Abstract—The breakthrough of fully homomorphic encryption (FHE) enables privacy-preserving arbitrary computation in the cloud, supporting both addition and multiplication over encrypted data. Current FHE implementations, however, suffer from high performance overheads and require expensive bootstrapping operations to decrease ciphertext noise. In this work, we discuss how homomorphic encryption primitives can implement a functionally complete set of homomorphic operations and enable arbitrary computation that is outsourced by a single party. We focus on obfuscated computation with or without special look-up tables, to enable branch decisions over encrypted values while preserving privacy. Since partial homomorphic encryption is orders of magnitude less expensive than FHE, it can be more practical for privacy-preserving applications in the cloud.

I. INTRODUCTION

Fully homomorphic encryption (FHE) has been described by researchers as the “holy grail” of cryptography, as it would allow evaluation of arbitrary functions over encrypted data [1]. Gentry’s breakthrough in 2009 made it theoretically possible to apply two orthogonal operations directly over ciphertexts [2]. One side effect of all known FHE cryptosystem variants is the increase of the ciphertext noise after applying homomorphic operations. If this noise increases above specific thresholds, recovery of the original plaintext using the decryption operation would yield incorrect results. Gentry also introduced the notion of bootstrapping, which allows reducing the noise of ciphertexts by homomorphically evaluating the decryption algorithm in the encrypted domain to essentially refresh the ciphertexts. This remarkable observation allows applying an arbitrary number of homomorphic operations over encrypted data while enabling correct decryptions of the final result.

The ability to evaluate arbitrary functions on the data enables an entire family of confidentiality applications, where the evaluator of a function can be different from the encrypted data provider. In such applications, the function evaluator gets no information whatsoever about the data or the evaluation result. In effect, this allows one party to outsource the computation of encrypted data to a third party, while preserving the confidentiality of the processed data (i.e., private outsourcing). Unsurprisingly, one particularly interesting application of privacy-preserving outsourced computation is cloud computing [3]. In this setting, users can encrypt their data and request remote evaluation of arbitrary algorithms over their data by a cloud provider. The cost of using the cloud keeps decreasing, adding data privacy guarantees would make cloud computing a cheaper alternative to expensive private datacenters for resource demanding applications.

One important drawback of current FHE cryptosystems, however, is the very high overhead associated with encryption, decryption, the homomorphic operations themselves, as well as bootstrapping. Thus, the practicality of FHE cryptosystems for real world applications has been questioned [4]. This constraint has driven research in the direction of improving the performance of current FHE cryptosystems [5], as well as discovering efficient alternatives to enable arbitrary computation over encrypted data under specific threat models [6]. In the latter case, arbitrary computation is achieved by leveraging partial homomorphic encryption (PHE) that only supports homomorphic addition of encrypted data, as well as branch decision information. Additionally, PHE is much more practical compared to FHE and a viable alternative for outsourced computation to the cloud.

It should be noted that, at a high level, the notion of privacy-preserving outsourced computation shares similarities with the notion of secure computation. Indeed, recent examples in that area have focused on secure function evaluation [7] and two-party computation [8]. Even though secure computation shares the same goals in preserving the privacy of the data contributed by individual parties, there are important differences that render outsourced computation a different problem. In the latter computational model, a single party owns all data and generates an algorithm to be applied upon said data and obtain the output of the computation. To reduce computation costs but maintain data privacy, the original party encrypts all data and outsources the evaluation of the algorithm to a third party that should learn nothing about the final result or the input data.

Conversely, in secure computation, at least two parties contribute inputs to a shared function, and through a privacy-preserving protocol, all parties learn the evaluation result and devote equivalent computation resources. Examples of this approach are based on oblivious transfer protocols and garbled circuits [9], and the generated circuit depends on the maximum input size. Evidently, learning even the computation result would leak some information: consider the example of a positive match of a DNA sample provided by one party, within a database provided by another. In this case, in addition to the “match” result, the first party also learns about an entry on the second party’s database, while the second party learns that the
query already exists within their database. Hence the scope of privacy-preserving outsourced computation is different from secure computation; in the former, shared knowledge of the final result is permitted and different circuits are evaluated depending on the input sizes, while in the latter, knowledge of the result in limited strictly to one party and the same algorithms can evaluate inputs of arbitrary sizes.

In this work, we focus on different approaches to achieve privacy in outsourced computation, leveraging the efficiency of a PHE cryptosystem as our baseline. We elaborate on the requirement of a functionally complete set of operations to enable evaluation of arbitrary functions, and analyze alternative approaches that can offer privacy using heuristic obfuscation along with our cryptographic primitives. Specifically, we discuss alternative PHE-based approaches to obfuscate computation with or without using look-up tables to enable branch decisions over encrypted data while preserving the privacy of the data.

The rest of the paper is organized as follows: Section II elaborates on the current limitations of FHE-based approaches for privacy-preserving outsourced computation, while Section III discusses a complete set of operations for arbitrary (Turing-complete) computations. Then, in Section IV we present two PHE-based approaches and we motivate the selection of a special architecture to support obfuscation. The paper concludes with final remarks in Section V.

II. FHE-BASED COMPUTATION LIMITATIONS

One important shortcoming of contemporary FHE schemes, which directly affects their practicality for arbitrary computation, is their efficiency. Indeed, known FHE schemes that follow Gentry’s original design inherit such limitations from their dependence on somewhat homomorphic encryption (SWHE) schemes as well as the need to reduce ciphertext noise using expensive bootstrapping operations [5]. Constructing a SWHE scheme is a prerequisite for building a FHE scheme, as such schemes are able to evaluate a limited number of homomorphic operations before the accumulated noise renders correct decryption of the result not possible. Specifically, homomorphic multiplications increase ciphertext noise rapidly, which ultimately limits the number of such operations a SWHE scheme can support.

As demonstrated by Gentry’s bootstrapping theorem [2], if it is possible to evaluate homomorphically the decryption function of a given SWHE scheme and one additional homomorphic operation, we can construct a “leveled” FHE scheme. Leveled schemes require public key sizes that grow depending on the number of homomorphic operations in the evaluated function (i.e., the size of the evaluated circuit), which is generally an undesired property. It is possible, however, to construct a FHE scheme with constant public-key sizes, if the assumption of “circular security” holds for the underlying scheme [5].

Remarkably, the requirement for SWHE schemes to homomorphically evaluate their own decryption function does not generally hold. The decryption functions are too complicated to allow their homomorphic evaluation within the limited number of operations supported by the SWHE scheme, so bootstrapping would not be possible. To overcome this limitation, it is possible to modify SWHE schemes so that decryption is adequately simplified (i.e. “squashed”) to allow its own homomorphic evaluation. In [2], this simplification is achieved using “hints” of the secret key within the public key.

Following the previous discussion, we observe that the inherent cost of existing bootstrapping methods is dominated by the cost of homomorphic decryption. In addition, since the homomorphic evaluation of decryption circuits requires that each secret key bit is encrypted as a separate ciphertext, bootstrapping costs are linear to the decryption cost multiplied by the number of ciphertext bits [5]. If \( \lambda \) is the security parameter (i.e., we require \( 2^\lambda \) security against known attacks), the complexity of decryption itself would be \( \Omega(\lambda) \). Moreover, the size of “fresh” ciphertexts in known SWHE schemes is \( \Omega(\lambda^3) \), in order to tolerate adequate amount of noise, ensure security and allow bootstrapping. Hence, the overall cost of bootstrapping a ciphertext is actually \( \Omega(\lambda^3) \) [5].

Evidently, the complexity of FHE-based cryptosystems can significantly impact their performance in evaluating arbitrary algorithms over encrypted data, as bootstrapping operations may be required after every few homomorphic operations. In addition, a hardware implementation of a FHE addition and multiplication module is yet to be reported in the literature, which can be attributed to the excessive area-power overhead and transistor switching activity, since each plaintext bit would be encrypted as a separate ciphertext of size \( \Omega(\lambda^3) \). Without such FHE modules in hardware, privacy-preserving outsourced computation is only possible as a software-level virtual machine; even though current implementations (e.g., [10]) may support optimizations such as SIMD-type vector operations, computation is still orders of magnitude more expensive compared to PHE operations. Indeed, PHE-based operations are typically based on modular multiplication operations, for which highly optimized algorithms exist (e.g., [11]), and efficient implementations have been reported in the literature (e.g., [12]).

The next Section elaborates on the requirement of having two orthogonal operations to support arbitrary computation.

III. FUNCTIONALLY COMPLETE SET OF OPERATIONS

Turing Machines. The inherent property of any system that is adequately powerful to recognize any algorithm, is typically referred to as Turing completeness [13]. Likewise, any computation by abstract Turing machines is equivalent to arbitrary or “general-purpose” computation (assuming adequate memory). In that respect, three grammar rules are essential to compute arbitrary programs: selection, sequence and repetition [14]. The selection rule defines a branch decision between the next address in order and an arbitrary (out of order) address, the sequence rule defines a direct branch to the next address, while the repetition rule defines a direct branch to an arbitrary address [15]. Hence, since it is possible to simulate sequence
and repetition as selections between replicated inputs, an inherent requirement to support arbitrary computation is the ability to make decisions.

Boolean Circuits. Due to the fundamental equivalence between uniform Boolean circuits and deterministic Turing machines [16], there exist a Boolean circuit that can evaluate the same total function as a resource-bounded Turing machine. Consequently, any decision within a Turing machine program would have an equivalent Boolean circuit implementation. Such decisions in Boolean circuits would require both multiplicative and additive functions (e.g., AND and OR gates respectively) so that the former supports replication or absorption of their inputs and a multiplicative absorbing element [17], while the latter combines its inputs and supports an additive identity element. In this context, decisions are possible if the multiplicative absorbing element is at the same time the additive identity element.

Algebraic Rings. The latter observation can also be demonstrated considering an algebraic ring with element zero as a multiplicative absorbing and additive identity element. Then, using additions and multiplications, we can select between inputs $X$ and $Y$ based on a binary value $S$ by evaluating the expression $X \cdot S + Y \cdot (1 - S)$ so that $S$ either replicates or absorbs the given factors. Evidently, a single operation would not be sufficient as it cannot have the same element both as absorbing and identity, by definition. Consequently, both addition and multiplication operations are necessary for making decisions and support arbitrary computation.

In the next Section, we discuss how PHE can be extended to support arbitrary computation.

IV. PHE-BASED ARBITRARY COMPUTATION

Since PHE cryptosystems support only one homomorphic operation (either addition or multiplication), arbitrary computation cannot be directly supported. The underlying limitation for Turing completeness, as already discussed, lies in the ability to make branch decisions. To overcome this limitation of PHE and benefit from its efficiency compared to FHE, the following two options are possible:

- An orthogonal homomorphic operation can be enabled through obfuscated decryption of ciphertext inputs, application of a regular operation on the corresponding plaintexts and re-encryption of the result; this option requires an obfuscated decryption key to be provided to the evaluator.
- Arbitrary computation can be enabled by implementing branch decisions through look-up tables of precomputed branch outcomes; this option requires look-up tables (but no obfuscated keys) to be provided to the evaluator (e.g., [18]). Moreover, depending on the available memory, these look-up tables can support probabilistic branch outcomes as well.

The following subsections provide further discussion on the effectiveness of obfuscation, as well as how one instruction set computer (OISC) architectures can provide both Turing completeness and heuristic obfuscation to support PHE-based computation.

A. Obfuscation Effectiveness

Definition. Obfuscation of a program or circuit refers to its transformation into a different one that is incomprehensible but functionally equivalent [19]. For that matter, a polynomial-time obfuscator program transforms the target program or circuit and satisfies two requirements: (a) anything the obfuscated target computes efficiently, is also computed efficiently through oracle (i.e., “black box”) access to the original circuit or program, and (b) exactly the same function is evaluated by the obfuscated target as well as the original. The effectiveness of obfuscation is then measured using the following metrics: (a) resilience, which refers to resistance against automated deobfuscation, (b) potency, which refers to resistance against humans, and (c) cost, which corresponds to the additional overhead introduced to the original program or circuit [20].

Applications. Obfuscation has several applications, especially for providing software protection and as a companion for encryption. Specifically, software applications can leverage obfuscation to protect the authors’ intellectual property, prevent reverse engineering, enforce rights management as well as apply watermarking to applications and data. Moreover, obfuscation allows converting public-key cryptosystems into homomorphic ones, as well as converting secret-key cryptosystems into public ones [19]. For the former, any homomorphic operation can be implemented using an obfuscated algorithm that incorporates a key-pair to decrypt ciphertexts, apply the required operation on the plaintexts and re-encrypt the result. Similarly, for the latter, private decryption and public encryption can be enabled by obfuscating the encryption permutation corresponding to the actual secret key.

Heuristic Obfuscation. One limitation of obfuscation as a protection mechanism, is that strong obfuscator programs (satisfying the two aforementioned requirements) do not exist in general [19]. Hence, even though it is possible in theory to construct obfuscator programs for certain point functions [21], there exist functions that inherently cannot be obfuscated. Therefore, assuming a realistic security model, heuristic obfuscation can be used for protecting the privacy of programs and circuits, given that it can offer high resilience and potency, at an acceptable cost.

B. Benefits of OISC Architectures

Any implementation of PHE-based computation would require a corresponding Instruction Set Architecture (ISA) for its execution engine (i.e., the processor executing instructions). Potential ISA candidates would be common CISC and RISC architectures with encrypted data and either encrypted or unencrypted opcodes for each instruction. Since Turing completeness requires two orthogonal operations to enable branch decisions, yet only one is supported by PHE, the execution engine would have to simulate the majority of the supported instructions using special algorithms that are based on the following available primitives: (a) one homomorphic
operation, and (b) precomputed branch outcomes or obfuscated decryption and re-encryption sequences.

A trivial solution could be to decrypt the data (and any encrypted opcodes) before decoding and executing each instruction in the unencrypted domain, and then re-encrypting the result. This approach, however, incurs significant overheads as it does not leverage the efficient homomorphic operations available. Furthermore, if the decryption/re-encryption operations are implemented using instructions of the same ISA, additional complexity is added to discriminate “original” program instructions from instructions of the decryption/re-encryption operations and prevent infinite recurrences. Similarly, implementing such decryption/re-encryption operations in hardware would require additional key management and provisioning, which would be susceptible to leakage through side channels or Trojans (e.g., [22], [23]).

A better solution for RISC/CISC architectures would be to leverage the aforementioned two primitives, to simulate all instructions not directly compatible with the supported homomorphic operation. The latter approach, however, would replace all such instructions with an equivalent sequence of the same homomorphic operation along with branch decisions. Interestingly, this is equivalent to simulating all non-PHE-compatible RISC and CISC instructions with a sequence of OISC instructions (with potentially increased overhead compared to executing OISC directly).

Conversely, certain OISC architectures provide a unique opportunity in this context, as they combine inherent obfuscation protection with native compatibility to additive PHE schemes (e.g., [24]). Indeed, the subleq OISC architecture is Turing-complete and its single instruction essentially consists of one additive operation (subtraction) and one branch decision [25]. Moreover, subleq programs are heavily based on self-modifying code, a native obfuscation technique also used by modern software packers to prevent human or automated disassembly [26]. Contrary to RISC/CISC architectures, OISC instructions are always the same (except for different arguments), which further obfuscates the control flow graph of programs to protect the confidentiality of the executed function. Besides, the simplicity of OISC reduces the implementation complexity for homomorphic computation engines, especially in hardware designs.

V. CONCLUDING REMARKS

In this work we discuss the performance limitations of FHE as a motivation for efficient, PHE-based, arbitrary computation. We elaborate on the requirement of having two orthogonal operations to support branch decisions and eventually Turing completeness, and present two alternative approaches to enable arbitrary computation using a single homomorphic operation and branch decision assistance. Moreover, we demonstrate the benefits of OISC architectures with respect to obfuscation and native support of homomorphic operations.

REFERENCES


