secure two-party computation and the EMP-AGMPC [91, 93] framework for actively secure multi-party computation. As we are manually changing the implementation of these schemes without changing the protocols, we substantially increase the deployability of these frameworks and dependent works as well as providing guidance to how similar effects can be achieved for similar frameworks.

Privacy-preserving machine-learning (PPML) is a popular application of MPC. Here, general machine-learning techniques are run on private data while also protecting the model parameters. The private output is the inference or training result [39]. PPML has become a hot topic in recent years and gained the attention of major software, service and hardware vendors, e.g., Facebook [66], Google [16], Intel [15], and Microsoft [83], all of whom are working on increasing its practicality. Applications of PPML include private healthcare-based inference, e.g., to predict illnesses [22, 69, 85], private healthcare model training to acquire models without having to reveal patient data [1], and private clustering to partition data according with common features [73]. In particular, in this work, we discuss private ML inference in the state-of-the-art framework Microsoft CrypTFlow2 [83] where one party holds a pre-trained model and the other a data item to be classified and then the protocol allows classification using the model without the two parties revealing their private inputs. We improve CrypTFlow2 [83] using VAES. As our focus lies on manual implementation improvements, we substantially increase such PPML applications' deployability without sacrificing compatibility or security.

Our Contributions. Our main contributions are as follows:

- We expand the focus of VAES from microarchitectural issues where the order of AES operations is fixed a priori, to protocol and implementation design where the sequence of AES operations is not known in advance. For this, we introduce automatic batch identification and computation techniques for efficient use of AES in complex security applications.
- We report the first performance measurements for VAES in the area of Multi-Party Computation (MPC) and show performance improvements for the MPC frameworks ABY, EMP-OT and EMP-AGMPC, as well as the PPML framework CrypTFlow2. Our improvements are summarized in Table 1.
- We provide our implementations for re-use by others and as guidance for future implementation efforts at https://encrypto.de/code/VASA.

Outline. The rest of this paper is organized as follows: We start with providing the necessary background on the investigated types of MPC and the hardware acceleration of AES in x86 processors (§ 2). Next, we provide context to our work with related work (§ 3). Following that, we describe our computational framework for efficient batch identification and computation and how we applied it (§ 4). Next, we evaluate and discuss the performance of the applications (§ 5). Finally, we conclude and provide possible future research directions (§ 6).

2 BACKGROUND

In this section, we provide a brief background on secure multi-party computation and how AES is computed using AES-NI and VAES on x86-based processors.

2.1 AES Computation

There are two instruction set extensions on x86 for providing functionality relating to the computation of AES: the AES new instructions (AES-NI) and the vector AES instructions (VAES) [5, 33, 56]. For the encryption direction, the key instructions from these extensions are `AESENC` and `AESENCLAST` which compute a single AES round and the last AES round, respectively. The difference between AES-NI and VAES is the instructions' width and how many blocks and round keys they work with: AES-NI is restricted to one and VAES also allows two or four. Thus, one can compute AES-128 by chaining an XOR operation with nine `AESENC` and one `AESENCLAST` using a pre-expanded key. The key expansion itself can also take advantage of the `AESENCLAST` instruction and is most efficiently done using the technique of Gueron et al. [40]. As most modern x86 processors providing the AES extensions are pipelined, the data dependency between the AES instructions can lead to pipeline stalls if not filled otherwise. This is the reason why multiple independent AES calls are batched together, allowing interleaved execution of the instructions, i.e., starting execution of the second round of all

batched AES calls before starting execution of the third round of any one of them.

This leads to optimal, minimal sizes for batches of AES calls which depend on the microarchitecture involved as they need to hide the latency of the instructions using the throughput and the width of the instructions. A summary of these performance characteristics using the data of [38] for modern x86 processor architectures is provided in Table 2. The performance characteristics of 128-bit AES instructions have remained the same for all successors of AMD's Zen architecture so far. Also the performance characteristics of the `AESENC` and `AESENCLAST` instructions are identical.

Table 2: AES-NI and VAES instruction latencies, throughput [38], and resulting minimal batch size for optimal efficiency. Width 128 bits corresponds to AES-NI and other values are VAES. Cycles per instruction is abbreviated as "cyc/instr".

Architecture	Width [bits]	Latency [cycles]	Throughput [cyc/instr]	Minimal Batch Size
Intel Haswell	128	7	1	7
Intel Skylake	128	4	1	4
Intel IceLake	128	3	0.5	6
	256	3	0.5	12
	512	3	1	12
AMD Zen	128	4	0.5	8
AMD Zen3	256	4	0.5	16

2.2 Secure Multi-Party Computation

The goal of secure multi-party computation (MPC) is to compute arbitrary functions among multiple parties on private inputs only known to one party each [12, 14, 79, 95, 96]. Most relevant for this work are protocols for oblivious transfer (OT), garbled circuits (GC), and privacy-preserving machine-learning (PPML).

Oblivious Transfer (OT). In oblivious transfer, one party (the receiver) inputs a choice bit and the other (the sender) supplies two messages. The receiver then learns only the message corresponding to the choice bit. The computation of OT protocols typically uses a small number of invocations of a public-key-based OT protocol [23, 75] to extend to a larger number of OTs using symmetric cryptography [8, 9, 57]. The primary bottleneck of these OT extension protocols are the communication time, the computation of a bit matrix transposition, and the computation of encryption operations using AES [8]. Common variants of the above OT functionality which allow to decrease communication are random OT (R-OT) where the sender gets two random strings and the receiver gets one of them depending on the choice bit, and correlated OT (C-OT) where the sender can input a correlation that the returned strings have to satisfy. Additionally, there has been a line of research looking to further minimize the communication needed for C-OT using a learning parity with noise (LPN) assumption [18, 19, 94]. These pseudo-random correlation generators (PCGs), like FERRET [94], reduce communication at the expense of computation, and increased complexity where a large matrix-vector product with randomized entries is computed.

Garbled Circuits (GC). Secure computation of general functions is typically performed using a circuit-oriented representation of that function. Garbled circuits (GCs) are one approach for this, originally proposed for two parties [96] and later generalized multiple parties [12]. In GC, the key invariant is that each wire's value is represented by two random keys which represent the zero and one bits. The garbling party knows both wire keys and the evaluating party only ever learns one key for each wire. For each gate a garbled table is generated forming the garbled circuit, to allow translation of a given pair of gate-input-wire keys to the output wire key corresponding to the correct output bit. The evaluator obtains the keys corresponding to the circuit input wires via OT. Early constructions [12, 76, 96] used garbled tables that could effectively be generated in parallel due to a lack of data dependencies. However, more modern schemes like free-XOR [67], HalfGates [98], or PRF-based garbling [40] require a topologically ordered processing of gates in exchange for requiring only two ciphertexts instead of three per AND gate, and XOR gates require no communication in free-XOR [67] or one ciphertext in PRF-garbling [40]. As these schemes require at least four applications of a cryptographic function on some counter or gate identifier as well as the gate input keys to generate the tables, most implementations use AES with a fixed key [13, 42] though instantiations with variable keys were also proposed in [40, 41]. Yao's garbled circuits protocol described above initially provides security against passive adversaries [70] and there have been extensions in research to security against active adversaries [51, 77, 78, 92, 93] that can arbitrarily deviate from the protocol specification. The latest of these schemes [92, 93] uses the free-XOR optimization [67] and parties jointly compute authenticated versions of the garbled tables so that a malicious garbler does not know the actual tables nor can tamper with them while a malicious evaluator only sees random-looking ciphertexts.

AES vs. LowMC. With free-XOR [67] and the S-box of [17], a Boolean circuit for AES consists of 5 210 AND gates [47]. Starting with LowMC [3], several dedicated MPC-friendly block ciphers have been designed that minimize the number of AND gates (or also multiplicative depth) over AES [3, 27, 28, 61]. Due to their smaller and/or shallower circuits, such *MPC-friendly block ciphers improve the function that is evaluated via MPC*, e.g., to privately evaluate a block cipher, called Oblivious Pseudo-Random Function (OPRF) [82], which has several applications like private set intersection for unbalanced set sizes in private contact discovery [62, 65]. However, the *MPC protocols themselves are still implemented with AES* (e.g., garbling schemes, OT extension, or PRFs). The reason for that is the superb performance of hardware acceleration of AES in today's CPUs which are highly optimized ASICs that require only ~ 1.3 cycles/byte on one core using AES-NI [2]. In our paper, we show how the efficiency of such implementations of MPC protocols can be further improved by using VAES.

Privacy-Preserving Machine-Learning (PPML). The goal of PPML is to apply machine-learning techniques while preserving the privacy of the data and models [37, 39, 45, 64]. While this

application can include training and inference [39], we focus on inference, in particular on inference for neural networks as done in Microsoft CrypTFlow2 [83]. This involves computing the linear and non-linear stages using optimized protocols for the client's private data input and the server's private model input, only yielding the result to the client. We note that the practicality of PPML has improved drastically over time to the point where now accurate, full-sized neural network inference is possible in a privacy-preserving setting even on moderately powerful hardware [83].

3 RELATED WORK

In this section, we discuss how our work relates to previous work. In particular, we discuss the relation to previous protocol-level and implementation-level improvements.

3.1 Protocol-Level Improvements

One primary direction for research in the past has been to improve the protocols themselves, e.g., by reducing the amount of communication or the number of invocations to computationally expensive primitives [10, 43, 67, 76, 86, 92, 98]. In addition, some works handle the circuit generation for MPC protocols from specifications in a high-level language by using industry-grade hardware synthesis tools and tweaking them for logic synthesis [24, 46, 80, 88]. Our work is largely orthogonal to these approaches as we focus on improving the implementations and the frameworks used for them. However, there are advances in protocol design which significantly complicate efficient implementation, e.g., the requirement for gates in circuits to be processed in topological order [40, 67, 98]. There have been prior works that modified the protocol and increased communication to allow for more efficient computation [55], but we do not follow their approach and maintain protocol compatibility. This focus on implementation improvements for relatively low-level building blocks allows protocol compatible performance improvements for the discussed protocols and those building on top of it. Such works include Cerebro [99], TinyGarble2 [52], and CrypTFlow2 [83] all of which build on EMP [91] and can thus profit from our improvements of EMP.

3.2 Implementation-Level Improvements

Another major direction has been improving the implementation of the protocols. This has seen four sub-directions: Improving the performance of individual operations, improving the parallelization of the implementation, improving the memory behavior, and using dedicated hardware to accelerate computationally expensive steps.

Operations. In OT extension, bit matrix transposition is one of the most computationally expensive operations [8]. Previous optimizations of this operation have been using an asymptotically optimal transposition algorithm [36], or 128-bit vector registers [91]. We improve on the latter through the use of wider AVX512 vector registers instead. Beyond this, OT extension has been a major application of fixed-key AES [13] on which we improve through the use of VAES instead of AES-NI for the implementation. Furthermore, there have been efforts to increase the performance of individual operations in GC, e.g., improving the implementation performance of the individual garbling and evaluation operations for individual gates [13, 40]. We improve upon these prior works by considering multiple gates of the same type at once. A natural question is, whether a library like OpenSSL can be used for implementing AES operations. This is an appropriate solution if only large batches of AES calls occur and these are well-supported by OpenSSL. However, this would not allow the use of VAES which is currently not used by OpenSSL, and it would bring significant overhead for smaller batches due to the memory abstraction needed.

Parallelization. Previous work to parallelize the evaluation of garbled circuits has seen coarse- and fine-grained approaches [11, 20, 50, 55]. Coarse-grained approaches [11, 20] are typically used to have multiple threads compute different parts of the same garbled circuit and are largely orthogonal to our in-thread optimizations of the computation strategy. Alternatively, they may have traded communication, e.g., not using free-XOR, for added parallelism to exploiting using dedicated hardware like graphics processing units or Intel Quick Assist Technology [55]. The more fine-grained approaches [11, 20] have primarily focused on using a layering technique, as we also discuss, however, intending to outsource the work to different threads instead of exploit the high instruction-level parallelism that modern processors provide. Additionally, previous work has suggested splitting the garbling and the evaluating roles with a suitable sub-division of circuits [20] or overlapping the computation with the garbling and evaluation operations [50], both of which are orthogonal to what we do.

Memory Behavior. A smaller line of previous research has explored the limitation of memory use for GC [48, 52, 68, 88, 97]. Their motivation for this was two-fold in allowing the computation of large circuits not fitting into most memory configurations and improving locality for caches through smaller code and data. We note that the techniques to only partially load circuits into memory are orthogonal to ours, requiring at most invoking early execution occasionally. We also consider cache locality important. However, our focus is more on the actual computation and the first-level cache as opposed to keeping the data in a cache at all.

Hardware-Acceleration. There has been a line of research using field-programmable gate-arrays (FPGAs) to accelerate garbled circuit operations [53, 54, 58, 59, 89]. Our work is independent of and alternative to the main contributions of these prior works. However, the scheduling discussed for FASE [53] is similar for hardware to what we do for identifying batches, though their techniques are focused on the specific dedicated hardware architecture they build, making it unsuitable for our software-oriented approach.

4 OUR FRAMEWORK

The first step in our manually implemented techniques to apply VAES is the identification of batches of independent AES calls for small-scale batch processing (§ 4.1). The second step is to process the AES operations (§ 4.2). Finally, we show how we used these techniques with the ABY [25], EMP-OT [91], CrypTFlow2 [83], and EMP-AGMPC [91, 93] frameworks (§ 4.3).

4.1 Batch Identification

For identifying batches, we use two approaches: *dynamic batching* and *static batching*. *Dynamic batching* primarily uses runtime information for minimally invasive batching. *Static batching* provides reusable batching information from preprocessing but requires more substantial changes to the code.

4.1.1 Dynamic Batching. The core idea behind dynamic batching is to defer execution of operations until they are actually needed and to compute all pending operations when *one* is needed. In processing circuits, the application of this works by modifying the main processing loop iterating over all gates and adding AES-based AND gates to a queue and processing all queued AND gates as a batch once any one of them is referenced as an input dependency. An example of when the processing is invoked is provided in Fig. 1. Implementing this technique requires potentially a few hours of manual effort to identify the core processing loop, to implement the deferred execution identification, and to identify relevant modifications and extensions which we briefly discuss next.

Correctness Extensions. The basic technique works well if there is one type of non-free gates requiring AES operations. However, some schemes have AND and XOR operations requiring AES operations using a shared gate index counter to uniquely produce values per-gate. For these, new design space choices manifest, in particular, whether it is possible and desirable to separate the domains of the counters or to track the gate identifiers as well and not just the minimal information for computing the gate. Additionally, one can imagine that it is possible to not maintain separate queues for the different gate types but rather join them into a shared one which complicates the gate processing at the potential of gained performance through more AES calls being potentially batched together to reach minimum optimal batch size even in complex circuits. Furthermore, we note that dynamic batching can be combined with the approach of having a variable number of cryptographic gates associated with an administrative gate, in which case it is beneficial to track the number of actual gate tasks associated with each administrative gate and keep a global count to allow the batch processing algorithm to choose appropriate sub-batches. Both of these extensions each require a few hours of effort for the architectural changes.

Optimizations. The basic batching techniques have further optimizations. First, the use of this batching can inadvertently lead to significant gaps in time between visiting and enqueuing a gate and processing it, meaning it might be pushed out of registers or lower-level caches. To avoid such unloads, one should consider to regularly empty the queue by processing the stored tasks even if more tasks could still be added without violating correctness. This holds especially true if any given processed sub-batch only processes a small number of gates, e.g., $b = 4$, and the queue has reached a size that is a multiple of b. Additionally, one can consider to only partially process the stored tasks in the queue using a multiple of the preferred processing width to potentially allow more gates to be directly enqueued without triggering processing at an undesirable length. When the basic technique encounters an AND gate referencing a queued AND gate, it will always trigger the computation of all queued AND gates. Another optimization in this scenario is to check whether the referenced AND gate is early enough in the queue which is guaranteed to have been processed once the processing reaches the current AND gate and then enqueueing the current AND gate without triggering processing. The implementation effort for these optimizations potentially requires a few hours of effort on top of the basic queue implementation.

4.1.2 Static Batching. A different approach than the dynamic technique is to preprocess the circuit to gain more holistic information on batching opportunities. These techniques can be paired with dynamic batching techniques for further improved efficiency. The three techniques we discuss are layering (identifying layers of dependencies), SIMD (grouping multiple guaranteed independent gates into one administrative one), and a more generic smart arrangement.

Layering. Layering techniques assign a gate to how many non-free gates lie between it and the original input. Non-free gates on equal layers are then necessarily independent and each layer can be seen as a batch of AES calls to be computed. An example of associated layers is provided in the right graph of Fig. 1. Layering can be done in addition to dynamic batching which can potentially identify independent tasks across layers, e.g., if the first gate of the second layer references the first gate of the first layer and early evaluation or peephole optimizations allow such batches. The effort to add layering support to an implementation varies significantly with the architecture and can range from a few hours for adding, computing and using the attribute to significantly more if a more complex processing strategy than a sequential loop is used.

SIMD. Single-instruction multiple-data (SIMD) gates are explicitly specified administrative gates that represent the same gate being applied to multiple input wires in parallel. They present natural opportunities for batches and even allow batching techniques in more complex gate scheduling scenarios where other techniques are not applicable. The cost to this is either the identification of such SIMD tasks or the need for the execution of a circuit several time as a batch as well as the need for explicit program-level representation. Similarly to layering, the implementation cost for SIMD gates varies with the architecture and can quickly take a dozen or more hours. As all gate processing methods need to be SIMD-aware, gates must be extracted and collected and SIMD gates must be specified or detected in a given circuit description.

Smart Arrangement. This technique is more general and provides heuristics for circuit generators and manually optimized building blocks of gates. For example, circuit generators should output circuits that allow circuit-internal SIMD gate operations and prefer larger layers over smaller layers. An example of such improved gate arrangement is provided from the left to the right graph in Fig. 1. Additionally, locality has to be considered when generating circuits, i.e., usage of wires must stay close to where they are generated as not to push the wire values out of caches, while maintaining enough distance to allow batching on current and more instruction-level parallel future architectures.

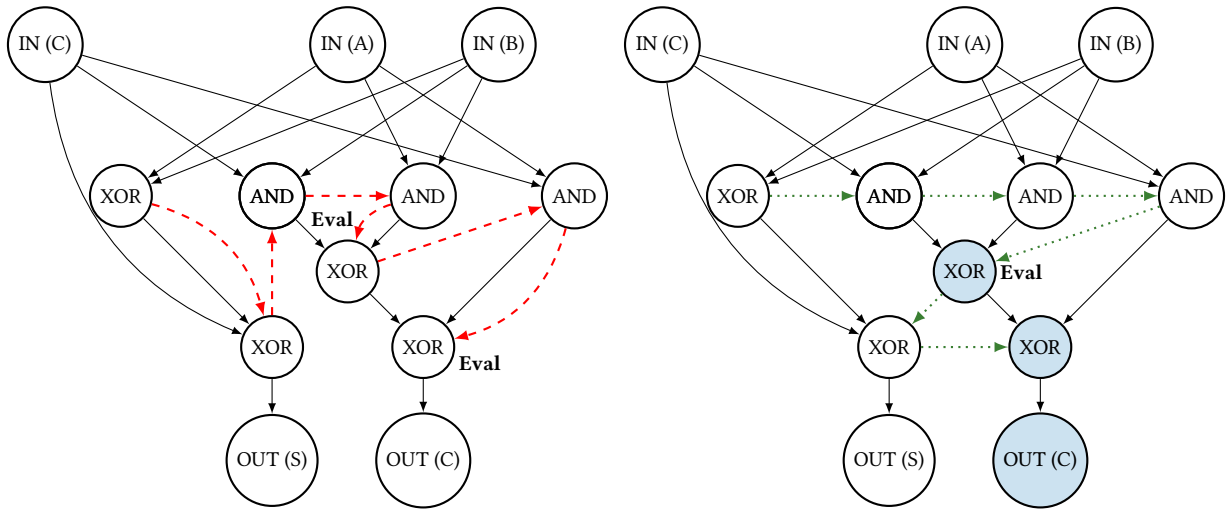

Figure 1: A simple 1-bit adder with different manually chosen gate orderings as an illustrative example for the freedom of topological ordering. Solid black arrows denote data dependencies, red dashed arrows denote one possible sub-optimal ordering in the left graph and green dotted arrows show a preferable ordering in the right graph. The "Eval" marks denote places where dynamic batching with free XORs would trigger processing of the queued fresh AND gates. All unfilled nodes are on the first layer in the right figure and all light-blue-filled nodes are on the second layer. A layer is defined to be the set of all nodes with the same amount of non-free (AND) gates between them and the input on the critical path.

4.2 Batch Computation

After one has identified a batch of independent AES calls, they need to be computed. For this, we have used two techniques: register-oriented computation, which focuses on performance and simplicity to the compiler, and memory-oriented computation, which focuses on modularity.

4.2.1 Register-Oriented Computation. Our primary technique for processing batches describes the task computations as low-level as possible without resorting to assembly. By using vector register types and constant-sized loops we give the compiler as many opportunities for optimization as possible while still allowing the conciseness of high-level code. Concretely, we have identified five steps executed continuously in a loop for all tasks.

1) Fill the appropriate lanes of the vector values with the task-specific data, both non-vector computable and loaded data, e.g., the lane 0 (the lowest 128 bit of the value) of all three virtual registers are assigned to gate 0, whereas lane 1 of all three is assigned to gate 1 and hold the input wire keys and a processed garbled table value. 2) Perform vectorizable operations on the input data, e.g., deriving computed inputs from loaded inputs with a global offset. 3) Perform the AES operations on the prepared inputs and keys with a sufficiently large batch size. 4) Execute vectorizable post-processing on the results and potentially other input values, e.g., XORing pairs of AES outputs as required by the scheme. 5) Do the remaining post-processing and scatter the data back to memory, e.g., handle operations that cannot be vectorized and where data needs to be extracted from the vectors first. Then, write the values back to the memory location where they are expected.

The cost of such a low-level approach is, of course, that not just the AES code needs to be re-written to satisfy the types of each used architecture and extension but also the immediately surrounding code leading to significant code duplication. An example implementation for HalfGate's [98] AND evaluation with fixed keys [13] and VAES is given in Listing 1 in Appendix A. Depending on the familiarity of the developer with the available platform instructions, their invocation, and the availability of validation methods, this register-oriented technique can be implemented within a few hours per optimized functionality.

4.2.2 Memory-Oriented Computation. Our memory-oriented technique addresses the code duplication concerns of the register-oriented one but can result in less performance. In particular, it only requires that a core primitive for this technique, e.g., electronic codebook mode, is implemented in an architecture-specific way. This core primitive is then used with a memory abstraction wherever needed while ensuring a sufficiently large number of AES calls for every invocation. The main loop for this only consists of three steps: 1) perform the data loading and preprocessing, 2) let the optimized library perform the operations, and 3) read the results using the memory abstraction and post-processing and store them. The pre-processing and post-processing steps for this approach can use platform-independent instructions lowering code duplication for handling a batch of gates at a time. However, this technique has performance overhead if implemented this way as implementing counter-mode can be significantly slower than with a dedicated implementation as the compiler might generate general-purpose 64-bit store instructions and adds from the abstract code. In contrast, a direct use of 64-bit vector additions might be significantly

Table 3: Overview of improved frameworks, used batch identification methods, and batch computation strategies used.

	Batch Identification (§ 4.1)		
Framework (§ 4.3)	Dynamic (§ 4.1.1)	Static (§ 4.1.2)	Computation (§ 4.2)
ABY [25] (§ 4.3.1)	Non-Free-XOR + SIMD	SIMD	Register-Oriented (§ 4.2.1)
EMP-OT [91] (§ 4.3.2)	—	—	Memory-Oriented (§ 4.2.2)
EMP-AGMPC [91, 93] (§ 4.3.3)	Regular-Early-Execution	—	Memory-Oriented (§ 4.2.2)
CrypTFlow2 [83] (§ 4.3.4)	—	—	Memory-Oriented (§ 4.2.2)

faster. An example implementation for EMP-AG2PC's [91, 92] AND evaluation with fixed keys [13] and VAES is given in Listing 2 in Appendix A. As this technique favors engineering efficiency over runtime efficiency, the required effort for its implementation is generally a few hours if a pre-existing implementation can be adapted and some form of batch identification has already been implemented.

4.3 Frameworks

To measure the performance impact of batching, VAES, and the above techniques we have applied them to the MPC frameworks and libraries ABY [25], EMP-OT [91], and EMP-AGMPC [91], and the PPML framework Microsoft CrypTFlow2 [83]. We will now briefly discuss our changes to each framework and library and provide an overview in Table 3.

4.3.1 ABY. We chose to use ABY [25] as it is a flexible, optimized framework for mixed-protocol secure two-party computation. For our modifications, we targeted the GC subcomponent of ABY which uses HalfGates garbling [98] with a fixed AES key [13] and invokes OpenSSL individually for every single AES operation used. We changed this fixed-key AES garbling, which we call "PRP" based on the public random permutation assumption used, to use a register-oriented computation. We furthermore added to ABY support for two more instantiations of the encryption functions in the HalfGates [98] garbling scheme: CIRC [98] is based on a circular security assumption and uses the wire keys as AES keys. MI [41] provides better multi-instance security and uses the wire key as the data input and the gate index as the AES key starting from a random offset. We note that these three schemes "PRP" / "CIRC" / "MI" need 0 / 4 / 2 computations of the AES key schedule to garble an AND gate respectively. Garbled circuit evaluation requires 0 / 2 / 2 key schedules per AND gate respectively. Neither the evaluation nor the garbling of XOR gates requires communication or AES operations with HalfGates.

Furthermore, we added an implementation of the PRF-based garbling scheme of Gueron et al. [40] which is secure in the standard model. It uses 8 AES operations with 4 keys for garbling an AND gate, 2 uniquely keyed operations for evaluating an AND, 3 uniquely keyed AES operations for XOR garbling, and 1-2 uniquely keyed AES operations for XOR evaluation. We identify batches using dynamic batching with support for SIMD gates and with support for two queues with shared indices for the PRF-based scheme. For all these four schemes, we implemented two register-oriented backends each for the batch processing: one using AES-NI and 128-bit operations, and another one using VAES and AVX512.

4.3.2 EMP-OT. We chose EMP-OT [91] because it is a state-of-the-art implementation for oblivious transfer and it is the underlying OT library for the two frameworks in § 4.3.3 and § 4.3.4 and other recent works [52]. We modified the main OT protocol implementations [8, 9, 57] by replacing the AES-NI based ECB and pseudo-random generator (PRG) implementations in the referenced EMP-Tool library [91] with VAES and widened the batch size from 8 to 16. Additionally, we widened the bit matrix transposition algorithm to use 512-bit AVX512 operations instead of 128-bit SSE operations. Finally, we changed the LPN-based FERRET OT [94] implementation to use VAES instead of AES-NI for selecting the matrix-vector multiplication entries.

4.3.3 EMP-AGMPC. The EMP-AGMPC [91, 93] framework provides a low-communication actively secure garbling scheme. For the implementation, we used a memory-oriented computation strategy mirroring the modular design of the EMP toolkit that strongly encourages modularity. We used basic dynamic batching with early execution for the online and preprocessing phases' circuit processing. In the corresponding EMP-OT library [91] which implements the actively secure OT extension of [9], we instantiate the PRG using VAES.

4.3.4 CrypTFlow2. Microsoft CrypTFlow2 [83] is a state-of-the-art framework for general PPML neural network inference. The implementation uses a sub-part of EMP-OT [91] for OT operations. We extended the modular implementation of CrypTFlow2 with VAES-based implementations for: 1) the 128-bit and 256-bit PRGs, 2) the AES-NI based ECB, and 3) the circular-secure correlation robust function in the garbling scheme of Gueron et al. [40].

5 EVALUATION

This section presents the benchmarking platform and the performance results we achieved for the frameworks from § 4.3.

5.1 Evaluation Platform

For all measurements, we use an Apple Macbook Pro with an Intel Core i7-1068NG7, 2x16GB of dual rank Samsung LPDDR4-3733 RAM (`K4UCE3Q4AA-MGCL`). It runs Arch Linux using the Linux 5.9.13.arch1-1 kernel along with GCC 10.2.0 and Clang 11.0.0 which

were used for compiling the code. For comparative AES-NI measurements we use the same machine.

5.2 ABY

For ABY (cf. § 4.3.1), we ran the benchmarks with both parties locally using a single sample per triple of circuit, scheme and implementation backend (reference, AES-NI, and VAES). For each measurement, the garbling times are taken from the logs of the party running the garbling operation and the data-input-dependent online time from the other party running the evaluation which are executed after each other in ABY. This is done to capture the pure computation time for garbling and evaluation. For the evaluation, we use circuits of AES (with 65× parallel SIMD), SHA-1 (with 512-bit input and 63× parallel SIMD), and for circuit-based private set intersection (PSI) the sort-compare-shuffle (SCS) circuit (1024 elements of 32-bits) [49], and circuit phasing (1024 elements per side of 32-bit, 3 hash functions, $\varepsilon = 1.2$, stash of size 1) [81]. For the summary in Table 4, we computed the geometric mean over the performance results of the four above circuits. The detailed measurements are given in Table 8 in Appendix B. The binaries were produced by GCC. We note a range of performance improvements from the use of batched execution of 67 - 161% and an additional 17 - 171% from the use of VAES. In particular, we observe better performance improvements from VAES for garbling schemes needing more cryptographic operations per gate, e.g., circularly secure computation (CIRC) benefits more than public-random permutation based computation (PRP) (cf. § 4.3.1).

Discussion. We make two key observations for the ABY benchmarks in Table 4: First, using batch sizes larger than one increases the throughput, as can be seen from the runtime decrease of the baseline reference (by 80-130%). Second, the use of VAES does increase performance further, more so in scenarios where more AES operations are done per gate, i.e., with the schemes not using fixed AES keys with HalfGates [13, 98]. Additionally, an investigation using a profiler showed a high miss-speculation rate for the AES-NI code using regular "if" branches with the condition depending on an unpredictable label bit. Therefore, the use of masking facilitated by AVX512 is a secondary factor contributing to performance as it does not invoke speculative execution miss-predicting the branch with 50% probability. Finally, we note the odd behavior that multi-instance secure computation (MI) is significantly slower than circular-secure computation (CIRC) for AES-NI during the evaluation even though they should be tied given that they perform similar AES operations. Concerning the impact of VAES beyond improving speculative execution behavior, we see performance increases of 27% (garbling) and 36% (evaluation) for fixed-key AES because the AES processing makes up only a somewhat small amount of processing time. The HalfGates variable-keyed schemes see a 47% (MI garbling), 43% (CIRC evaluation), and 57% (CIRC garbling) performance increase. PRF-based garbling schemes see the largest increase with 51% (garbling) and 75% (evaluation) due to a large amount of AES operations necessary, given that each AND gate garbling requires 8 AES operations, each AND evaluation 2, each XOR garbling 3, and each XOR evaluation at least 1.

5.3 EMP-OT

For oblivious transfers, we evaluated EMP-OT [91] (cf. § 4.3.2). We ran it single-threaded with 100 million OT operations computed on localhost. For the one-time base OT operations, that use public-key crypto, the default number of OT operations was used, and times were excluded from the throughput results. As base OT protocols, we use the protocol of Naor and Pinkas [75] for passive security assumptions and SimplestOT [23] for active security, except for FERRET OT [94] which uses its own base OT protocol. The library uses fixed-key AES for its PRG [13], the optimized version of [8] of the protocol by Ishai et al. [57] for passive security, and the variant by Asharov et al. [9] for active security.

In addition, we also measured the performance of FERRET-OT [94] as it is a protocol with very little communication after the initial base OTs. EMP-OT was compiled with Clang. The results are shown in Table 5. We note the range of performance improvements of 14.8 - 30.1% from the use of VAES. We also observe that the performance increase is particularly high for random OTs (R-OTs) which can be attributed to a lower amount of system interaction due to the reduced amount of communication for R-OTs.

Table 4: Geometric means of the run-times in milliseconds of ABY [25] for the evaluation of AES, SHA-1, SCS-PSI, and Phasing-PSI with the detailed parameters as described in § 5.2. "Ref" indicates the reference ABY implementation, AES-NI and VAES indicate batched implementations. Garbling scheme names are as introduced in § 4.3. Improv% shows the performance improvement of VAES over AES-NI.

		Garbling Scheme			
Operation	Impl.	PRP	MI	CIRC	PRF
Garbling	Ref [25]	110.6	—	—	—
	AES-NI	47.1	61.0	72.1	197.4
	VAES	37.0	41.3	46.0	130.3
	Improv%	**27.2%**	**47.5%**	**56.7%**	**51.5%**
Evaluation	Ref [25]	56.5	—	—	—
	AES-NI	31.1	59.8	41.3	103.3
	VAES	22.9	29.4	28.9	59.0
	Improv%	**36.1%**	**103.5%**	**43.0%**	**75.0%**

Discussion. From the OT performance data in Table 5, we see that AVX512 and VAES notably improve performance, by 20 - 30% for the EMP libraries' traditional OT implementation, which use VAES for the PRG and AVX512 for bit transposition. Additionally, we observe mild performance improvements of 16.6% for the FERRET protocols, mainly using AES to generate the random matrices in the core matrix-vector multiplication.

5.4 EMP-AGMPC

For EMP-AGMPC [91, 93] (cf. § 4.3.3), we ran SHA256 with three parties on localhost with binaries compiled with Clang. The runs were performed 11 times and then averaged. After the initial measurements, we decided to benchmark with batching applied and while

Table 5: Run-times in seconds of 10 million OTs for EMP-OT [91] before "Ref" and after implementation of VAES support. The functionalities are general OT (OT), Correlated OT (C-OT), and Random OT (R-OT). Improv% shows the performance improvement of VAES over AES-NI. Higher throughput is better.

			OT Functionality		
Security	Library	Impl	OT	C-OT	R-OT
Passive	EMP-OT IKNP [8, 57]	Ref [8, 57, 91]	0.35	0.20	0.33
		VAES	0.28	0.16	0.25
		Improv%	**20.0%**	**20.0%**	**24.2%**
	EMP-OT FERRET [94]	Ref [91, 94]	1.33	1.14	1.32
		VAES	1.13	0.99	1.09
		Improv%	**15.0%**	**10.4%**	**17.4%**
Active	EMP-OT ALSZ [9]	Ref [9, 91]	0.39	0.24	0.38
		VAES	0.32	0.19	0.29
		Improv%	**17.9%**	**20.8%**	**23.7%**
	EMP-OT FERRET [94]	Ref [91, 94]	1.38	1.2	1.37
		VAES	1.21	1.04	1.16
		Improv%	**12.3%**	**13.3%**	**15.3%**
	+ Random Choice	Ref [91, 94]	—	0.94	—
		VAES	—	0.80	—
		Improv%	**—**	**14.8%**	**—**

Table 6: Run-times in milliseconds for the evaluation of various parts of SHA256 in EMP-AGMPC [91, 93] (§ 5.4). The computation backend ("Comp. Backend") indicates the implementation strategy used. The evaluated parts are the one-time setup, the function-independent preprocessing, the function-dependent preprocessing, and the input-dependent online phase. The values in parenthesis show the performance improvement in percent over the reference. Lower run-times are better.

	Operation			
Comp. Backend	Setup	Function-Independent	Function-Dependent	Online
Ref [91, 93]	45.0	564.5	247.0	7.0
VAES	45.9 (**−2.1%**)	580.7 (**−2.8%**)	250.6 (**−1.4%**)	6.7 (**5.0%**)
Batched + VAES	45.4 (**−0.9%**)	453.0 (**24.6%**)	250.7 (**−1.5%**)	7.0 (**0.7%**)

Table 7: Geometric mean of run-times in seconds for CrypTFlow2 [83] inference (§ 5.5) using the SqueezeNetImgNet, SqueezeNetCIFAR, ResNet50, and DenseNet121 networks. Ring32-OT denotes the 32-bit ring-based implementation using OT. "Ref" indicates the reference implementation using AES-NI and VAES indicates our version using VAES. Improv% shows the performance improvement of VAES over AES-NI. Lower run-times are better.

		Sub-Operation						
Type	Impl	Convolution	Truncation	ReLU	MatrixMultiplication	BatchNormalization	MaxPool	Total
Ring32-OT	Ref [83]	96.5	30.7	9.6	94.0	15.6	3.7	126.8
	VAES	97.0	21.0	6.8	94.5	13.5	2.5	119.1
	Improv%	**−0.5%**	**46.5%**	**40.4%**	**−0.5%**	**15.9%**	**47.1%**	**6.5%**

using only a VAES-enabled library implementation of AES-ECB, the PRG, and the OT functionalities. The resulting performance numbers are shown in Table 6. In this table, the computation backend indicates the implementation strategy used, with the numbers in parenthesis being the performance improvements over the previous row.

Here, VAES allow to improve performance by up to 28%. The most substantial performance improvement is in the function-independent pre-processing phase. During that phase, the code uses additional garbling and evaluation techniques to prepare for the following phases based on the number of gates of the MPC function to be computed.

Discussion. The AGMPC performance data (in Table 6) shows substantial performance differences. The performance increase from VAES in the online phase stems from the OT used with the extra batching moving values out of registers again due to the gap between successive accesses. The most notable improvement is the 25% performance increase through batching in the function-independent preprocessing phase combined with VAES. This is because the garbling operations used in that phase benefit sufficiently from the batching, and there are not too many XORs sparsing out the AND gates and their memory.

5.5 CrypTFlow2

As CrypTFlow2 [83] (cf. § 4.3.4) uses EMP-OT internally, it is a natural target to investigate how the internal improvements benefit the overall performance of a more end-to-end application. As benchmarks we run inference for the SqueezeCIFAR, ResNet50, DenseNet121, and SqueezeNetImgNet networks. Each of these networks has its dedicated driver executable as usual for this application, was compiled using GCC and run via localhost with both parties on the same machine, to focus on the computational. The default settings used did utilize multiple load-intensive threads for both the client and the server, but had no noticeable impact on performance consistency.

A summary of the results using the geometric mean is given in Table 7 and the details are shown in Table 9 in Appendix B. Times below 1 second were omitted from the table.

Discussion. Table 7 shows that the VAES-based speed-up for the OT-based Ring32 implementation is 6.5% in total. The non-linear layers have particularly contributed to this improvement, with both the ReLU and MaxPool layers improving by over 40%. In particular, we observe no performance changes for the linear convolution and matrix multiplication steps for the Ring32 implementation. This is because these are primarily bound by the speed of the operating system interaction. We can also conceive that the performance improvement for the Ring32 implementation does stem from the relatively short focus on VAES during the operations.

6 CONCLUSION AND FUTURE WORK

In this work, we have shown how AES-NI and VAES can be used to speed up MPC protocols and applications, in particular for the case where operations are not known a priori.

Summary. We started with discussing how dynamic batching and its extensions and optimizations use deferred execution to provide better batches of AES calls to the hardware units. Next, we have discussed how more explicit measures in the code like SIMD gates and layering find batches of tasks with more invasive code modifications. Furthermore, we have discussed how to compute the batched calls using abstract pre- and post-processing and platform-specific AES computation in our memory-oriented computation strategy. Our alternative register-oriented strategy accepted code duplication for a low-level register value oriented code description that the compiler and the processor can execute well more easily. Following that, we applied these techniques to ABY [13, 25, 98], EMP-OT [91], EMP-AGMPC [91, 93], and Microsoft CrypTFlow2 [83]. For ABY we implemented additional garbled circuit variants [40, 41, 98] for comparison. We then evaluated the performance impact of the use of VAES and batching techniques. In ABY, these batching techniques have significantly increased performance without changing the hardware requirements. The use of VAES has yielded further significant performance improvements in ABY, EMP-OT, Microsoft CrypTFlow2, and some parts of EMP-AGMPC.

Future Work. Our research can be extended in multiple directions.

Improved Modelization. The techniques presented in § 4.1 and § 4.2 could be further improved. A more theoretical modelization and a more detailed analysis of the interaction with cache effects could yield valuable insights for future implementations.

Merging Register- and Memory-oriented Computation. Our computation techniques from § 4.2 require to make a manual choice between low code duplication, high performance, and clarity to the compiler. Further research could find techniques to automatically achieve low code duplication, high performance and clarity. For this, techniques from programming language and compiler research might be useful.

Further Applications in MPC. VAES and the other AVX512 extensions can be used to improve performance in further applications in MPC such as the most recent garbling schemes [10, 43, 86] that reduce communication (which is the main bottleneck in MPC) at the cost of more computation.

AVAILABILITY

The open source code of our changed VAES implementations is freely available under the permissive Apache license at https://encrypto.de/code/VASA.

ACKNOWLEDGMENTS

We sincerely thank Nir Drucker as well as Shay Gueron for contacting us with very helpful comments and pointers to the history of VAES which helped us to substantially improve our paper. Shay Gueron was the inventor of the concept, perceived usages and motivation, architecture, and microarchitectural implementation for vectorized AES in Intel processors when he was with Intel.

This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the Deutsche Forschungsgemeinschaft (DFG) — SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within ATHENE.

REFERENCES

[1] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J. Kusner, and Adrià Gascón. 2019. QUOTIENT: Two-Party Secure Neural Network Training and Prediction. In *CCS*.

[2] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. 2010. Breakthrough AES performance with intel AES new instructions. *White paper* (2010). https://www.intel.ua/content/dam/www/public/us/en/documents/white-papers/aes-breakthrough-performance-paper.pdf.

[3] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2015. Ciphers for MPC and FHE. In *EUROCRYPT*.

[4] MPC Alliance. 2020. *MPC Alliance*. https://www.mpcalliance.org/

[5] AMD. 2021. *AMD64 Architecture Programmer's Manual: Volumes 1-5*. Advanced Micro Devices, Inc. https://www.amd.com/system/files/TechDocs/40332.pdf.

[6] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, and Carlos Aguilar Melchor. 2017. BIKE: Bit Flipping Key Encapsulation. (2017).

[7] Arm. 2021. *Arm Architecture Reference Manual*. Arm Limited. https://documentation-service.arm.com/static/60119835773bb020e3de6fee?token=.

[8] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *CCS*.

[9] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2015. More Efficient Oblivious Transfer Extensions with Security for Malicious Adversaries. In *EUROCRYPT*.

[10] Tomer Ashur, Efrat Cohen, Carmit Hazay, and Avishay Yanai. 2021. A New Framework for Garbled Circuits. Cryptology ePrint Archive. https://eprint.iacr.org/2021/739.

[11] Mauro Barni, Massimo Bernaschi, Riccardo Lazzeretti, Tommaso Pignata, and Alessandro Sabellico. 2014. Parallel Implementation of GC-Based MPC Protocols in the Semi-Honest Setting. In *Data Privacy Management and Autonomous Spontaneous Security*.

[12] D. Beaver, S. Micali, and P. Rogaway. 1990. The Round Complexity of Secure Protocols. In *STOC*.

[13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. 2013. Efficient Garbling from a Fixed-key Blockcipher. In *S&P*.

[14] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. 2016. Optimizing Semi-honest Secure Multiparty Computation for the Internet. In *CCS*.

[15] Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. 2020. MP2ML: A Mixed-protocol Machine Learning Framework for Private Inference. In *ARES*.

[16] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-preserving Machine Learning. In *CCS*.

[17] Joan Boyar, Philip Matthews, and René Peralta. 2013. Logic Minimization Techniques with Applications to Cryptology. *Journal of Cryptology* 26 (2013).

[18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. 2019. Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation. In *CCS*.

[19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. 2019. Efficient Pseudorandom Correlation Generators: Silent OT Extension and More. In *CRYPTO*.

[20] Niklas Buescher and Stefan Katzenbeisser. 2015. Faster Secure Computation through Automatic Parallelization. In *USENIX Security*.

[21] Tom Caputi. 2016. ZFS-Native Encryption. OpenZFS Developer Summit.

[22] Sergiu Carpov, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. 2021. GenoPPML – A Framework for Genomic Privacy-preserving Machine Learning. Cryptology ePrint Archive. https://eprint.iacr.org/2021/733.

[23] Tung Chou and Claudio Orlandi. 2015. The Simplest Protocol for Oblivious Transfer. In *LATINCRYPT*.

[24] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. 2015. Automated Synthesis of Optimized Circuits for Secure Computation. In *CCS*.

[25] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY–A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*.

[26] Jintai Ding and Dieter Schmidt. 2005. Rainbow, a New Multivariable Polynomial Signature Scheme. In *ACNS*.

[27] Itai Dinur, Daniel Kales, Angela Promitzer, Sebastian Ramacher, and Christian Rechberger. 2019. Linear Equivalence of Block Ciphers with Partial Non-Linear Layers: Application to LowMC. In *EUROCRYPT*.

[28] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. 2018. Rasta: A Cipher with Low ANDdepth and few ANDs per Bit. In *CRYPTO*.

[29] Nir Drucker and Shay Gueron. 2019. CTR DRBG with Vector AES NI. Code: https://github.com/aws-samples/ctr-drbg-with-vector-aes-ni.

[30] Nir Drucker and Shay Gueron. 2019. Generating a Random String with a Fixed Weight. In *International Symposium on Cyber Security Cryptography and Machine Learning (CSCML)*. Springer.

[31] Nir Drucker and Shay Gueron. 2021. Speed Up Over the Rainbow. In *International Conference on Information Technology-New Generations (ITNG)*. Springer. Online: https://ia.cr/2020/408.

[32] Nir Drucker, Shay Gueron, and Dusan Kostic. 2020. QC-MDPC Decoders with Several Shades of Gray. In *PQCrypto*.

[33] Nir Drucker, Shay Gueron, and Vlad Krasnov. 2019. Making AES Great Again: The Forthcoming Vectorized AES Instruction. In *16. International Conference on Information Technology-New Generations (ITNG)*. Springer. Online: https://ia.cr/2018/392.

[34] Morris Dworkin. 2010. *Special Publication 800-38E: Recommendation for Block Cipher Modes of Operation: the XTS-AES Mode for Confidentiality on Storage Devices*. National Institute for Standards and Technology. https://csrc.nist.gov/publications/detail/sp/800-38e/final

[35] Intel. 2018. Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[36] J.O. Eklundh. 1972. A Fast Computer Method for Matrix Transposing. In *IEEE Transactions on Computers*.

[37] Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Helen Möllering, Thien Duc Nguyen, Phillip Rieger, Ahmad Reza Sadeghi, Thomas Schneider, Hossein Yalame, and Shaza Zeitouni. 2021. SAFELearn: Secure Aggregation for private FEderated Learning. In *DLS*.

[38] Agner Fog. 2021. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf.

[39] Thore Graepel, Kristin Lauter, and Michael Naehrig. 2013. ML Confidential: Machine Learning on Encrypted Data. In *ICISC*.

[40] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. 2015. Fast Garbling of Circuits Under Standard Assumptions. In *CCS*.

[41] Chun Guo, Jonathan Katz, Xiao Wang, Chenkai Weng, and Yu Yu. 2020. Better Concrete Security for Half-Gates Garbling (in the Multi-instance Setting). In *CRYPTO*.

[42] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. 2020. Efficient and Secure Multiparty Computation from Fixed-Key Block Ciphers. In *IEEE S&P*.

[43] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling. In *CRYPTO*.

[44] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. In *EUROCRYPT*.

[45] Aditya Hegde, Helen Möllering, Thomas Schneider, and Hossein Yalame. 2021. SoK: Efficient Privacy-preserving Clustering. *PETS* (2021).

[46] Tim Heldmann, Thomas Schneider, Oleksandr Tkachenko, Christian Weinert, and Hossein Yalame. 2021. LLVM-Based Circuit Compilation for Practical Secure Computation. In *ACNS*.

[47] Wilko Henecka and Thomas Schneider. 2013. Faster Secure Two-Party Computation with Less Memory. In *ASIACCS*.

[48] Wilko Henecka and Thomas Schneider. 2013. Faster Secure Two-Party Computation with Less Memory. In *CCS*.

[49] Yan Huang, David Evans, and Jonathan Katz. 2012. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?. In *NDSS*.

[50] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. 2011. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security*.

[51] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. 2014. Amortizing Garbled Circuits. In *CRYPTO*.

[52] Siam Hussain, Baiyu Li, Farinaz Koushanfar, and Rosario Cammarota. 2020. TinyGarble2: Smart, Efficient, and Scalable Yao's Garble Circuit. In *ACM Workshop on Privacy-Preserving Machine Learning in Practice (PPMLP)*.

[53] Siam U. Hussain and Farinaz Koushanfar. 2019. FASE: FPGA Acceleration of Secure Function Evaluation. In *FCCM*.

[54] Siam U. Hussain, Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar. 2018. MAXelerator: FPGA Accelerator for Privacy Preserving Multiply-Accumulate (MAC) on Cloud Servers. In *DAC*.

[55] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. 2013. GPU and CPU Parallelization of Honest-but-Curious Secure Two-Party Computation. In *ACSAC*.

[56] Intel. 2021. Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf.

[57] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. 2003. Extending Oblivious Transfers Efficiently. In *CRYPTO*.

[58] Kimmo Järvinen, Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. 2010. Embedded SFE: Offloading Server and Network Using Hardware Tokens. In *FC*.

[59] Kimmo Järvinen, Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. 2010. Garbled Circuits for Leakage-Resilience: Hardware Implementation and Evaluation of One-Time Programs. In *CHES*.

[60] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-Knowledge Using Garbled Circuits: How to Prove Non-Algebraic Statements Efficiently. In *CCS*.

[61] Daniel Kales, Léo Perrin, Angela Promitzer, Sebastian Ramacher, and Christian Rechberger. 2017. Improvements to the Linear Operations of LowMC: A Faster Picnic. Cryptology ePrint Archive. https://ia.cr/2017/1148.

[62] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. 2019. Mobile Private Contact Discovery at Scale. In *USENIX Security*.

[63] Emilia Käsper and Peter Schwabe. 2009. Faster and Timing-Attack Resistant AES-GCM. In *CHES*.

[64] Hannah Keller, Helen Möllering, Thomas Schneider, and Hossein Yalame. 2021. Balancing Quality and Efficiency in Private Clustering with Affinity Propagation. In *SECRYPT*.

[65] Ágnes Kiss, Jian Liu, Thomas Schneider, N Asokan, and Benny Pinkas. 2017. Private Set Intersection for Unequal Set Sizes with Mobile Applications. *PETS* (2017).

[66] B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. 2021. CrypTen: Secure Multi-Party Computation Meets Machine Learning. In *arXiv 2109.00984*.

[67] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP*.

[68] Ben Kreuter, Benjamin Mood, Abhi Shelat, and Kevin Butler. 2013. PCF: A Portable Circuit Format for Scalable Two-party Secure Computation. In *USENIX Security*.

[69] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow: Secure TensorFlow Inference. In *IEEE S&P*.

[70] Yehuda Lindell and Benny Pinkas. 2008. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology* (2008).

[71] Chris M. Lonvick and Tatu Ylonen. 2006. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253. https://rfc-editor.org/rfc/rfc4253.txt

[72] Microsoft and Contributors. 2019. *Transparent Data Encryption (TDE)*. https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption?view=sql-server-ver15

[73] Payman Mohassel, Mike Rosulek, and Ni Trieu. 2020. Practical Privacy-preserving K-means Clustering. *PETS* (2020).

[74] Jean-Pierre Münch, Thomas Schneider, and Hossein Yalame. 2021. VASA: Vector AES Instructions for Security Applications. In *ACSAC*.

[75] Moni Naor and Benny Pinkas. 2001. Efficient Oblivious Transfer Protocols. In *Society for Industrial and Applied Mathematics*.

[76] Moni Naor, Benny Pinkas, and Reuban Sumner. 1999. Privacy Preserving Auctions and Mechanism Design. In *ACM conference on Electronic commerce*.

[77] Jesper Buus Nielsen and Claudio Orlandi. 2009. LEGO for Two-Party Secure Computation. In *TCC*.

[78] Jesper Buus Nielsen and Claudio Orlandi. 2016. Cross and Clean: Amortized Garbled Circuits with Constant Overhead. In *TCC*.

[79] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved mixed-protocol secure two-party computation. In *USENIX Security*.

[80] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. SynCirc: Efficient Synthesis of Depth-OptimizedCircuits for Secure Computation. In *HOST*.

[81] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. 2015. Phasing: Private Set Intersection Using Permutation-based Hashing. In *USENIX Security*.

[82] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. 2009. Secure Two-Party Computation Is Practical. In *ASIACRYPT*.

[83] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-Party Secure Inference. In *CCS*.

[84] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://rfc-editor.org/rfc/rfc8446.txt

[85] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. 2019. XONN: XNOR-Based Oblivious Deep Neural Network Inference. In *USENIX Security*.

[86] Mike Rosulek and Lawrence Roy. 2021. Three Halves Make a Whole? Beating the Half-Gates Lower Bound for Garbled Circuits. In *CRYPTO*.

[87] Palash Sarkar. 2008. A General Mixing Strategy for the ECB-Mix-ECB Mode of Operation. In *Information Processing Letters*. https://www.sciencedirect.com/science/article/pii/S0020019008002652

[88] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. 2015. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *IEEE S&P*.

[89] Ebrahim M. Songhori, Shaza Zeitouni, Ghada Dessouky, Thomas Schneider, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2016. GarbledCPU: A MIPS Processor for Secure Computation in Hardware. In *DAC*.

[90] The OpenSSL Project. 2003. OpenSSL: The Open Source toolkit for SSL/TLS. www.openssl.org.

[91] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty Computation Toolkit. https://github.com/emp-toolkit.

[92] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation. In *CCS*.

[93] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-Scale Secure Multiparty Computation. In *CCS*.

[94] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. 2020. Ferret: Fast Extension for Correlated OT with Small Communication. In *CCS*.

[95] Andrew C. Yao. 1982. Protocols for secure computations. In *FOCS*.

[96] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *FOCS*.

[97] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. Cryptology ePrint Archive. https://eprint.iacr.org/2015/1153.

[98] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*.

[99] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. 2021. Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning. In *USENIX Security*.

A EXAMPLE CODE FOR OUR IMPLEMENTATION

We present example code for the register-oriented batch computation strategy from § 4.2.1 in Listing 1 and for the memory-oriented one from § 4.2.2 in Listing 2.

Listing 1: Register-oriented implementation of HalfGates's evaluation [13, 98] using fixed-key VAES and AVX512F.

```
template<size_t width>
inline void FixedKeyLTEvaluatingVaesProcessor::
    computeAESOutKeys(uint32_t tableCounter, size_t
    queueStartIndex, size_t simdStartOffset, size_t
    numTablesInBatch, const uint8_t* receivedTables) {
  constexpr size_t div_width = (width + 3) / 4; //
      ceiling division
  constexpr size_t num_buffer_words = std::min(width,
      size_t(4));
  constexpr size_t KEYS_PER_GATE_IN_TABLE = 2; //
      HalfGates needs 2 keys per gate in the garbled
      table

  static_assert((width < 4) || (width % 4 == 0));

  const __m512i ONE = _mm512_set_epi32(
    0, 0, 0, 1,
    0, 0, 0, 1,
    0, 0, 0, 1,
    0, 0, 0, 1);
  constexpr size_t offset = std::min(size_t(4) *
      KEYS_PER_GATE_IN_TABLE, width *
      KEYS_PER_GATE_IN_TABLE);
  const __m512i FULL_OFFSET = _mm512_set_epi32(
    0, 0, 0, offset,
    0, 0, 0, offset,
    0, 0, 0, offset);
  __m512i counter = _mm512_set_epi32(
    0, 0, 0, (tableCounter + 3) * KEYS_PER_GATE_IN_TABLE,
    0, 0, 0, (tableCounter + 2) * KEYS_PER_GATE_IN_TABLE,
    0, 0, 0, (tableCounter + 1) * KEYS_PER_GATE_IN_TABLE,
    0, 0, 0, (tableCounter + 0) * KEYS_PER_GATE_IN_TABLE)
        ;

  __m512i leftData[div_width];
  __m512i rightData[div_width];
  __m512i leftKeys[div_width];
  __m512i rightKeys[div_width];
  __m512i finalMask[div_width];
  uint8_t* targetGateKey[width];
  __m512i aes_keys[11];
  const uint8_t* gtptr = receivedTables + tableCounter *
      KEYS_PER_GATE_IN_TABLE * 16;

  for (size_t i = 0; i < 11; ++i) {
    __m128i temp_key = _mm_load_si128((__m128i*)(
        m_fixedKeyProvider.getExpandedStaticKey() + i *
        16));
    aes_keys[i] = _mm512_broadcast_i32x4(temp_key);
  }

  size_t currentGateIdx = queueStartIndex;
  uint32_t currentOffset = simdStartOffset;

  for (size_t i = 0; i < numTablesInBatch; i += width) {
// pre-processing
    for (size_t w = 0; w < div_width; ++w) {
      for (size_t k = 0; k < num_buffer_words; ++k) {
        const GATE* currentGate = m_gateQueue[
            currentGateIdx];
        const uint32_t leftParentId = currentGate->
            ingates.inputs.twin.left;
        const uint32_t rightParentId = currentGate->
            ingates.inputs.twin.right;
        const GATE* leftParent = &m_vGates[leftParentId];
        const GATE* rightParent = &m_vGates[rightParentId
            ];
        const uint8_t* leftParentKey = leftParent->gs.
            yval + 16 * currentOffset;
        const uint8_t* rightParentKey = rightParent->gs.
            yval + 16 * currentOffset;

        const __m128i leftParentKeyLocal =
            _mm_loadu_si128((__m128i*)leftParentKey);
        leftKeys[w] = mm512_insert_128(leftKeys[w],
            leftParentKeyLocal, k);
        const __m128i rightParentKeyLocal =
            _mm_loadu_si128((__m128i*)rightParentKey);
        rightKeys[w] = mm512_insert_128(rightKeys[w],
            rightParentKeyLocal, k);

        targetGateKey[4 * w + k] = currentGate->gs.yval +
             16 * currentOffset;

        const uint8_t lpbit = leftParentKey[15] & 0x01;
        const uint8_t lpbit11 = (lpbit << 1) | lpbit;
        const uint8_t rpbit = rightParentKey[15] & 0x01;
        const uint8_t rpbit11 = (rpbit << 1) | rpbit;

        __m128i finalMaskLocal = _mm_maskz_loadu_epi64(
            lpbit11, (__m128i*)gtptr);
        gtptr += 16;
        const __m128i rightTable = _mm_loadu_si128((
            __m128i*)gtptr);
        const __m128i rightMaskUpdate = _mm_xor_si128(
            rightTable, leftParentKeyLocal);
        finalMaskLocal = _mm_mask_xor_epi64(
            finalMaskLocal, rpbit11, finalMaskLocal,
            rightMaskUpdate);
        gtptr += 16;


```

```
        finalMask[w] = mm512_insert_128(finalMask[w],
            finalMaskLocal, k);

        currentOffset++;
        if (currentOffset >= currentGate->nvals) {
          currentGateIdx++;
          currentOffset = 0;
        }
      }
    }

    for (size_t w = 0; w < div_width; ++w) {
      // use this because addition has a latency of 1 and
          a throughput of 0.5 CPI
      leftData[w] = counter;
      rightData[w] = _mm512_add_epi32(counter, ONE);
      counter = _mm512_add_epi32(counter, FULL_OFFSET);
    }

    for (size_t w = 0; w < div_width; ++w) {
    // this is a 1-bit 128-bit left shift of the left
        input with a XOR of the right one
      leftData[w] = vaes_mix_keys(leftKeys[w], leftData[w
          ]);
      rightData[w] = vaes_mix_keys(rightKeys[w],
          rightData[w]);

      leftKeys[w] = leftData[w]; // keep as a backup for
          post-whitening
      rightKeys[w] = rightData[w]; // keep as a backup
          for post-whitening
    }
// AES processing
        for (size_t w = 0; w < div_width; ++w) {
        leftData[w] = _mm512_xor_si512(leftData[w],
            aes_keys[0]);
        rightData[w] = _mm512_xor_si512(rightData[w],
            aes_keys[0]);
    }

    for (size_t r = 1; r < 10; ++r) {
        for (size_t w = 0; w < div_width; ++w) {
            leftData[w] = _mm512_aesenc_epi128(leftData[w
                ], aes_keys[r]);
            rightData[w] = _mm512_aesenc_epi128(rightData
                [w], aes_keys[r]);
        }
    }

    for (size_t w = 0; w < div_width; ++w) {
        leftData[w] = _mm512_aesenclast_epi128(leftData[w
            ], aes_keys[10]);
        rightData[w] = _mm512_aesenclast_epi128(rightData
            [w], aes_keys[10]);
    }
// post-processing
    for (size_t w = 0; w < div_width; ++w) {
      leftData[w] = _mm512_xor_si512(leftData[w],
          leftKeys[w]);
      rightData[w] = _mm512_xor_si512(rightData[w],
          rightKeys[w]);
      leftData[w] = _mm512_xor_si512(leftData[w],
          rightData[w]);
      leftData[w] = _mm512_xor_si512(leftData[w],
          finalMask[w]);
    }

    for (size_t w = 0; w < div_width; ++w) {
      for (size_t k = 0; k < num_buffer_words; ++k) {
      // helper function to extract a 128-bit word from a
          4x128 bit vector
      // uses the dedicated instruction with a switch-
          case
        const __m128i extracted = mm512_extract_128(
            leftData[w], k);
        _mm_storeu_si128((__m128i*)(targetGateKey[4 * w +
            k]), extracted);
      }
    }
  }
}
```

Listing 2: Memory-oriented implementation of the batched AND evaluation for actively secure garbled circuits [91, 92].

```
// ONLINE_BATCH_SIZE is an upper bound
void EvaluateANDGates(uint8_t* mask_input,int indices[
    ONLINE_BATCH_SIZE], size_t num_gates, int& ands) {
int mask_indices[ONLINE_BATCH_SIZE];
block lefts[ONLINE_BATCH_SIZE],rights[ONLINE_BATCH_SIZE];
block H[ONLINE_BATCH_SIZE][2];
for (size_t ii = 0; ii < num_gates; ++ii) {
  // preprocessing
  int i = indices[ii];
  int index = 2 * mask_input[cf->gates[4 * i]] +
      mask_input[cf->gates[4 * i + 1]];
  mask_indices[ii] = index;
  lefts[ii] = labels[exec_times][cf->gates[4 * i]];
  rights[ii] = labels[exec_times][cf->gates[4 * i + 1]];
}
// AES processing
Hash(H, lefts, rights, indices, mask_indices, num_gates);
for (size_t ii = 0; ii < num_gates; ++ii) {
  // postprocessing
  int i = indices[ii];
  int index = 2 * mask_input[cf->gates[4 * i]] +
      mask_input[cf->gates[4 * i + 1]];
  GT[exec_times][ands][index][0] = GT[exec_times][ands][
      index][0] ^ H[ii][0];
  GT[exec_times][ands][index][1] = GT[exec_times][ands][
      index][1] ^ H[ii][1];
  block ttt = GTK[exec_times][ands][index] ^ fpre->Delta;
  ttt = ttt & MASK;
  GTK[exec_times][ands][index] = GTK[exec_times][ands][
      index] & MASK;
  GT[exec_times][ands][index][0] = GT[exec_times][ands][
      index][0] & MASK;
  if (cmpBlock(&GT[exec_times][ands][index][0], &GTK[
      exec_times][ands][index], 1))
    mask_input[cf->gates[4 * i + 2]] = false;
  else if (cmpBlock(&GT[exec_times][ands][index][0], &ttt
      , 1))
    mask_input[cf->gates[4 * i + 2]] = true;
  else
    cout << ands << "no match GT!" << endl;
  mask_input[cf->gates[4 * i + 2]] = logic_xor(mask_input
      [cf->gates[4 * i + 2]], getLSB(GTM[exec_times][
      ands][index]));
  labels[exec_times][cf->gates[4 * i + 2]] = GT[
      exec_times][ands][index][1] ^ GTM[exec_times][ands
      ][index];
  ands++;
}
```

B DETAILED MEASUREMENTS

We present the detailed performance measurements for ABY (cf. § 5.2) in Table 8 from which the summary in Table 4 was computed. Additionally, we present the detailed performance measurements for CrypTFlow2 (cf. § 5.5) in Table 9 from which the summary in Table 7 was computed.

Table 8: Run-times in milliseconds of ABY [25] for the evaluation of AES, SHA-1, SCS-PSI, and Phasing-PSI with the detailed parameters as described in § 5.2. "Ref" indicates the reference ABY implementation, AES-NI indicates the batched one using AES-NI and VAES the one using VAES. Improv% shows the performance improvement of VAES over AES-NI based PRGs and ECB implementations. Garbling scheme names are as introduced in § 4.3. Lower run-times are better.

			Garbling Scheme			
Operation	Circuit		PRP	MI	CIRC	PRF
Garbling	AES	Ref [25]	47.3	—	—	—
		AES-NI	20.5	27.6	31.3	98.5
		VAES	16.6	19.0	20.8	66.2
		Improv%	**23.4%**	**45.4%**	**50.4%**	**48.6%**
	SHA1	Ref [25]	236.7	—	—	—
		AES-NI	95.4	118.6	145.7	576.2
		VAES	69.8	79.3	87.9	378.3
		Improv%	**36.6%**	**49.6%**	**65.8%**	**52.3%**
	SCS-PSI	Ref [25]	153.0	—	—	—
		AES-NI	75.3	98.9	112.3	288.1
		VAES	63.9	74.2	79.7	192.7
		Improv%	**17.8%**	**33.3%**	**40.9%**	**49.5%**
	PSI-Phasing	Ref [25]	87.3	—	—	—
		AES-NI	33.4	42.6	52.7	92.9
		VAES	25.3	26.1	30.7	59.6
		Improv%	**31.8%**	**63.2%**	**71.6%**	**55.8%**
Evaluation	AES	Ref [25]	23.0	—	—	—
		AES-NI	12.5	23.1	15.6	47.9
		VAES	8.6	11.7	10.2	25.1
		Improv%	**45.0%**	**97.1%**	**53.4%**	**91.1%**
	SHA1	Ref [25]	108.8	—	—	—
		AES-NI	56.0	139.7	80.9	261.7
		VAES	38.2	51.5	52.3	151.3
		Improv%	**46.5%**	**171.5%**	**54.7%**	**73.0%**
	SCS-PSI	Ref [25]	76.2	—	—	—
		AES-NI	41.9	92.5	57.3	135.7
		VAES	33.1	43.5	41.9	78.0
		Improv%	**26.5%**	**112.7%**	**36.8%**	**74.1%**
	PSI-Phasing	Ref [25]	53.2	—	—	—
		AES-NI	31.9	42.7	40.1	66.9
		VAES	25.0	28.4	31.1	41.0
		Improv%	**27.5%**	**50.5%**	**28.7%**	**62.9%**

Table 9: Run-times in seconds for CrypTFlow2 [83] (§ 5.5) inference using the SqueezeNetImgNet (SqzImg), SqueezeNetCIFAR (SqzCIFAR), ResNet50, and DenseNet121 networks. Ring32-OT denotes the 32-bit ring-based implementation using OT. Ref indicates the reference implementation (using AES-NI) and VAES indicates the version using VAES. Improv% shows the performance improvement of VAES over AES-NI. Lower run-times are better.

			Sub-Operation						
Type	Network	Impl	Convolution	Truncation	ReLU	MatMul	BatchNormalization	MaxPool	Total
Ring32-OT	SqzImg	Ref [83]	28.1	—	4.0	27.2	—	4.7	39.0
		VAES	28.0	—	2.9	26.9	—	3.1	35.6
		Improv%	**0.6%**	**—**	**36.7%**	**0.9%**	**—**	**53.0%**	**9.6%**
	SqzCIFAR	Ref [83]	28.0	—	4.0	27.0	—	4.4	38.5
		VAES	28.2	—	2.9	27.2	—	3.2	35.8
		Improv%	**−0.8%**	**—**	**38.9%**	**−0.9%**	**—**	**37.1%**	**7.5%**
	ResNet	Ref [83]	439.7	30.8	18.7	436.1	12.7	3.2	513.3
		VAES	448.2	20.9	12.7	444.5	11.2	2.1	503.1
		Improv%	**−1.9%**	**47.5%**	**46.5%**	**−1.9%**	**13.2%**	**52.1%**	**2.0%**
	DenseNet	Ref [83]	250.1	30.6	28.6	244.3	19.2	2.7	335.6
		VAES	250.0	21.1	20.5	243.9	16.2	1.9	313.8
		Improv%	**0.1%**	**45.5%**	**39.5%**	**0.2%**	**18.6%**	**46.6%**	**6.9%**