VIP-Bench: A Benchmark Suite for Evaluating Privacy-Enhanced Computation Frameworks

Lauren Biernacki
University of Michigan
lbiernac@umich.edu

Meron Zerihun Demissie
Addis Ababa University
mdemissi@umich.edu

Kidus Birkayehu Workneh
Addis Ababa University
kworkneh@umich.edu

Galane Basha Namomsa
Addis Ababa University
ganamoms@umich.edu

Plato Gebremedhin
Addis Ababa University
plgizaw@umich.edu

Fitsum Assamnew Andargie
Addis Ababa University
fitsum.assamnew@aait.edu.et

Brandon Reagen
New York University
bjr5@nyu.edu

Todd Austin
University of Michigan
austin@umich.edu

***Abstract*—Privacy-enhanced computation enables the processing of encrypted data without exposing underlying sensitive information. Such technologies are extremely promising for the advancement of data privacy, as they remove plaintexts from the attackers' reach. However, each privacy technology provides varying degrees of computational capabilities and performance overheads, creating challenges for adoption. For example, some publicly available homomorphic encryption schemes are limited in expressiveness or cannot support deep computation without incurring significant overheads. This diversity warrants a benchmark suite that can adequately assess capability and performance while supporting a variety of privacy-enhanced software architectures. We propose VIP-Bench, a benchmark suite designed with varying operational complexity and computational depth to evaluate competing privacy frameworks fairly and directly. VIP-Bench defines a forward-looking privacy-enhanced computation model and then develops under that model an array of privacy-focused benchmarks. The benchmark set is designed to flexibly cover the whole range of expected computational power and capability, enabling VIP-Bench to evaluate the privacy-enhanced computation capabilities of both today and tomorrow.**

I. Introduction

Recently, there has been a concerted effort to improve users' rights for privacy on digital platforms. The combination of increased user concern and privacy legislation threaten the computational practices that the computing industry is built around: high-quality, personalized services that require access to intimate data (*e.g.*, deep learning). ***Privacy-enhanced computation (PEC)*** is a new computational paradigm that leverages advanced oblivious-computing cryptosystems to break the privacy-quality of service tradeoff. With PEC, computation can occur on encrypted data directly while producing the same results as the classic plaintext equivalent. Thus, PEC enables the processing of sensitive data values without exposing any underlying information to software. Currently, this form of execution is best embodied in the work of homomorphic encryption [1], where advanced cryptography is used to operate directly on encrypted ciphertext without a key.

Privacy-enhanced computation wrestles trust away from software, as software can no longer discern the sensitive data it is computing on. All software-accessible values are rendered as ciphertexts. As such, hacking into software provides no leverage in exposing always-encrypted sensitive data, stopping data breaches in their tracks. Moreover, *any* programs operating on sensitive data have no ability to expose that data, making it possible for software to operate on third-party data without the possibility of information exposure. Thus, PEC holds great promise for the future of computing, as it can provide users the same high-quality services integrated in our daily lives while ensuring strong privacy guarantees.

PEC's property of maintaining data confidentiality in the presence of ***zero software trust*** has the potential to create online services that cannot violate privacy. This potential, coupled with impending privacy legislation, has sparked significant research excitement in the field. Researchers are rushing to propose new solutions and optimizations that overcome the performance and usability challenges impeding the wide-scale deployment of existing PEC technologies. Specifically, many PEC technologies, like homomorphic encryption (HE) [1], remain too slow and complex for general use. Research on high-performance software libraries [2], [3], highly-optimized algorithms [4], accelerators [5], [6], and compilers [7] continues to lessen the burden of adoption. In some sense, this work to show that PECs can be made practical can be seen as the first wave of PEC research.

The successes of the first wave researchers and demands for increased privacy have established PEC systems research as a fast-growing field. However, during the first wave of research, robust experimental infrastructure was largely overlooked in favor of advancing as fast as possible. This is a natural course of action, since PEC is (still) new, and papers that explain what can and cannot be done are essential. We argue that, for many technologies, the field has now matured to a point where a common set of benchmarks is needed for commensurability. Additionally, we argue that forward-looking benchmarks are needed to sustain the current rate of innovation. Such a set of common and challenging benchmarks will not only improve PEC research today but also push the limits of PEC capabilities tomorrow.

In this paper we propose VIP-Bench—a benchmark suite to enable the commensurate evaluation of PEC research and challenge the status-quo of what today's technologies are

capable of computing. To support benchmark development, VIP-BENCH defines a central computation model to which all benchmarks adhere. The VIP computation model supports always-encrypted computation, as done by HE computation models, and also integrates the tenets of data-oblivious computing [8], [9]. This model assumes that sensitive data is always encrypted, including in registers and memory, and that these encrypted variables cannot be used to resolve branches or compute memory addresses. Data-oblivious computing permits the expression of non-linear functions, if-conditions, and a range of other computational patterns that have been traditionally difficult for early PEC frameworks to express. Thus, this computation model is forward-looking and challenging for most of today's PEC frameworks. To provide a suitable framework for specifying existing and future PEC applications for use with VIP-BENCH, we include a software-based implementation of our computation model within our distribution, as detailed in Section III.

Using our PEC computation model, VIP-BENCH provides 18 benchmarks that were selected to be representative of existing and emerging applications that would benefit from enhanced privacy. VIP-BENCH also provides a unified interface for different PEC capabilities, enabling all benchmarks to run from a single-sourced implementation. Thus, VIP-BENCH facilitates comparisons within and across PEC technologies, making it straightforward to measure and compare performance differences between solutions as well as against native, non-protected versions of the benchmarks. While VIP-BENCH provides a unified interface, it is not expected that every PEC framework supports all VIP workloads, as some PEC technologies have limited computational capabilities. Consequently, we have included benchmarks that run the gamut from simplistic and approachable workloads supported by all existing PEC frameworks to computationally complex and deep workloads that will challenge even the most capable of today's PEC technologies.

A. Contributions of This Work

VIP-BENCH aims to define a forward-looking PEC computation model and deliver a range of meaningful, privacy-enhanced benchmarks under that model. In this paper, we make the following contributions toward this overarching goal:

- We detail the development and release of the VIP-BENCH benchmark suite, which is accessible at https://bitbucket.org/vip-benchmarks/vip-bench.
- We describe our forward-looking VIP computation model based on existing HE computation frameworks and the tenets of data-oblivious computing.
- We detail a novel, unified programming interface for PEC that allows VIP-BENCH workloads to be built from a single-sourced implementation for a variety of PEC frameworks.
- We present an initial demonstrational study of the VIP-BENCH benchmarks, showing their performance overhead and memory requirements.

```
1  VIP_ENCDOUBLE
2  LeakyReLU(VIP_ENCDOUBLE x_enc)
3  {
4     VIP_ENCBOOL    cond_enc     = x_enc > 0;
5     VIP_ENCDOUBLE trueval_enc  = x_enc;
6     VIP_ENCDOUBLE falseval_enc = x_enc*0.01;
7
8     // Compute LeakyReLU Function using CMOV
9     VIP_ENCDOUBLE result_enc = VIP_CMOV(cond_enc,
10                                        trueval_enc,
11                                        falseval_enc);
12    return result_enc;
13 }
```

Fig. 1: **Encrypted Computation of Leaky ReLU.** This program computes a popular neural network nonlinear activation function, Leaky ReLU. To adhere to the VIP computational model, the program was modified to use encrypted data types and a conditional move in place of an if-statement. These types are provided by the VIP unified programming interface (Section III-C).

The remainder of this paper is organized as follows. Section II details our VIP privacy-enhanced computation model, under which all of the VIP-BENCH benchmarks are built. Section III presents the VIP workloads available in the current distribution. Section IV presents a demonstrational evaluation of the VIP-BENCH benchmarks, showing their performance and memory characteristics on native, non-protected platforms and on a few existing PEC frameworks. Finally, Section V presents background material and related work, and Section VI concludes the paper.

II. VIP COMPUTATIONAL MODEL

The VIP-BENCH framework assumes that the host system supports a privacy-enhanced computation (PEC) model that allows software to directly operate on always-encrypted data without having access to or exposing their plaintext values. Specifically, the VIP computation model assumes that the underlying PEC capability allows software to define encrypted variables and perform some subset of scalar or vector operations directly on encrypted variables. Additionally, this computation model requires data-oblivious programming as programs cannot make visible decisions on always-encrypted values. We assume that the host system preserves the secrecy of encrypted variables by restricting the operations on encrypted data to execute data-obliviously. We discuss these three requirements that constitute our computation model further in the following subsections.

VIP-BENCH integrates this computation model in a unified C++ interface used by each workload to define and operate on encrypted variables. This organization enables all VIP workloads to run off of a single-sourced implementation, streamlining the porting efforts required to make comparisons across different PEC technologies. This unified interface is described in Section III-C.

A. Defining Encrypted Variables

Always-encrypted computation frameworks enable programmers to define always-encrypted variables. Across PEC technologies, programmers typically have access to encrypted integers, encrypted floating point numbers, and encrypted Booleans. VIP-BENCH implements these type classes, as well as encrypted characters, in its programming interface. These data types are enumerated in Table I. The format and size of these types are dependent on the underlying PEC capability.

It is expected that some PEC frameworks will not support all of the encrypted data types included in VIP-BENCH. As a result, these frameworks will not be able to execute all of the workloads contained in VIP-BENCH. To maximize utility of the VIP-BENCH workloads, we have included a number of benchmarks that *only* use encrypted integer types, which all PEC platforms support in some form.

Figure 1 shows a privacy-enhanced implementation of the Leaky ReLU (Rectified Linear Unit) function in VIP-BENCH. Leaky ReLU is commonly used as an activation function in artificial neural networks. This code example uses both encrypted Boolean and encrypted double-sized floating point variables within its computation. The `VIP_ENCBOOL` and `VIP_ENCDOUBLE` symbols are defined in VIP-BENCH's unified C++ interface (Section III-C). During compilation, these symbols are replaced with the host system's encrypted Boolean and encrypted floating point types, respectively.

Type Class	VIP Data Types
Boolean	`VIP_ENCBOOL`
Character	`VIP_ENCCHAR`, `VIP_ENCUCHAR`
Integer	`VIP_ENCINT`, `VIP_ENCUINT`, `VIP_ENCINT64`, `VIP_ENCUINT64`
Floating Point	`VIP_ENCFLOAT`, `VIP_ENCDOUBLE`

TABLE I: **Encrypted Data Types.** VIP-BENCH implements the above type classes in its programming model.

B. Computation on Encrypted Variables

VIP-BENCH's computational model assumes that C++ operators can operate directly on encrypted variables in the same way that they operate on plaintext variables. Table II lists the operator support expected by VIP-BENCH, ordered by increasing computational complexity. Linear arithmetic operators are supported by all existing PEC frameworks, while nonlinear and conditional operators are only supported by a few frameworks. Some VIP-BENCH workloads only require linear arithmetic operators to accommodate this range of capabilities.

While encrypted variable operators are intended to operate on encrypted data types, they can also operate on plaintext variables and constants. The result of any operation utilizing an encrypted operator will *always* produce an encrypted result. For example, Line 6 of Figure 1 computes the operation $x_{enc} * 0.01$, which will execute as a multiply operator with one encrypted operand (x_{enc}) and one constant (0.01) that produces an encrypted result.

Operator Class	Example Semantics
Linear Arithmetic *e.g.,* `+, -, *`	`x = enc(dec(y) + dec(z))`
Nonlinear Arithmetic *e.g.,* `/, %`	`x = enc(dec(y) % dec(z))`
Nonlinear Relational *e.g.,* `==, >, >=, <, <=`	`x = enc(dec(y) < dec(z))`
Nonlinear Boolean *e.g.,* `&, |, ^, ~, &&, ||, !`	`x = enc(dec(y) & dec(z))`
Type Cast Operators *e.g.,* `(VIP_ENCINT)`	`x = enc((int)dec(y))`
Conditional *e.g.,* `VIP_CMOV(p, x, z)`	`x = enc(dec(p) ? dec(x) : dec(y))`
Control Flow	n/a
Memory Access	n/a

TABLE II: **Operator Classes for Encrypted Data Types.** VIP-BENCH assumes that each C++ operator listed in this table operates correctly on encrypted data types, as they operate on plaintext data types. Control flow and memory operations are not permitted on encrypted data types, as these operations could be used to discover their plaintext values.

Type cast operators enable the conversion from one encrypted data format (*e.g.*, integer, floating point, and Boolean) to any of the other encrypted types. The inputs to the cast operator can be either encrypted or plaintext. When the inputs are plaintext, this is a convenient mechanism to encrypt a plaintext value, *e.g.*, $x_{enc} = 42$. It is not possible to assign an encrypted cast operator to a plaintext result since this would require decrypting the encrypted value, which is not permitted.

Conditional operators, termed `VIP_CMOV`, enable programs to implement conditional logic in accordance with our assumption that computation is data oblivious (Section II-D). The `VIP_CMOV` operator returns one of two source operands depending on some encrypted Boolean condition. Both source operands are computed in their entirety prior to execution of the conditional move. In Figure 1, Line 10 will return `trueval_enc` when `cond_enc` is true, or `falseval_enc` when `cond_enc` is false.

C. Vector Computation on Encrypted Variables

Some privacy-enhanced computation frameworks, in particular homomorphic encryption frameworks, support vectorized execution directly on encrypted variables. Thus, these operations are assumed to be available in the underlying privacy-enhanced computation framework. If the underlying framework only supports scalar computation, the VIP-BENCH run-time will convert the vector accesses to comparable scalar computation sequences.

D. Encrypted Computation Must be Data Oblivious

Always-encrypted computation frameworks must operate data obliviously to preserve the secrecy of encrypted variables. Specifically, program execution must be completely oblivious of encrypted variables. Conditional branches and if-statements cannot utilize encrypted variables (*e.g.,* `if` (x_{enc} `> 0`)).

Such usage would create trivial avenues for an untrusted entity to discover plaintext values by simply observing program traces. Similarly, an encrypted variable cannot be used to compute a memory address. An encrypted value cannot be combined with a memory pointer (*e.g.*, added or subtracted) or used as an array index (*e.g.*, `a[`x_{enc}`]`). If such logic were permitted, the encrypted variable could be inferred by monitoring how the secret computation accesses memory.

VIP-BENCH assumes that the underlying PEC capability ensures data-oblivious computation by restricting the use of encrypted variables to influence program control flow and memory accesses. VIP-BENCH's computation model supports this assumption by not allowing encrypted data to influence if-statements or memory address computations.

Despite these restrictions, conditional logic and memory indexing can still be expressed using the available operators in VIP-BENCH. Conditionals must be expressed the encrypted ternary operator `VIP_CMOV`. This programming requirement results in the conversion of all if-statements to corresponding predication statements. Using this conditional execution pattern, the resulting code path is always invariant, and any decisions made are resolved in secret by the underlying implementation of `VIP_CMOV`. For example, consider the code in Figure 2a which counts the number of odd and even numbers. While this code is unsafe and reveals properties of encrypted variable `x`, it can be implemented in a safe manner using conditional operators, as shown in Figure 2b. By using two `VIP_CMOV` assignments with inverse conditions, the code only performs one increment of `odd` or `even`, depending on the value of `cond`. We assume that the underlying PEC framework utilizes high-entropy ciphers. Otherwise, it would be trivial to determine the value of `cond` by simply seeing whether `odd` or `even` changed. With high-entropy ciphers, the assignment of either variable to itself will result in re-encryption and a new unrecognizable ciphertext value.

```
if ((x & 1) == 1) {
    odd = odd + 1;
}

else {
    even = even + 1;
}
```

(a) Unsafe Conditional Logic.

```
VIP_ENCBOOL cond =
            (x & 1) == 1;
odd = VIP_CMOV(cond,
               odd+1,
               odd);
even = VIP_CMOV(!cond,
                even+1,
                even);
```

(b) Safe Conditional Logic.

Fig. 2: **Data Oblivious Program Transformations.** The VIP computational model requires programs to be data oblivious. To adhere to this requirement, if-conditions (2a) must be transformed to use the conditional move operators provided by VIP-BENCH (2b).

Furthermore, it is still possible to index an array with an encrypted variable by comparing each index to the encrypted variable, and then returning the value that has the matching index. Using this *private information retrieval* [10] approach, each entry of the array will be touched and the correct entry will be returned without revealing the secret index. The resulting code sequence that is executed remains invariant, and the `VIP_CMOV` operations will secretly select the correct array value. The pseudo-code in Figure 3 demonstrates privately accessing an array.

```
// Return a_enc[idx_enc] in a safe manner, or
// return -1 if not found
VIP_ENCINT
PrivateArrRead(VIP_ENCINT a_enc[], VIP_ENCINT idx_enc)
{
    VIP_ENCINT result_enc = -1;
    for (int i=0; i < MAX_INDEX; i++)
    {
       VIP_ENCBOOL pred_enc = (idx_enc == i);
       result_enc = VIP_CMOV(pred_enc,
                             a_enc[i],
                             result_enc);
    }
    return result_enc;
}
```

Fig. 3: **Data Oblivious Array Indexing.** While data-oblivious programming forbids memory accesses based on encrypted variables, programs can still use encrypted array indices by leveraging a private information retrieval approach [10], as demonstrated in the above pseudo-code.

The requirement of not utilizing encrypted variables in if-conditions or pointer computations incurs performance overheads during the execution of privacy-enhanced programs since data-oblivious execution generally results in more code executing. Currently, this requirement is unavoidable if confidentiality is to be maintained, as the program could be analyzed to infer the value of encrypted variables otherwise. A further implication of data-oblivious execution is that algorithms cannot build heuristics that utilize decisions made upon encrypted variables. A clear example of this can be seen in the VIP-BENCH benchmark *Bubble-Sort*. With native execution, *Bubble-Sort* can exit early when the array is detected to be fully sorted. But, when sorting encrypted variables, the algorithm must perform a worst-case execution of N^2 swaps (for N elements). VIP-BENCH includes native and data-oblivious variants of all workloads (see Section III-B) to enable developers to analyze the impact of these programming requirements on runtimes.

III. VIP-BENCH BENCHMARKS

Our approach to building the set of VIP-BENCH benchmarks was to: *i)* Identify existing programs or algorithms that could benefit from performing privacy-enhanced computation, *ii)* Locate the appropriate sensitive variable in these benchmarks, *iii)* Rewrite the code to utilize the VIP data types and PEC computational model, and *iv)* Integrate the benchmark with the VIP-BENCH unified programming and build interface.

Currently, VIP-BENCH includes 18 benchmarks that were selected to be representative of existing and emerging privacy-sensitive applications. These workloads are enumerated in Table III, alongside the operator classes required to execute each workload. The benchmarks are roughly listed from the easiest to execute to the most challenging to execute benchmarks, where more challenging benchmarks require more classes of

Benchmark	Description	Linear Arith.	Nonlinear Arith.	Nonlinear Rel.	Nonlinear Boolean	Cast Operators	Conditional Moves
Hamming-Distance	Computes distance between two equal length Boolean vectors by taking the logical AND and running a popcount on the result.	●	○	○	○	○	○
Dot-Product	Computes the distance between two integer vectors, the computation involves element-wise multiplication and a reduction across vector elements.	●	○	○	○	○	○
X-Gradient	Calculates the X-gradient of an image using a 3×3 filter of constants.	●	○	○	○	○	○
Linear-Regression	Computes the forward pass and loss for provided input and label pairs.	●	○	○	○	○	○
Poly-Regression	Similar to Linear Regression but fits a quadratic rather than a line.	●	○	○	○	○	○
Roberts-Cross	An edge detection kernel computed by sliding two 2×2 filters over an image, estimating X/Y-gradients. The gradients are each squared and summed using PEC, square root is left to post processing.	●	○	○	○	○	○
Eulers-Approx	Iterative approximation of Euler's constant (e)	●	○	○	○	○	○
Triangle-Count	Counts the number of triangles found in a graph, represented by an adjacency matrix	●	○	○	○	○	○
Mersenne	Implementation of Mersenne Twister (MT19937) to generate pseudorandom integers	●	○	○	●	○	○
Bubble-Sort	Sorts an array of 256 integers using bubble sort	○	○	●	○	○	●
Nonlinear-NN	Computation of ReLU and LeakyReLU activation functions for neural networks	●	○	●	○	○	●
Edit-Distance	Wagner–Fischer algorithm for computing Levenshtein edit distance for 8-character genomic sequences	●	○	●	○	○	●
FFT-Int	Performs a fixed-point fast Fourier transform (regular and inverse)	●	○	●	●	●	●
NR-Solver	Approximates roots of a real-valued function using the Newton-Raphson method	●	●	●	○	○	●
LDA	Performs a linear discriminant analysis	●	●	●	○	○	●
Kepler	Computes Kepler's equation for orbital bodies using different methods (*i.e.,* simple iteration, Newton's method, binary search, power series, and Fourier Bessel series)	●	●	●	○	○	●
Parrondo	Simulates trials of chance-based Parrondo's game and reports statistics on their outcomes	●	●	●	●	○	●
MNIST-CNN	7-layer CNN doing MNIST image recognition	●	●	●	●	●	●

TABLE III: **Enumeration of VIP-Bench Workloads.** VIP-BENCH constitutes 18 benchmarks that were selected to be representative of existing and emerging applications that would benefit from advanced privacy protections. The benchmarks are sorted roughly by increasing operational complexity, with the benchmark's required operator classes listed on the right-hand side. Some PEC technologies only provide support for linear arithmetic. For this reason, eight of our included workloads only use operators from this class (*i.e.,* `+, -, *`).

operators to enable their execution. In Table III, the first eight benchmarks only require linear arithmetic operations on integer variables. Thus, these benchmarks are appropriate for execution on any PEC framework, including existing HE frameworks. Benchmarks further down the table require more complex operators and support for deeper computation. Thus, these benchmarks may not be capable of executing on more limited PEC frameworks.

In developing VIP-BENCH, we drew upon existing open source applications wherever possible, and we ported them into the VIP framework (*e.g.*, *MNIST-CNN*). The first six benchmarks in Table III are implemented using the optimized schedules from Porcupine [7]. In other cases, we built the benchmarks from scratch, implementing both a native and protected version of the benchmarks (*e.g.*, *Triangle-Count*). To learn more about the provenance of individual benchmarks, please consult the VIP-BENCH source distribution, at https://bitbucket.org/vip-benchmarks/vip-bench

A. Workload Organization

As shown in Table IV, we organize our benchmarks across two axes: operation complexity and depth of computation. Such an organization is relevant to researchers and developers,

as some PEC technologies support limited operation classes and have restrained computation depth. Specifically, many software HE libraries only include support for linear arithmetic and nonlinear Boolean operations. Additionally, some HE libraries do not include support for bootstrapping [3]. Thus, computation depth is restricted by the scheme's allotted noise budget. By organizing our workloads across these two axes, we enable developers to easily determine which workloads are relevant to their PEC technology while challenging researchers to develop innovative ways to support the full VIP-BENCH benchmark suite.

1) Organization by Operation Complexity: We categorize workloads as having low operational complexity if they only use linear arithmetic or nonlinear Boolean operators when computing on encrypted data types. These operators are supported by a majority of existing PEC technologies. We categorize workloads as having high operational complexity if they utilize any other operator classes, as enumerated in Table II, including nonlinear arithmetic or nonlinear relational operators, like equality ($==$), greater than ($>$), or less than ($<$). Workloads with high operational complexity may use conditional operators (`VIP_CMOV`) to perform conditional arithmetic or to perform data-oblivious memory accesses, as illustrated in Section II-D.

2) Organization by Depth of Computation: Additionally, we characterize workloads by their computational depth or logic depth. We regard workloads as having shallow computational depth if their logical depth is no more than five operations (*i.e.,* there are five or fewer operations between source inputs and results). Similarly, we categorize workloads as having deep computation if they perform five or more operations on any encrypted variable. Some benchmarks exhibit especially deep computation, such as *Eulers-Approx*, which estimates Euler's constant (e) over a large number of trials.

	Low Operational Complexity	High Operational Complexity
Shallow Computation	*X-Gradient* *Linear-Regression* *Roberts-Cross*	*Nonlinear-NN*
Deep Computation	*Hamming-Distance* *Dot-Product* *Poly-Regression* *Eulers-Approx* *Triangle-Count* *Mersenne*	*Bubble-Sort* *Edit-Distance* *FFT-Int* *NR-Solver* *LDA* *Kepler* *Porrondo* *MNIST-CNN*

TABLE IV: **Workload Classification Matrix.** We classify the VIP-BENCH workloads by operational complexity and computation depth. Benchmarks of low operational complexity only use linear arithmetic and nonlinear Boolean operators to compute on encrypted variables, whereas benchmarks with high operational complexity use any of the operators listed in Table II. Benchmarks exhibiting shallow computation have a logic depth of no more than five operations, whereas benchmarks with deep computation have unrestricted depth.

B. Workload Modes of Operation

To enable commensurability and fine-grained analysis of overheads, VIP-BENCH provides three modes of operation: native (NA), data-oblivious (DO), and encrypted (ENC). The former two modes (NA, DO) use native C++ types and can be used as baselines for performance benchmarking. The latter mode (ENC) takes advantage of the VIP-BENCH unified interface to substitute native types with the PEC capability's encrypted types when appropriate.

*1) Native (*NA*):* Native mode is the original version of the benchmark where the algorithm is free to base execution off of any program state and does not need to adhere to data-oblivious restrictions. This mode uses native C++ types. Native mode provides the performance of the unsecured workload in its original, non-privatized form. PEC technologies should compare against this baseline to fully assess their overheads.

*2) Data-Oblivious (*DO*):* Data-oblivious mode similarly uses native C++ types but modifies the algorithm to adhere to data-oblivious restrictions. Specifically, benchmarks are reorganized to never use privatized variables for control decisions or memory accesses. Like native mode, data-oblivious mode does not utilize any encryption support from the underlying PEC capability. Data-oblivious mode can be used to assess the performance impact of the algorithmic changes that are necessary to accommodate data-oblivious execution.

*3) Encrypted (*ENC*):* Encrypted mode runs the benchmarks in its privatized mode with encrypted data types and data-oblivious execution supported by the underlying PEC capability. In this version of the benchmark, the software is not allowed to make decisions on secret values because they are always encrypted. This version represents the full extent of the performance impact of always-encrypted execution.

C. Unified PEC Programming Interface

VIP-BENCH provides a single unified programming interface between the benchmark implementations and the PEC capability. This interface allows a single-source version of the benchmark to support all three modes of execution (*i.e.*, NA, DO, and ENC) for a range of underlying PEC capabilities. When a researcher or developers wants to evaluate a new PEC execution capability, they simply need to "hook" their PEC variable and operator definitions into the unified PEC programming interfaces. The resulting build process will utilize their specific PEC capability without any changes to the benchmark source code.

Figure 4 shows the VIP unified programming interface for the data-oblivious (DO) mode of execution. In this mode, all VIP data types are native C++ types and the `VIP_CMOV` operator is implemented with the C++ ternary operator.

When running in encrypted (ENC) mode, the VIP unified programming interface should specify how to declare each of the encrypted variables. In addition, it is assumed that the underlying PEC capability will utilize C++ operator overloading to implement the necessary operator classes for encrypted scalar and vector variables. The `VIP_DEC` interface

```
#if defined(VIP_DO_MODE)

#define VIP_INIT
#define VIP_ENCBOOL     bool
#define VIP_ENCCHAR     char
#define VIP_ENCINT      int
#define VIP_ENCUINT     unsigned int
#define VIP_ENCUINT64   uint64_t
#define VIP_ENCFLOAT    float
#define VIP_ENCDOUBLE   double
#define VIP_DEC(X)      (X)
#define VIP_CMOV(P,A,B) ((P) ? (A) : (B))
...
```

Fig. 4: **VIP-Bench Unified Programming Interface.** This code snippet shows the VIP unified programming interface for data-oblivious (DO) mode. Benchmarks are implemented as a single source version using VIP's data types (*e.g.*, `VIP_ENCINT`). These types are then replaced by the types specified by the unified interface at compile time (*e.g.*, `int`).

is provided for benchmark validation purposes, and it must be able to decrypt values so that the program outputs can be displayed and checked. Typically, the underlying PEC capability will be configured with a known secret key, so that this decryption process is straightforward.

To aid in the VIP-BENCH workload development and verification, the distribution includes a reference PEC capability called the ***VIP Functional Library***. The VIP Functional Library implements the VIP PEC computation model fully, using a software-based implementation based on x86 AES-NI [11]. The x86 AES-NI instruction set extension is used to encrypt variables and accelerate crypto processing. While the VIP Functional Library does not fully achieve zero software trust, it does have some interesting security features. Namely, keys and plaintext for the always-encrypted variables never exist in memory. We consider the VIP Functional Library in our benchmark analysis.

IV. EXPERIMENTAL EVALUATION

In this section, we perform a demonstrational analysis of the VIP workloads, examining their performance and memory requirements. To demonstrate running VIP-BENCH with an underlying PEC capability, we evaluate a subset of the benchmarks using the Microsoft SEAL homomorphic encryption library [12] and the remainder using the VIP Functional Library due to the computational constraints of SEAL. We reserve the evaluation of competing homomorphic encryption libraries for future work.

A. Experimental Framework

All VIP workloads were compiled with GCC version 7.5.0 with compile-time options '-O2'. Experiments were run on a Intel Xeon Gold based system with a 2.60GHz clock and 252 GB of DRAM, running Ubuntu 18.04.5 LTS. All reported metrics were averaged over 100 runs of each benchmark.

For the HE experiments, denoted ENC-BFV and ENC-CKKS, we built the VIP benchmarks with the Microsoft SEAL library version 3.4 [12] using both the BFV and CKKS homomorphic encryption schemes. Currently, we only support 6 of the VIP-BENCH benchmarks with our SEAL interface. The remainder are run on the VIP Functional Library.

For the remaining workloads, we built the VIP benchmarks with the VIP Functional Library included in the VIP-BENCH distribution, denoted ENC-VIP. This library is packaged with the VIP benchmark suite and serves as a functional implementation of VIP's expected computation model, detailed in Section II. The VIP Functional Library uses x86 AES-NI [11] to perform "almost" always-encrypted computation. Encrypted variables are expressed as 128-bit values consisting of the native data type (*e.g.*, integer) padded to 64-bits, appended by a 64-bit true random salt value, and encrypted using the x86 AES-NI extensions [11]. When an encrypted variable is processed, it is loaded into a register, decrypted, operated on, and then re-encrypted. While the VIP Functional Library is primarily meant to be a reference implementation of the VIP computational model, it also has some useful security properties. The VIP Functional Library only exposes sensitive variables in the register file. Thus, when using this library, a program would not be subject to memory side channels (*e.g.*, Spectre or Meltdown). In addition, the library implements high-entropy ciphers by packing all data types with 64-bits of true random salt values, sufficiently diversifying sensitive variables in memory to thwart cryptanalysis attacks.

B. Performance and Memory Analyses

To measure performance, the VIP benchmarks were instrumented to measure their core computation times and instruction count, excluding the preparation of input/output data and the comparison of results against ground-truth outputs. Timing analyses is performed using the StopWatch class library [13], which performs timing measurements on a microsecond resolution. Instruction count was recorded using the Linux performance monitoring utility `perf_event` [14]. Memory usage was analyzed by reading each process's peak memory usage via the Linux process table status variable `VmPeak`. To reduce the effects of system noise in our performance measurements each experiment was performed 100 times, and the average run-time is reported.

Table V shows the measured performance for each of the VIP benchmarks, running in *native (NA)*, *data-oblivious (DO)*, and *encrypted (ENC)* mode when built with the Microsoft SEAL or VIP Functional Library. All run times are reported in microseconds (μs). Homomorphic encryption libraries, including SEAL, have limited support for complex operators and deep computations. Thus, some of the benchmarks could not be run with these frameworks. However, the VIP Functional Library is able to run all of the benchmarks, including the HE-friendly benchmarks.

Comparing the native (NA) executions to the data-oblivious (DO) executions, it is possible to see the cost of restricting software from seeing its secret data. These overheads range of negligible (*e.g., FFT-Int*) to sizeable (*e.g.*, *Bubble-Sort*, *Kepler*, and *Parrondo*). These inefficiencies arise because the

Benchmark	Mode	Insn. Count	Runtime (μs)		VSZ (kB)
Hamming-Distance	NA	571,746	258		14,160
	DO	571,746	258		14,160
	ENC-BFV	379,721,213	50,553	*(196x)*	135,692
	ENC-CKKS	203,778,393	30,038	*(116x)*	148,292
Dot-Product	NA	589,272	167		14,160
	DO	589,272	167		14,160
	ENC-BFV	213,441,935	30,089	*(180x)*	112,804
	ENC-CKKS	235,657,417	34,125	*(204x)*	148,292
X-Gradient	NA	35,688	4		14,160
	DO	35,688	4		14,160
	ENC-BFV	357,762,502	42,439	*(10,427x)*	112,292
	ENC-CKKS	555,413,370	65,828	*(16,174x)*	148,032
Linear-Regression	NA	556,518	149		14,160
	DO	556,518	149		14,160
	ENC-BFV	255,475,787	36,883	*(248x)*	130,048
	ENC-CKKS	132,301,587	21,769	*(147x)*	146,368
Poly-Regression	NA	1,269,371	354		14,296
	DO	1,269,371	354		14,296
	ENC-BFV	509,721,588	63,544	*(179x)*	132,676
	ENC-CKKS	212,743,858	31,197	*(88x)*	151,816
Roberts-Cross	NA	37,270	4		14,160
	DO	37,270	4		14,160
	ENC-BFV	705,305,953	81,173	*(18,490x)*	122,300
	ENC-CKKS	717,004,004	80,650	*(18,371x)*	154,828
Eulers-Approx	NA	40,027,740	21,347		14,012
	DO	40,027,740	21,439		14,012
	ENC-VIP	7,743,956,899	1,282,983	*(60x)*	14,028
Triangle-Count	NA	65,856	11		14,016
	DO	65,863	11		14,016
	ENC-VIP	8,648,695	4,360	*(411x)*	14,048
Mersenne	NA	902,318	806		14,016
	DO	902,316	784		14,016
	ENC-VIP	20,659,263	9,551	*(12x)*	14,044
Bubble-Sort	NA	532,805	241		14,016
	DO	548,660	264		14,016
	ENC-VIP	309,979,528	66,329	*(276x)*	14,032
Nonlinear-NN	NA	10,508,736	9,834		14,020
	DO	10,506,651	9,745		14,020
	ENC-VIP	18,810,869	11,834	*(1.2x)*	14,036
Edit-Distance	NA	145,459,282	72,033		14,350
	DO	145,459,283	72,156		14,349
	ENC-VIP	33,022,300,601	4,477,836	*(62x)*	14,365
FFT-Int	NA	163,633	46		14,020
	DO	163,651	46		14,020
	ENC-VIP	74,916,460	20,408	*(447x)*	14,060
NR-Solver	NA	341,605	233		14,012
	DO	346,015	247		14,012
	ENC-VIP	10,668,773	4,710	*(20x)*	14,028
LDA	NA	206,709	85		14,076
	DO	207,966	88		14,076
	ENC-VIP	73,534,227	20,159	*(236x)*	14,184
Kepler	NA	76,200	54		14,079
	DO	93,881	65		14,079
	ENC-VIP	18,895,701	8,108	*(150x)*	14,123
Parrondo	NA	25,047,453	9,535		14,012
	DO	143,643,026	24,058		14,012
	ENC-VIP	71,487,961,658	9,028,822	*(947x)*	14,036
MNIST-CNN	NA	205,331,153	48,438		21,132
	DO	205,521,747	48,383		21,136
	ENC-VIP	44,887,346,878	7,454,505	*(154x)*	36,552

TABLE V: **Performance and Memory Analyses.** *NA* and *DO* report the results for the native and data-oblivious variants of each workload running with native C++ data types. *ENC-BFV* and *ENC-CKKS* report the results for the benchmark running in encrypted mode with the support from the Microsoft SEAL libraries in BFV and CKKS mode, respectively. *ENC-VIP* reports the results for the benchmark running in encrypted mode with the VIP Functional library PEC capability. Timing analysis was amortized over 100 trials and reported in microseconds (μs). For encrypted modes, the performance overhead compared to native execution is also shown. Memory usage is presented as peak virtual memory size (VSZ) reported in kilobytes (kB).

algorithms in data-oblivious mode cannot utilize heuristics to improve their overall run times. For instance, in the case of *Bubble-Sort*, the algorithm cannot terminate early when it detects that all the elements are sorted (as it does for the native execution). This restriction forces the algorithm to complete a worst-case (N^2) number of compare-and-swap operations to ensure that any input set is properly sorted.

Comparing the data-oblivious (DO) and encrypted (ENC) executions highlights the cost of *i)* Operating directly on encrypted data and *ii)* Processing relatively larger ciphertext variables. These overheads are generally very high, ranging to over 18,000$\times$ for *Roberts-Cross*. It is interesting to compare the ENC-mode executions utilizing the VIP Functional Library to the HE-based executions. With the VIP Functional framework, slowdowns are on the order of 1-1,000$\times$, while the HE frameworks have slowdowns in the range of 200-20,000$\times$. While it isn't entirely fair to compare these two frameworks given that the security assurances of HE far outstrip the protections of the VIP Functional framework (which utilizes register-based AES operations), these results clearly show that HE homomorphisms are significantly more expensive than AES bulk cipher operations.

Table V also lists the memory overheads associated with each benchmark, reported as the peak virtual memory size in kilobytes (kB). Memory overheads vary from moderate (*e.g.*, Mersenne) to large (*e.g.*, *MNIST-CNN*) to massive (*e.g.*, *Linear-Regression* and *Roberts-Cross*). These overheads are reflective of the increase in data sizes due to added encryption protections. In general, a high-entropy ciphertext values will increase the size of the data type by at least as much as the true random salt added to all values before encryption. In the case of the VIP Functional Library, all data values are paired with a 64-bit true random salt value. For HE frameworks, it is expected for these overheads to grow acutely, since HE ciphertexts can become very large.

V. Background and Related Work

A. Background

VIP-Bench is interested in evaluating PEC technologies that enable oblivious computing. The programming interface currently supports data-oblivious computing and homomorphic encryption. We briefly introduce these technologies below and conclude by listing alternative PECs we plan to support in the future.

1) Data-Oblivious Computing: Prior work has demonstrated that data-oblivious computing provides programs significant immunity from side channels [8], [9], [15]–[18]. For example, Kocher's classic timing attack [19] exploits conditionals that branch on sensitive key data. Such conditionals are not permitted in data-oblivious programming frameworks. Similarly, data-oblivious computing prohibits sensitive data from being used in the creation of memory addresses. Removing data-dependent conditionals makes the system much more challenging to program, as in the case of HE. Thus, these systems often introduce conditional move predication, as in VIP-Bench, to permit control decisions on sensitive data

without introducing side channels. Furthermore, to alleviate restrictions on memory addressing, approaches have proposed using ORAM to sufficiently diffuse memory accesses [9]. Recent work has took to integrating encryption alongside data-oblivious computing [20], which moves towards supporting always-encrypted computation on top of the data-oblivious programming model.

2) Homomorphic Encryption: Homomorphic encryption (HE) is a form of encryption that enables computation directly on encrypted data. Since Gentry's initial construction [1], improvements have been made to significantly reduce computational overheads. There are two general classes of HE used to compute on integer/fixed-point or boolean data types. BFV [21] and CKKS [22] are (arguably) the most commonly used integer schemes. They have slight differences but provide the same set of operators, support vectors as native data types, and use noise-based encryption. Boolean HE schemes are still rapidly developing and gaining lots of traction (*e.g.*, TFHE). Although VIP-BENCH currently supports only integer schemes, we plan to explore integration with Boolean frameworks in future work. Thus, whenever we say HE, it is assumed to be integer HE. Below, we provide an HE primer to help readers understand this paper. See the above referenced papers for complete details on individual schemes.

Modern HE schemes all use large vectors of modular integers as the fundamental data type. Vectors typically range in size from 1024 to 128k elements and each element uses an encrypted data type of a dozens to thousands of bits. Vector length and data type parameter selection has complex impacts on security and performance, and prior work has proposed methods for automatically tuning them [2], [5]. A practical benefit of vector ciphertexts is it enables users to pack (roughly) one plaintext data into each HE vector element, or slot. Therefore, even though the vectors are large and slow to compute on, vector packing can offer significant throughput improvement without affecting latency. Both BFV and CKKS support element-wise vector addition and multiplication as well as vector slot rotation, which can be used to re-align vector elements within a vector.

In HE, encryption involves adding noise to plaintext data. Each time an operation is applied to a ciphertext the added noise compounds. Therefore, each freshly encrypted ciphertext has a noise budget that limits the computational depth of a function a ciphertext can compute before exceeding the budget. When the noise budget is exceeded, decryption fails, returning random results and rendering the computation ineffectual. Noise is commonly dealt with be either allocating a sufficiently large budget by increasing the ciphertext data type length, or by bootstrapping, where the noise is reset using a form of homomorphic decrypt/re-encrypt. Bootstrapping is generally considered prohibitively expensive, and most solutions try to limit computation depth and allocate sufficient noise budget for the full computation.

3) Additional PEC technologies: Data-oblivious computing and HE were selected as examplar PEC technologies for designing VIP-BENCH, but there exist many more.

Secure Multiparty Computation (MPC) is a way for multiple parties to jointly compute a function without either party learning the others' input values. There are generally two classes of MPC: secret sharing and garbled circuits.

Secret sharing can work over integer (called arithmetic secret sharing) or Boolean data types. It works by dividing private inputs into shares and blinding raw values by adding random numbers. Blinded secrets can then be shared with involved parties to compute functions obliviously. Once each party finishes computing, the results can be re-shared and summed to produce the output result.

Garbled circuits [23] support arbitrary computation by expressing all inputs as binary and all functions as logic gates. It is a two party protocol where one party (garbler) generates a representation of the function to be computed as a series of Boolean tables and the other (evaluator) executes the tables using encrypted representations of both inputs, which are often called labels. Each gate in the circuit represents one Boolean gate and requires several decryptions to evaluate. In the future, we hope to support both secret sharing and garbled circuits in VIP-BENCH.

Trusted Execution Environments (TEE) reduce the risk of a program getting hacked by walling off the execution of trusted software from the rest of the system (including the operating system and drivers) through encryption and selective isolation. Examples of these technologies include Intel SGX [24] and ARM TrustZone [25]. Using TEEs, developers should only have to worry about the security and integrity of software running inside the TEE, ignoring the rest of the system's vulnerabilities. In practice, the approach falls short in two important ways: *i)* Software remains inside of the TEE, which can be hacked, and *ii)* Trusted data resides in the same resources utilized by the main core's software, leading to resource sharing that enables side-channel attacks to exfiltrate secrets out of TEEs. As such, TEEs provide better security guarantees than native systems, but fall short of solving data security challenges, as they are plagued by a wide variety of effective attack scenarios. While it would be possible to implement a VIP computational model using a TEE, it would not be possible to achieve zero software trust using such an approach.

B. Related Work

1) Homomorphic Encryption: **Software frameworks:** The major frameworks implementing HE include SEAL [3], HEAAN [26], Palisade [2], and HElib [27]. Each of these PEC frameworks should be capable of executing a sizeable fraction of the VIP-BENCH benchmarks. Currently, we provide support for SEAL, and we look forward to expanding this pallete in the future.

Compilers: A major challenge when using HE is handling noise growth, packing the large vectors, and reasoning about rotations. Prior work has employed compilers as an effective way to alleviate these challenges. EVA [28] and Alchemy [29] automate the selection of HE parameters for a given computation, tailoring a noise budget to the needs of specific

applications. Porcupine [7] and RAMPARTS [30] are compilers for generating HE programs from a more user-friendly representation. RAMPARTS uses Julian as a native language and directly translates the encoded program. Porcupine proposes a domain-specific language and uses program synthesis to generate noise-latency optimal HE programs. Significantly more attention has been given to compiling programs for Boolean HE schemes, including Cingulata [31], E3 [32], and Google's transpiler [33]. Another avenue of research has been application-specific support, which has primarily focused on neural networks [34], [35].

Hardware: While compiler support paves the way for HE's mainstream adoption, the primary reason for its limited use are its extreme performance overheads. This has lead to the proposal of many novel hardware accelerators to speed up HE. HEAX [6] proposes using FPGAs to speed up ciphertext-ciphertext multiplication, achieving nearly two orders of magnitude improvement over a CPU. In [36], authors also speed up HE operators using FPGAs and report 13× speedup. GPUs have also been a popular platform for accelerating HE computation. Many have ported the core HE kernels, namely number theoretic transform (NTT), to run on GPUs and typically report speedups ranging from one to two orders of magnitude compared to CPU implementations [37]–[39].

While FPGAs and GPUs provide significant speedup, even 100× speedup is insufficient as HE can be as much as six orders of magnitude slower than plaintext execution. Understanding these performance needs, another line of research has developed ASICs tailored to HE. Cheetah [5] proposes a large, GPU-scale ASIC to speedup CNN inference with HE. The authors focus on CNNs as a case study to understand whether HE can be processed in near real time using large custom hardware. They conclude that the extreme degrees of parallelism found in HE and deep learning can be leveraged in custom logic to bring CNN inference within a small constant factor of real-time inference. Sapphire [40] develops an accelerator for client-side encryption, proposing low-level hardware design optimizations for highly-efficient lattice-based encryption. CHOCO-TACO [41] also focuses on the client-side costs of HE. The authors develop communication optimizations to alleviate data exchange and custom hardware for substantial energy and performance improvements.

VI. Conclusion

In this paper, we presented the VIP-Bench benchmark suite, a collection of 18 privacy-enhanced benchmarks suitable for evaluating the capabilities and overheads of a wide range of privacy-enhanced computation (PEC) frameworks. The benchmark set is built on the VIP computational model, which defines a forward-looking and challenging PEC model that implements zero software trust. VIP-Bench utilizes a unified programming interface and single-sourced implementation to facilitate the porting of workloads to the benchmark suite and the porting of the suite to new PEC capabilities. Furthermore, VIP-Bench includes the VIP Functional Library, a software implementation of the VIP computational model, to assist in the development and validation of new benchmarks.

VIP-Bench aims to define a forward-looking computational model and deliver a range of meaningful benchmarks under that model. We did not set out to create a benchmark suite that all existing PEC capabilities could support—quite the contrary. The limited capabilities of existing PEC frameworks, which often limit computation complexity and depth, create an often too-large barrier to entry for applications that could truly benefit from these groundbreaking privacy technologies. Rather, our goal was to create a benchmark suite that would become a stretch goal for many existing PEC capabilities. It is our true hope that the VIP-Bench will facilitate the development of future PEC frameworks with improved computation capabilities and performance characteristics.

To learn more about VIP-Bench and to access the latest distribution, please go to: https://bitbucket.org/vip-benchmarks/vip-bench

Acknowledgements

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semi- conductor Research Corporation (SRC) program co-sponsored by DARPA. The authors would particularly like to thank the ADA Center sponsors and task liaisons for their ongoing input on this project.

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: ACM, 2009, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1536414.1536440

[2] "PALISADE Lattice Cryptography Library (release 1.11.3)," https://palisade-crypto.org/, May 2021.

[3] "Simple Encrypted Arithmetic Library (release 2.3.1)," http://sealcrypto.org, Jun. 2018, microsoft Research, Redmond, WA.

[4] J. H. Cheon, M. Kim, and K. Lauter, "Homomorphic computation of edit distance," in *International Conference on Financial Cryptography and Data Security*. Springer, 2015, pp. 194–212.

[5] B. Reagen, W. Choi, Y. Ko, V. T. Lee, G. Wei, H. S. Lee, and D. Brooks, "Cheetah: Optimizations and methods for privacypreserving inference via homomorphic encryption," *CoRR*, vol. abs/2006.00505, 2020. [Online]. Available: https://arxiv.org/abs/2006.00505

[6] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1295–1309. [Online]. Available: https://doi.org/10.1145/3373376.3378523

[7] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: A synthesizing compiler for vectorized homomorphic encryption," *CoRR*, vol. abs/2101.07841, 2021. [Online]. Available: https://arxiv.org/abs/2101.07841

[8] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 431–446.

[9] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher, "Data oblivious ISA extensions for side channel-resistant and high performance computing." in *NDSS*, 2019.

[10] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 1995, pp. 41–50.

[11] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, "Breakthrough AES performance with Intel AES new instructions," *White paper, June*, p. 11, 2010.

[12] "Microsoft SEAL (release 3.6)," https://github.com/Microsoft/SEAL, Nov. 2020, microsoft Research, Redmond, WA.

[13] CPlusPlus, "chrono," https://www.cplusplus.com/reference/chrono/, 2000.

[14] M. Kerrisk, "Linux man pages: perf_event_open," https://man7.org/linux/man-pages/man2/perf_event_open.2.html, 2021.

[15] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 283–298.

[16] S. Eskandarian and M. Zaharia, "An oblivious general-purpose SQL database for the cloud," *CoRR, abs/1710.00458*, vol. 6, pp. 93–94, 2017.

[17] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious memory primitives from Intel SGX." *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 549, 2017.

[18] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 279–296.

[19] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology*, vol. 96, 1996, pp. 104–113.

[20] E. Chielle, N. G. Tsoutsos, O. Mazonka, and M. Maniatakos, "Encrypt-everything-everywhere: ISA extensions for private computation," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2020.

[21] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[22] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers." in *ASIACRYPT (1)*, ser. Lecture Notes in Computer Science, T. Takagi and T. Peyrin, Eds., vol. 10624. Springer, 2017, pp. 409–437. [Online]. Available: http://dblp.uni-trier.de/db/conf/asiacrypt/asiacrypt2017-1.html#CheonKKS17

[23] A. C.-C. Yao, "How to generate and exchange secrets," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE, 1986, pp. 162–167.

[24] Intel, "Software guard extensions programming reference," *Intel Corporation*, 2014.

[25] T. Alves, "Trustzone: Integrated hardware and software security," *White paper*, 2004.

[26] SNU-Crypto, "HEAAN," https://github.com/snucrypto/HEAAN, 2016.

[27] S. Halevi and V. Shoup, "Algorithms in HElib," in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 554–571.

[28] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '20, 2020. [Online]. Available: https://doi.org/10.1145/3385412.3386023

[29] E. Crockett, C. Peikert, and C. Sharp, "Alchemy: A language and compiler for homomorphic encryption made easy," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243828

[30] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "RAMPARTS: A programmer-friendly system for building homomorphic encryption applications," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC'19, 2019. [Online]. Available: https://doi.org/10.1145/3338469.3358945

[31] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: a compilation chain for privacy preserving applications," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, 2015, pp. 13–19.

[32] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling C++ programs with encrypted operands," Cryptology ePrint Archive, Report 2018/1013, 2018, https://eprint.iacr.org/2018/1013.

[33] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak, P. Schoppmann, S. Kulankhina, A. Forget, D. Marn, C. Tew, R. Misoczki, B. Guillen, X. Ye, D. Kraft, D. Desfontaines, A. Krishnamurthy, M. Guevara, I. M. Perera, Y. Sushko, and B. Gipson, "A general purpose transpiler for fully homomorphic encryption," 2021.

[34] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '19, 2019. [Online]. Available: https://doi.org/10.1145/3314221.3314628

[35] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "NGraph-HE2: A high-throughput framework for neural network inference on encrypted data," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC'19, 2019. [Online]. Available: https://doi.org/10.1145/3338469.3358944

[36] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 387–398.

[37] S. Akleylek, O. Dağdelen, and Z. Y. Tok, "On the efficiency of polynomial multiplication for lattice-based cryptography on GPUs using CUDA," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, Conference Proceedings, pp. 155–168.

[38] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.

[39] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, Conference Proceedings, pp. 169–186.

[40] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *arXiv preprint arXiv:1910.07557*, 2019.

[41] M. van der Hagen and B. Lucia, "Practical encrypted computing for IoT clients," *CoRR*, vol. abs/2103.06743, 2021. [Online]. Available: https://arxiv.org/abs/2103.06743