Compilation Techniques for Efficient Encrypted Computation

Christopher Fletcher
MIT CSAIL
cwfletch@mit.edu

Marten van Dijk
RSA Laboratories
marten.vandijk@rsa.com

Srinivas Devadas
MIT CSAIL
devadas@mit.edu

ABSTRACT

Fully homomorphic encryption (FHE) techniques are capable of performing encrypted computation on Boolean circuits, i.e., the user specifies encrypted inputs to the program, and the server computes on the encrypted inputs. Applying these techniques to general programs with recursive procedures and data-dependent loops has not been a focus of attention. In this paper, we take a first step toward building a compiler that, given programs with complex control flow, generates *efficient* code suitable for the application of FHE schemes.

We first describe how programs written in a small Turing-complete instruction set can be executed with encrypted data and point out inefficiencies in this methodology. We then provide examples of transforming (a) the greatest common divisor (GCD) problem using Euclid's algorithm and (b) the 3-Satisfiability (3SAT) problem using a recursive backtracking algorithm into a *path-levelized* form to which FHE can be applied. We describe how path levelization reduces control flow ambiguity and improves encrypted computation efficiency. Using these techniques and data-dependent loops as a starting point, we then build support for hierarchical programs made up of *phases*, where each phase corresponds to a *fixed point computation* that can be used to further improve the efficiency of encrypted computation.

In our setting, the adversary learns an estimate of the number of steps required to complete the computation, which we show is the least amount of leakage possible.

1. INTRODUCTION

Fully homomorphic encryption (FHE) techniques are capable of performing encrypted computation on Boolean circuits. In encrypted computation a program receives encrypted inputs and computes an encrypted result based on the encrypted inputs. Only the user who knows the decryption key can decrypt and interpret the result. Applying these techniques to general programs with recursive procedures, data-dependent loops and complex control flow has not received significant attention. In this paper, we show how to improve the efficiency of encrypted computation that uses FHE schemes by applying *path levelization* and *conditional fixed point* techniques during the compilation step of a program with complex control constructs.

Since the server deals with encrypted data it may not know what instruction to execute in the case of programs with complex control flow, and therefore it may need to execute many instructions at each time step or clock cycle. This ambiguity may make encrypted computation prohibitively expensive, even with efficient FHE schemes. We illustrate this by describing the execution of programs written using a small Turing-complete instruction set with encrypted data.

To decrease this control flow ambiguity, we show how a program can be transformed to a path levelized form, even when there are complex control flow constructs such as recursion in the original program description. We then show how the decomposition of a program into hierarchical *phases*, where each phase corresponds to a fixed point computation, can be used to significantly improve the efficiency of encrypted computation.

In our setting, given a program with encrypted inputs, the server performs encrypted computation for a specified number of steps, which the user or server believes is "more than enough." When the server exceeds its budget, it sends the most current program state back to the user. The user then determines whether the program state corresponds to the final result, or an intermediate result. The program is written so it either takes a deterministic amount of time, or reaches a fixed point after the final result is computed; in the latter case, the result is not corrupted even if there is additional encrypted computation. The adversary learns an estimate of the number of clock cycles required to complete the computation. We show that revealing an estimate on the number of cycles is the least amount of leakage possible.

Organization: The rest of this paper is organized as follows. We provide background in FHE methods and discuss related work in **Section 2**.

In **Section 3**, we define the two-interactive protocol and security model that we assume. We then show how arbitrary programs written in a small Turing-complete instruction set can be executed with encrypted data and point out the inefficiency in this approach.

In **Section 4**, we introduce path levelization as one way of reducing control flow ambiguity and fixed point computation as a way to protect program results. We demonstrate these ideas by transforming an algorithm that calculates the greatest common divisor (GCD) of two encrypted numbers, using Euclid's algorithm, and showing how the transformed algorithm is more efficient under encryption.

In **Section 5**, we describe a programming model where a program is decomposed into phases, and each phase corresponds to a fixed point computation. We provide an example of transforming a program that solves the 3-Satisfiability (3SAT) problem on an encrypted SAT formula using a recursive backtracking algorithm. Both path levelization and fixpoint computation techniques are applied.

We show that the two-interactive protocol results in the least amount of leakage to the server in **Section 6**, assuming a semi-honest server. We discuss the case of a malicious server in **Section 7**. **Section 8** concludes the paper.

2. BACKGROUND AND RELATED WORK

In his seminal paper [10], Craig Gentry presented the first fully homomorphic encryption (FHE) scheme [23, 9] that allows a server to receive encrypted data and perform, without access to the secret

decryption key, arbitrarily-complex dynamically-chosen computations on that data while it remains encrypted. Until recently, subsequent FHE schemes [7, 26, 24, 12, 6, 5] followed the same outline as in Gentry's construction: to construct a somewhat homomorphic encryption (SWHE) scheme (which is an encryption scheme capable of evaluating "low-degree" polynomials homomorphically), to squash the decryption circuit of the SWHE scheme (by adding a "decryption hint") such that it can be evaluated under encryption by the SWHE scheme itself (using an encrypted secret key that is given as one of the components of the public key), and to use Gentry's bootstrapping which refreshes a ciphertext (such that the ciphertext can be used in new homomorphic evaluations of low-degree polynomials).

The efficiency of FHE is measured by ciphertext/key size, encryption/decryption time, and the per-gate computation overhead defined as the ratio between the time it takes to compute a circuit homomorphically to the time it takes to compute it in the clear. The FHE schemes that follow the outline as in Gentry's original construction are inefficient in that their per-gate computation overhead is a large polynomial in the security parameter (see also the implementation results in [12, 6]).

In more recent developments, Gentry and Halevi [11], and Brakerski and Vaikuntanathan [4, 5] introduced new methods to construct FHE without using the squashing step, but still based on bootstrapping and an SWHE scheme. Even though their per-gate computation is still a large degree polynomial, their results have led to even more recent constructions that achieve dramatic (asymptotical) efficiency improvements: In [3] Brakerski, Gentry and Vaikuntanathan achieve asymptotically very efficient FHE schemes; their RLWE based FHE scheme that uses "bootstrapping as an optimization" has $\tilde{O}(\lambda^2)$ per-gate computation and is based on the ring-LWE problem [19] with quasi-polynomial approximation factor that has 2^λ security against known attacks ([4] bases security on LWE for sub-exponential approximation factors). We use this scheme throughout the paper to determine asymptotical performance when performing computation under encryption. They also present an L-leveled RWLE FHE scheme (which is able to evaluate circuits up to depth L) that does not use the bootstrapping procedure and which has $\tilde{O}(\lambda L^3)$ per-gate computation (quasi-linear in the security parameter λ) and is based on the ring-LWE problem with approximation factor exponential in L that has 2^λ security against known attacks. Both schemes apply the idea of modulus switching [4] which is used to refresh ciphertexts and manage the "noise" in the FHE scheme: Noise grows quadratically with every multiplication before refreshing. Modulus switching is used after each multiplication to reduce the noise growth to linear.

The most recent work of Brakerski [2] presents a new tensoring technique for LWE-based fully homomorphic encryption. This technique reduces the noise growth to linear with every multiplication. The resulting FHE scheme is scale-invariant in that its properties only depend on the ratio between the used modulus q and the initial noise level N. The scheme uses the same modulus throughout the evaluation process without the need for modulus switching and allows a modulus that is a power of 2. Finally, its security can be classically reduced to the worst-case hardness of the GapSVP problem with quasi-polynomial approximation factor (whereas previous constructions only exhibit a quantum reduction to GapSVP). The scheme can be generalized to RLWE which improves the efficiency even more. This has been detailed in [8], which also introduces optimizations and, based on [18], provides concrete parameter settings: for a 128-bit security level, plaintexts are bits and ciphertexts/public key each consist of two elements in R_q where $q = 2^{1358}$ is the modulus and R is the ring $\mathbb{Z}[x]/(x^{2^{10}} + 1)$.

In comparison, the SWHE scheme of [5] (without refresh procedures or noise reduction techniques) has been implemented in [22] for a 120-bit security level and is based on a ring R_q with $q \approx 2^{58}$ and $R = \mathbb{Z}[x]/(x^{2^{11}} + 1)$. Plaintexts are bits and ciphertexts/public key each consist of two or three elements in R_q. Key-generation runs in 250 ms, encryption takes 24 ms and decryption takes 15-26 ms. Homomorphic addition takes less than 1 ms and homomorphic multiplication takes about 41 ms. For completeness, we notice that adding or multiplying a ciphertext with a plaintext bit is almost instantaneous since the plaintext bit does not need to be encrypted with noise.

The most recent implementation paper [14] manages to execute one AES-128 encryption homomorphically in eight days using AES-specific optimizations (and using methods based on [25, 13]). Using single-instruction multiple-data (SIMD) techniques, close to 100 blocks are processed in each evaluation, yielding an amortized rate of roughly 2 hours per block.

A domain-specific language based on Haskell for cryptographically-secure cloud computing was proposed in [1]; simple programs without data-dependent loops were compiled for FHE application. [21] shows how information flow tagging can limit leakage due to control flow constructs. In our work we transform programs with complex control flow including recursion ensuring minimum leakage.

3. METHODOLOGY

To start, we introduce a general framework for performing computation under encryption for arbitrary programs. We will expand and optimize this framework in the later sections of the paper.

In this paper we use the notation `FHE.Enc`, `FHE.Dec`, `FHE.Add` and `FHE.Mult` for FHE encryption, FHE decryption, and addition and multiplication under FHE, respectively. We assume the existence of these operations and use FHE as a "black-box" module in our translation of programs into flows of instructions that can be evaluated under encryption. This translation can be viewed as a first step toward developing a compiler for computation under encryption using FHE (as opposed to Fairplay [20] which is a compiler for secure function evaluation based on garbled circuits [27]). We show how the translation can be made significantly more efficient in Section 4, through techniques that reduce control flow ambiguity.

3.1 Two-Interactive Protocols

We model general computation with a two-interactive protocol between a user and server. Formally, a two-interactive protocol Π for computing a deterministic algorithm $\mathcal{A} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ runs as follows:

1. The user wants the server to evaluate algorithm $\mathcal{A}$ on inputs from the user, collectively denoted by x, and inputs from the server, collectively denoted by y.

2. By using `FHE.Enc`, the user encrypts its inputs to form the ciphertext $\mathcal{M}_0$ and then chooses a number of steps S, where a step is some atomic unit of work ($\mathcal{M}_0$ is notation for "the encrypted state after zero steps"). The server then transmits to the server the pair $(\mathcal{M}_0, S)$ together with algorithm $\mathcal{A}$ if $\mathcal{A}$ has not already been transmitted to the server.

3. After receiving the pair $(\mathcal{M}_0, S)$, the server evaluates $\mathcal{A}$ under encryption on $\mathcal{M}_0$ and its own inputs y by using `FHE.Add` and `FHE.Mult` for S steps. The result is a ciphertext $\mathcal{M}_S$, which the server transmits to the user.

4. By using FHE.Dec, the user decrypts $\mathcal{M}_S$ and checks whether S was sufficient to complete $\mathcal{A}(x, y)$ (without loss of generality, we assume that the algorithm outputs an "I am done" message as part of its final result).

A correct execution of Π outputs to the client the evaluation $\mathcal{A}(x, y)$ (if S was sufficient) or some *intermediate result*.

The disadvantage of only two interactions is that the user may receive an intermediate result (rather than the final one) indicating that the computation was not finished. The advantage is no additional unnecessary privacy leakage about the final result, i.e., the server does not gain additional understanding about the output of the algorithm evaluated on the unencrypted inputs besides what the server is already able to extract from the algorithm itself, the number and sizes of the encrypted inputs, and other a-priori knowledge. We will prove in Section 6 that this leakage is optimal.

3.2 Security Model

We assume the semi-honest model: the server is "honest but curious." The server is honest in that it executes $\mathcal{A}$ under encryption for the required number of steps and sends back the result exactly as specified (no deviations, malicious or otherwise). In particular, the server does not send back to the user the result produced by executing a different algorithm, or evaluating $\mathcal{A}$ on different inputs, or evaluating $\mathcal{A}$ on the user's input for less than S steps. The server is curious in that it may try to learn as much as possible about the input of the user from its view; its view is the algorithm and encrypted inputs as given by the user (and possibly other a-priori information). This means that the server may execute other algorithms on other inputs in order to satisfy its curiosity (e.g., a small deviation of the specified algorithm, or the specified algorithm where some intermediate results are altered). We want the view of the server to not leak more knowledge (i.e., not compromise more privacy) than necessary. The interested reader can read more about malicious servers in Section 7.

3.3 A Turing Machine Approach

A first (straightforward) approach to general computation under encryption is to use Turing machines (TMs) to model program execution. This methodology would translate TMs to arithmetic circuits which can be evaluated under encryption through FHE schemes. A TM consists of a finite-state control unit and a tape (which we refer to as $\mathcal{M}$). Communication between the two is provided by a single head, which reads symbols from $\mathcal{M}$ and is also used to change the symbols on $\mathcal{M}$. The program tape $\mathcal{M}$ is analogous to the encrypted input $\mathcal{M}$ from Section 3.1 and each movement of the TM head can be viewed as one step.

The control unit operates in discrete steps. At each step it performs one of two functions in a manner dependent on its current state and the symbol currently scanned on $\mathcal{M}$: (1) Write a symbol on $\mathcal{M}$ at the square currently scanned, replacing the one already there; or (2) move the read/write head one square on $\mathcal{M}$ to the left or right. A TM is fully described by its state, the position of its head, and the content of $\mathcal{M}$. We include a detailed translation from a TM into an arithmetic circuit in Appendix A for reference.

Since state, position of the head, and the content of $\mathcal{M}$ are encrypted, any possible combination needs to be taken into account in each TM iteration. This means that each TM iteration involves a computation on the position of the head, each square on $\mathcal{M}$ and the state. This is costly and turns out not to be necessary. Over the next several subsections, we introduce a program structure approach in which the TM state corresponds to a global program counter and $\mathcal{M}$ corresponds to the program's state (memory)—both of which are encrypted. We will then present efficiency improvements to this base model in Sections 4 and 5.

Table 1: Summary of data used by the server during an encrypted computation.

Encrypted	**Unencrypted**
Data memory ($\mathcal{M}$)	*Instruction memory*
Program inputs	The program $\mathcal{A}$
Intermediate values	Boolean networks for each $b \in \mathcal{B}$
Results	PC addresses for each $b \in \mathcal{B}$
PC'	S, a number of steps to run for

3.4 Program Structure

An arbitrary program $\mathcal{A}$ is represented as a set $\mathcal{B}$ of basic blocks. (We refer to the j^{th} basic block as $\mathcal{B}_j$ for $j = 1 \ldots |\mathcal{B}|$). Each basic block is made up of an ordered sequence of instructions and has one entry and one exit point. Basic blocks are evaluated *atomically*—program flow enters at the top of a basic block and all instructions within the basic block are evaluated in the specified order. Each basic block has a public, fixed address that corresponds to a unique *program counter*, denoted PC.

Programs contain two types of instructions: *sub* (for subtract) and *branch*. The *sub, a, b* operation performs:

$$\mathcal{M}[b] = \mathcal{M}[b] - \mathcal{M}[a]$$

where $\mathcal{M}$ is the program *data memory* (equivalent to the Turing machine tape from Section 3.3). Each basic block is terminated by a *branch a, BBX, BBY* instruction, which performs

$$\text{if } \mathcal{M}[a] < 0 : PC = \text{PC}(BBX)$$
$$\text{else} : PC = \text{PC}(BBY)$$

where $\text{PC}(BBX)$ is notation for "the PC address of basic block BBX." Together, these instructions are Turing-complete [15]. If instruction *branch a, BBX, BBY* occurs in $\mathcal{B}_j$ for some j, we say that BBX and BBY are the *successor* basic blocks to $\mathcal{B}_j$ (without loss of generality, each basic block can have at most two successors from its branch exit point). Likewise, we say that $\mathcal{B}_j$ is a predecessor to BBX and BBY (every basic block other than the entry block to the program must have at least one predecessor). To logically terminate the program, we use a special basic block whose PC address is PC_{done} and has an unconditional *branch* instruction back to $PC' = PC_{\text{done}}$.

3.5 Computation under Encryption

We now show how to map an unencrypted program $\mathcal{A}$ from the previous section to an FHE scheme. We focus on two-party computation between a trusted user and semi-honest server as has been described so far. A summary of the encrypted and unencrypted values used by the server is shown in Table 1.

To perform computation under encryption, the server converts each basic block $\mathcal{B}_j$ for $j = 1 \ldots |\mathcal{B}|$ to a Boolean network that can be used to perform FHE-based computation on encrypted inputs. Since $\mathcal{A}$ is public, each Boolean network and its corresponding $\text{PC}(\mathcal{B}_j)$ is not encrypted and is known to the server—we say that the server stores these values in an unencrypted *instruction memory*. The user then augments $\mathcal{M}$ with an encrypted PC called PC', which is updated under encryption after each basic block is evaluated. PC' corresponds to the program's data-dependent control flow, which is not known to the server. An important point is that the server can deduce the value of PC' at the beginning of the program, and also at certain points during execution (e.g., if straight-line code is being executed). The server also knows where PC' is located in $\mathcal{M}$.

The user starts an algorithm by sending the server (a) an unencrypted number of *steps* S and (b) an encrypted initial data memory $\mathcal{M}_0$ as described in Section 3.1. For $s = 0 \ldots S-1$, the server performs the following *state update* operation:

$$\mathcal{M}_{s+1} = \sum_{j=1}^{|\mathcal{B}|} c_j * \mathcal{B}_j(\mathcal{M}_s) \quad (1)$$

where c_j is given by:

$$\text{if } \mathrm{PC}(\mathcal{B}_j) \stackrel{?}{=} PC' : c_j = 1 \quad (2)$$
$$\text{else} : c_j = 0$$

and $\mathcal{B}_j(\mathcal{M}_s)$ means "evaluate the Boolean network corresponding to $\mathcal{B}_j$, given the encrypted program state $\mathcal{M}_s$." All operators in the update are mapped to the primitive FHE operations that we specified at the beginning of Section 3.

Notice that at any point in the computation, $\mathrm{PC}(\mathcal{B}_j)$ is equal to PC' for *exactly* one $\mathcal{B}_j$; we are assuming sequential program semantics. Thus, the server makes *forward progress* in $\mathcal{A}$ after every step. The server evaluates every basic block in parallel at each step because it does not know the true value of PC'.

Equivalently, one can perform the state update operation per-instruction instead of per-basic block. For simplicity and efficiency reasons, we treat basic blocks as the atomic unit of work in this paper.

After S steps, the server returns $\mathcal{M}_S$ back to the user. If $\mathcal{A}$ given $\mathcal{M}_0$ requires S' basic blocks to complete without encryption and $S \geq S'$, $PC' \stackrel{?}{=} PC_{\mathrm{done}}$ holds and $\mathcal{M}_S$ corresponds to the final program state. In a realistic setting, the user cannot always know S for an arbitrary $\mathcal{A}$ given arbitrary inputs. If the user decrypts PC' and sees that $PC' \stackrel{?}{=} PC_{\mathrm{done}}$ is false, $\mathcal{M}_S$ corresponds to an intermediate result.

3.6 Efficiency Under Encryption

Let S be large enough such that when an arbitrary input is run on $\mathcal{A}$, $PC' \stackrel{?}{=} PC_{\mathrm{done}}$ holds at some step $s \leq S$. With the above state update function, $O(S * |\mathcal{B}|)$ basic blocks are evaluated in order to complete $\mathcal{A}$; we call this the *absolute computation* performed by the server.

Assuming an RLWE FHE scheme and N initial noise on $\mathcal{M}_0$, the *noise* after running $\mathcal{A}$ will be $O(N^{2S})$. The noise comes from two flavors of `FHE.Mult` operations:

1. Since the server does not know PC', we perform an `FHE.Mult` operation between each c_j and $\mathcal{B}_j(\mathcal{M}_s)$ as shown in Eqn. 1.
2. For the j such that $c_j = 1$, the server does not know what branch direction will be taken in $\mathcal{B}_j$. Let $x = \mathcal{M}[a] < 0$. Then the PC' update operation can be viewed as
$$PC' = x * \mathrm{PC}(BBX) + (1-x) * \mathrm{PC}(BBY),$$
which results in an additional `FHE.Mult` and a corresponding increase of noise on PC'. This noise factor is not added for basic blocks with a single successor.

We do not count additive noise through `FHE.Add` operations.

4. EFFICIENT COMPUTATION

The program transformation in Section 3 is theoretically useful for its generality, but is not an efficient transformation. Suppose that $\mathcal{A}$ can be completed in S steps. Clearly, the server can complete $\mathcal{A}$ on unencrypted data after evaluating $O(S)$ basic blocks because $\mathcal{A}$'s data-dependent control flow is known to the server in this situation. Thus, in addition to the overhead of performing operations under encryption, the construction from Section 3.5 performs a factor of $O(|\mathcal{B}|)$ additional work because the server does not know PC'.

To address this efficiency issue, we propose two compilation techniques. The first technique is called *path levelization*, which lowers the $O(S * |\mathcal{B}|)$ and $O(N^{2S})$ bounds from Section 3.5 by reducing *PC ambiguity*. The second technique is a program-level notion of *fixed point* behavior. After explaining the transformations, we demonstrate how to transform and evaluate Euclid's Greatest Common Divisor (GCD) algorithm under encryption.

For the rest of this section, we consider $\mathcal{A}$ to be an arbitrary data-dependent loop with a single *entry* basic block and a single *exit* basic block. The *exit* block transitions to *entry* to start another iteration, or exits the loop. ($\mathcal{A}$ can be viewed as a `do while` loop). We expand the scope of our transformations to larger programs (such as those with data-dependent recursive calls) in Section 5.

4.1 Program Counter Ambiguity

Our transformations are based on trying to decrease *program counter (PC) ambiguity*. If a program $\mathcal{A}$ executes exactly s steps for some s, PC' points to some basic block in set $\mathcal{C}_s$ where $\mathcal{C}_s \subseteq \mathcal{B}$. While evaluating under encryption for some number of steps s, the server needs to assume that any $\mathcal{B}_j \in \mathcal{C}_s$ might satisfy $\mathrm{PC}(\mathcal{B}_j) \stackrel{?}{=} PC'$.

Let $\mathrm{BBX}(\mathcal{B}_j)$ and $\mathrm{BBY}(\mathcal{B}_j)$ be the target basic blocks for the branch instruction in $\mathcal{B}_j$. Then

$$\mathcal{C}_{s+1} = \{\mathrm{BBX}(\mathcal{B}_j) : \mathcal{B}_j \in \mathcal{C}_s\} \cup \{\mathrm{BBY}(\mathcal{B}_j) : \mathcal{B}_j \in \mathcal{C}_s\}$$

Given an arbitrary $\mathcal{A}$, $\mathcal{C}_s$ could be as large as $\mathcal{B}$ at some step $s < S$, which means that the only way to guarantee forward progress under encryption is to perform the state update given by Eqn. 1 (cf. Section 3.5). If $|\mathcal{C}_s| = 1$ for some step s, we say that PC' is unambiguous at s.

4.2 Path Levelization

The idea behind path levelization is to evaluate the basic blocks in $\mathcal{A}$ by *level* to shrink program counter ambiguity. Given basic blocks X and Y and number of steps L, path levelization ensures that the following property holds:

$$\forall s : \mathcal{C}_s = \{X\} \Rightarrow \mathcal{C}_{s+L} = \{Y\}$$

For the remainder of this section, we assume that basic block X is set to the *entry* basic block in $\mathcal{A}$, that Y is set to the *exit* block, and that L is the longest path (in terms of basic blocks) between the *entry* and *exit*. In this context, the property is saying that ambiguity on PC' is independent of the number of data-dependent iterations in $\mathcal{A}$.

To levelize $\mathcal{A}$, each $\mathcal{B}_j$ is assigned to a level (a set) $\mathcal{L}_l$ for $l = 1 \ldots L$, relative to the *entry* block such that

$$\mathcal{L}_l = \{\mathcal{B}_j : (1 + \mathrm{distance}(entry, \mathcal{B}_j)) \stackrel{?}{=} l, \mathcal{B}_j \in \mathcal{B}\}$$

where $\mathrm{distance}(entry, \mathcal{B}_j)$ is the maximum path length (in terms of basic blocks) from *entry* to $\mathcal{B}_j$.

Figure 1 shows an example loop $\mathcal{A}$ overlaid with levels. $\mathcal{C}_1 = \{A\}$ and $\mathcal{C}_2 = \{B, C\}$ correspond to level 1 and level 2. At step 3, however, $\mathcal{C}_3 \neq \mathcal{L}_3$ because the branch in block C may jump to level 3 or level 4. Eventually (when $s > 12$), this slip causes $\mathcal{C}_s = \mathcal{B}$. Thus, the state update function from Section 3.5 must evaluate every basic block at each subsequent step to make progress. We now show how levelization prevents the slip.

To perform computation on a levelized loop, we use the following state update function:

$$\textbf{if } \mathrm{dominator}(\mathcal{L}_{l,1}): \ \mathcal{M}_{s+1} = \mathcal{L}_{l,1}(\mathcal{M}_s) \quad (3)$$

$$\textbf{else}: \ \mathcal{M}_{s+1} = \sum_{j=1}^{|\mathcal{L}_l|} c_j * \mathcal{L}_{l,j}(\mathcal{M}_s) + \underbrace{\left(1 - \sum_{j=1}^{|\mathcal{L}_l|} c_j\right)}_{\text{identity operation}} * \mathcal{M}_s \quad (4)$$

where $l = (s \bmod L)+1$, $\mathcal{L}_{l,j}$ corresponds to the j^{th} basic block in level l, $\mathcal{L}_{l,1}$ is the first basic block in level l, and each c_j is defined in the same way as in Eqn. 2.

Eqn. 3 is an optimization: when PC ambiguity disappears, we do not have to perform a multiplication with c_j. $\mathrm{dominator}(\mathcal{L}_{l,1})$ returns $true$ if $\mathcal{L}_{l,1}$ is a *dominator* basic block—that is, if every path from the *entry* to the *exit* block must pass through $\mathcal{L}_{l,1}$—and $false$ otherwise. Notice that when the server evaluates $\mathcal{A}$ level by level, $|\mathcal{C}_s| = 1 \Rightarrow \mathrm{dominator}(\mathcal{L}_{l,1}) \Rightarrow |\mathcal{L}_l| = 1$ (we refer to this case with $\mathrm{dominator}()$ to aid the reader). Thus, the server can decide when to evaluate Eqn. 3 by keeping track of $\mathcal{C}$ at each step.

The *identity operation* term corresponds to the case when $\mathcal{C}_s \neq \mathcal{L}_l$ and means that evaluating the current level does not necessarily update the state. Note that when $\mathcal{C}_s = \mathcal{L}_l$ (which the server can detect), the server does not need to evaluate the identity operation (e.g., level 2 in Figure 1).

Putting these ideas together, the schedule of basic blocks that the server evaluates (per step) for Figure 1 are:

$$\mathcal{L} = [\{A\}^{\bullet}, \{B, C\}, \{D, E, F\}, \{G\}^{\bullet}]$$

(read left to right). We mark each $\mathcal{L}_l$ with '$^{\bullet}$' if $\mathrm{dominator}(\mathcal{L}_{l,1})$ holds (meaning that multiplications were optimized away). Going back to our example, notice that $C_3 = \{D, E, F, G\}$ but level 3 only has $\{D, E, F\}$. If basic block C transitions to H, no forward progress is made in the loop when level 3 is evaluated. This ensures that when level 4 is evaluated: $\mathcal{C} = \{G\}$. Thus, for any loop iteration, the server knows that when it evaluates level 1: $\mathcal{C} = \{A\}$.

To perform S steps of computation on $\mathcal{A}$, the state update function evaluates each level in order from left to right and repeats until it evaluates $I = \frac{S}{L}$ loop iterations.

4.3 Fixed Point

We say that $\mathcal{A}$ is in fixed point form if the following property holds: *if $\mathcal{A}$ completes in S' steps and $S > S'$, useful program state in $\mathcal{M}$ is not corrupted.* We say that *useful* program state is that which is used by the user after the computation. Since $\mathcal{A}$ is a data-dependent loop that depends on $\mathcal{M}$, the server cannot generally guess when to exit and we have to protect state from getting corrupted. To put $\mathcal{A}$ in fixed point form, the programmer or compiler performs four steps:

1. Augment $\mathcal{M}$ with an encrypted *done* flag ($true/false$).

2. Remove the data-dependent loop condition and replace it with a condition that loops until I iterations are reached, where I is public.

3. Add a *header* basic block to $\mathcal{A}$ that is only evaluated during the first step in the computation. This header initializes *done* to $false$ and then unconditionally transitions to the *entry* basic block in $\mathcal{A}$. It is the programmer or compiler's responsibility to wrap $\mathcal{A}$'s useful program state in `if (not done) {...}` logic to prevent it from being corrupted.

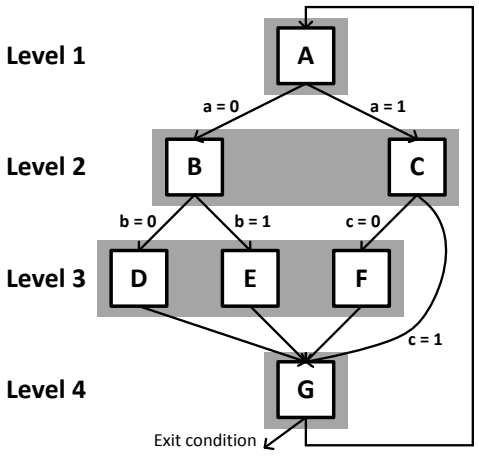

Figure 1: Control flow diagram for an example loop, overlaid by the 4 levels of execution. Arrows indicate legal branch transitions; branch directions are given by lowercase letters for their basic block (e.g., a is the branch direction for basic block A). When control transitions from basic block C to G, level 3 does not make forward progress.

4. Evaluate a *transition check* that performs the following operation once, after the I^{th} iteration of $\mathcal{A}$ completes (say this happens at some step s):

   $$PC'_{s+1} = done * PC(BB_{\text{next}}) + (1 - done) * PC'_s$$

   Where BB_{next} is the basic block that $\mathcal{A}$'s loop exits to. Note that when $done \stackrel{?}{=} false$ holds, *PC'* gets stuck at the *entry* basic block of $\mathcal{A}$. This rule will be used in Section 5.1.1 to support programs with multiple and hierarchical data-dependent loops.

The *done* flag is a program-level construct that may seem redundant because of the c_j terms (which also implement fixed point-like behavior for each basic block) from Section 3.5. We have introduced a second level of fixed point behavior to enable the optimization in Eqn. 3 from Section 4.2 where we remove certain c_j terms.

4.4 Efficiency with Transformations

Consider the "worst case input" a to $\mathcal{A}$ which flows through L basic blocks per iteration and runs for some number of iterations I. Path levelizing $\mathcal{A}$ yields $\mathcal{A}'$; by the definition of path levelization, the longest path through $\mathcal{A}$ is the *same* sequence of basic blocks as that through $\mathcal{A}'$. (We assume that fixed point logic adds a negligible number of basic blocks to $\mathcal{A}$).

4.4.1 Absolute computation

Let $\mathcal{L}$ follow the description from Section 4.2. Recall that absolute computation using the update function from Section 3.5 requires $O(S * |\mathcal{B}|)$ basic blocks to be evaluated, which is equivalent to $O(I * L * |\mathcal{B}|)$. Notice that per-iteration, the path-levelized scheme evaluates each basic block once, plus potentially an identity operation per-level. Thus, $\mathcal{A}'$ can be completed by evaluating $O(I * (L + |\mathcal{B}|))$ basic blocks.

4.4.2 Noise

Since the maximum path length after path levelization is no worse than before path levelization, the upper bound on the noise

after our transformations is still $O(N^{2S})$. The compiler in [20] uses single variable assignment for program values. We adapt this technique to computation under encryption to reduce noise on $\mathcal{M}$. Given $\mathcal{M}_s$ for some step s, break $\mathcal{M}$ bitwise into $\mathcal{M}_{s,b}$ for $b = 1 \ldots B$ where B is the encrypted bit-length of $\mathcal{M}$. We say that $\mathcal{M}_{s,b}$ only incurs noise during step s if some basic block in $\mathcal{L}_l$ (where $l = (s \bmod L) + 1$) updates $\mathcal{M}_{s,b}$ and makes forward progress in $\mathcal{A}'$ ($\mathcal{L}_l$ is public).

With the single variable assignment technique, prior to any optimization, we make the following observation: *the noise limiting variable in $\mathcal{M}$ is PC'*. At step s, noise is contributed to every variable modified in a basic block in $\mathcal{L}_l$. Since, PC' is changed in every basic block, it accumulates the most noise between two steps s_1 and s_2 where $s_2 > s_1$.

When $\mathrm{dominator}(\mathcal{L}_{l,1})$ holds for some l, two optimizations to the noise occur:

1. Any variable modified in basic block $\mathcal{L}_{l,1}$ incurs one less multiplication operation because the corresponding c_j term has been eliminated. For example, if k is the number of levels where $\mathrm{dominator}(\mathcal{L}_{l,1})$ holds, the noise added to PC' in the path levelized program is $O(N^{2S-k*I})$.

2. *PC' can be reset to $PC(\mathcal{L}_{l,1})$—a constant value known to the server.* This operation performs an implicit refresh operation on PC', eliminating any noise it has accumulated. Thus, if the RLWE FHE scheme permits at least N^{2L} noise on PC', PC' will never have to be explicitly refreshed.[1] This is true because $\mathrm{dominator}(\mathcal{L}_{l,1})$ holds at the *entry* basic block in $\mathcal{A}'$ by design.

Conceptually, the noise limiting variable (v_{limit}) becomes the variable that is modified at least one time in the most levels. v_{limit} can be determined by the server and is program dependent. Note that for some $\mathcal{A}'$, v_{limit} could still be PC'.

4.5 Alternate Approaches

Boolean network flattening. Path levelization updates PC' after each level is evaluated. Another approach is to update PC' once per iteration of $\mathcal{A}$; that is, flatten $\mathcal{A}$ into a single Boolean network and run that entire Boolean network as a single step, I times. Expanding the resulting Boolean network, this approach evaluates every possible path from the *entry* to *exit* basic block. Consider the example in Figure 1, where lowercase letters are branch directions from the figure. Let $\mathcal{M}_{\mathrm{old}}$ be the state at the beginning of an iteration; the equation needed to evaluate a single iteration is:

$$\begin{aligned}\mathcal{M}_{\mathrm{new}} =&((1-a)*(1-b))*G(D(B(A(\mathcal{M}_{\mathrm{old}}))))+\\ &((1-a)*b)*G(E(B(A(\mathcal{M}_{\mathrm{old}}))))+\\ &(a*(1-c))*G(F(C(A(\mathcal{M}_{\mathrm{old}}))))+\\ &(a*c)*G(C(A(\mathcal{M}_{\mathrm{old}})))\end{aligned}$$

The number of paths from *entry* to *exit* can increase exponentially with depth L. A smarter implementation can memoize common subpaths (e.g., $B(A(\mathcal{M}_s))$ is used twice) to save work. If the server memoizes, it needs to copy some or all of $\mathcal{M}$ for every entry in the memoization table, which requires potentially exponential space. The advantage of this scheme is that it reduces noise due to multiplication operations by $\sim \frac{1}{2}$—each path only needs to be checked against the $O(L)$ branch directions that form that path.

Event-driven algorithms. It is tempting to try and decrease PC ambiguity with event-driven algorithms that maintain work queues of basic blocks, and evaluate basic blocks that are "ready to fire." This approach performs more work and adds more noise than path levelization because each basic block j in the queue would be evaluated using the update function:

$$\mathcal{M}_{s+1} = c_j * \mathcal{B}_j(\mathcal{M}_s) + \underbrace{(1-c_j)*\mathcal{M}_s}_{\text{identity operation}}$$

This adds an additional multiplication's worth of noise and absolute computation in the identity operation *per-step*; the update function from Section 4.2 only adds this computation *per-level*.

4.6 Example: Greatest Common Divisor

We now show how to apply path levelization and program-level fixed points to Euclid's Greatest Common Divisor (GCD) algorithm, as shown in Algorithm 1, and then how to run the transformed program under an FHE scheme.[2]

4.6.1 GCD Transformation

The original algorithm (GCD) has a `while` loop that depends on the GCD inputs (a and b). The transformation (GCD$\star$, shown graphically in Figure 2) runs for I iterations where I is independent of a and b.

Algorithm 1 Original and transformed GCD algorithms. The main loop in $\mathcal{A}$ is shown between lines 3 and 12 in GCD$\star$.

```
    GCD(a, b):
    while true do
      if a ?= b then
        return a
5:    else if a < b then
        b = b − a
      else
        a = a − b
      end if
10: end while

    GCD⋆(a, b):
    d = 0
    for i = 0; i < I; i++ do
      d = a ?= b
5:    if d then
        r = a
      else if a < b then
        b = b − a
      else
10:     a = a − b
      end if
    end for
    return a, b, r
```

Suppose that for a given a and b, GCD$\star$ can find the GCD after running for I' iterations. We have two cases for r, a, b after the I^{th} iteration of $\mathcal{A}$ completes:

$I \geq I'$: $a \stackrel{?}{=} b$ holds (d is $true$). $r = \mathrm{GCD}$.

$I < I'$: $a \neq b$ (d is $false$). $r = \mathrm{invalid}$.

The behavior of variable d implements the *done* fixed point flag: once d is set, a and b are no longer modified, regardless of S.

[1] Our technique is portable to other FHE schemes; the only difference is that one must respect different noise growth functions.

[2] Our GCD implementation assumes a subtract instead of a *mod* operation for simplicity of presentation.

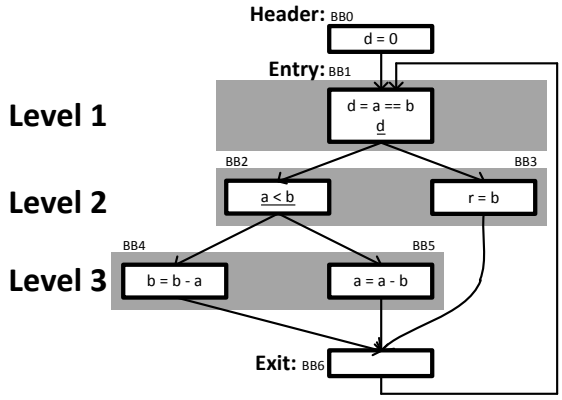

Figure 2: The control flow graph for GCD⋆. Basic blocks are prefixed with BB. $BB6$ is not counted as a level and is only shown to conceptually give the GCD⋆ loop a single exit point.

4.6.2 Transformed GCD to FHE

See Figure 2. Under encryption, the user sends $\mathcal{M}_0$ which contains encrypted values for a, b and the initial PC'. The server synthesizes

$$\mathcal{L}_{GCD} = [\{BB1\}^{\bullet}, \{BB2, BB3\}, \{BB4, BB5\}]$$

and then evaluates $BB0$ one time to initialize d and then performs I iterations worth of work. The server must evaluate both basic blocks in $\mathcal{L}_2 = \{BB2, BB3\}$ because it does not know the value of PC' after $BB1$'s branch; in $\mathcal{L}_3$, the server must evaluate $\{BB4, BB5\}$ as well as an identity operation in case $BB3$ transitioned to $BB6$. After performing I iterations, the server performs the *transition check* from Section 4.3; if $d \stackrel{?}{=} true$ holds, PC' is set to PC_{done}; otherwise it stays at $PC(BB1)$, the entry block. After PC' is finalized, the server returns back an updated $\mathcal{M}$ containing r and PC'.

When the user receives a server response, it decrypts PC' and checks if $PC' \stackrel{?}{=} PC_{\text{done}}$. If the condition holds, the user decrypts r which is the GCD. Otherwise, the user has received an intermediate result.

4.6.3 Efficiency for Transformed GCD

We analyze GCD⋆ using the state update function from Section 3.5 (*baseline*) and that from Section 4.2 (*levelized*). To perform a single iteration's worth of work with *baseline*, we must evaluate $5 + 5 + 5 = 15$ basic blocks, as each iteration has a path length of 3 basic blocks (e.g., while $done$ is $false$) and the GCD⋆ loop contains 5 basic blocks. The same computation with *levelized* only evaluates $1+2+3 = 6$ basic blocks ($|\mathcal{L}_1| = 1, |\mathcal{L}_2| = 2, |\mathcal{L}_3| = 2$ with an identity operation added).

Table 2 compares the noise between *baseline* and *levelized*. *Baseline* performs 6 multiplications per iteration on PC' while *levelized* performs 4. In addition, *levelized* implicitly refreshes the noise on PC' after each iteration—it almost certainly will not need to be explicitly refreshed. Note that we need two multiplications on PC' in *levelized ($l = 2$)* because there is PC ambiguity and $BB2$ can transition to one $BB4$ or $BB5$. PC' only requires one multiplication during *levelized ($l = 3$)* because all basic blocks in $l = 3$ unconditionally transition to $BB6$. *Baseline* requires $3\times$ multiplications for variables a, b, r and d since the server makes no assumptions about where it is in the program.

Table 2: Comparison of the number of multiplications per-step, per-variable, between a baseline state update function (Section 3.5) and a levelized state update (Section 4.2).

		Levelized		
Variable	Baseline	$l = 1$	$l = 2$	$l = 3$
PC'	2	refreshed, 1	2	1
d	1	1	0	0
a	1	0	0	1
b	1	0	0	1
r	1	0	1	0

5. PROGRAM HIERARCHY

We will now broaden the scope of our transformations and evaluation approach to larger programs. Section 5.1 expands our definition of $\mathcal{A}$ to include programs with multiple data-dependent loops and introduces the notion of *program phases*. Section 5.2 then explains how our method can be applied to programs with data-dependent recursive calls. Using these ideas, we transform a recursive backtracking 3-Satisfiability algorithm in Section 5.3.

5.1 Hierarchical Data-Dependent Loops

We now show how to transform a program $\mathcal{A}$, containing a set of (potentially) data-dependent loops and *simple* function calls, to path levelized and fixed point form. We assume that each loop $\mathcal{P}_k$ for $k = 1 \ldots |\mathcal{P}|$ takes on the form described in Section 4. Loops may be disjoint from other loops or contain other loops—we say that a loop $\mathcal{P}_k$ is an *inner-most* loop if no basic block within $\mathcal{P}_k$ is part of some other loop $\mathcal{P}_j$ for $j = 1 \ldots |\mathcal{P}| \wedge k \neq j$. We define *simple* function calls as those that (a) are made up of basic blocks and data-dependent loops (like $\mathcal{A}$) and (b) are not recursive.

To transform $\mathcal{A}$, we perform a two-step process. First, all simple function calls in $\mathcal{A}$ are inlined. $\mathcal{A}$ is now composed of basic blocks, some of which may be part of one or more loops. Second, if a loop $\mathcal{P}_k$ is an inner-most loop, path levelize and apply fixed point transformations to $\mathcal{P}_k$ as described in Section 4. This process assigns a public iteration count I_k to $\mathcal{P}_k$. Post-transformation, we make the following observation for $\mathcal{P}_k$: *if (a) $\mathcal{P}_k$ requires I'_k iterations to complete without encryption, (b) L_k is the number of levels in $\mathcal{P}_k$, and (c) $I_k \geq I'_k$: then $\mathcal{P}_k$ can be thought of as a basic block that requires $L_k \times I_k$ steps to complete (we call this a phase*[3]*).* Phases can be assigned to levels in the same way as basic blocks from Section 4.2. Once all inner-most loops are turned into phases, we form a new set of inner-most loops from $\mathcal{P}$ (not including existing phases) and repeat the above process until all loops are transformed into phases. Once all loops are turned into phases, we levelize the outermost scope of $\mathcal{A}$; $\mathcal{A}$ can be thought of as a phase that runs for a single iteration.

The outcome of this process is shown in Figure 3 for an example two-level loop. (We will use this example throughout the rest of Section 5.1). First, the inner-most loop $\mathcal{P}_2$ is path levelized and given fixed point logic. This gives $\mathcal{P}_2$ the appearance of a basic block, so $\mathcal{P}_2$ is assigned to level 2 of the outer phase $\mathcal{P}_1$. Finally, $\mathcal{P}_1$ can be path levelized, put into fixed point form, and given an unconditional back-edge from $D \rightarrow A$.

5.1.1 Computation with Program Phases

After the hierarchical transformation from Section 5.1, $\mathcal{A}$ is composed of a hierarchy of phases and each phase is made up of basic

[3]In the compiler community, a phase is a type of super block.

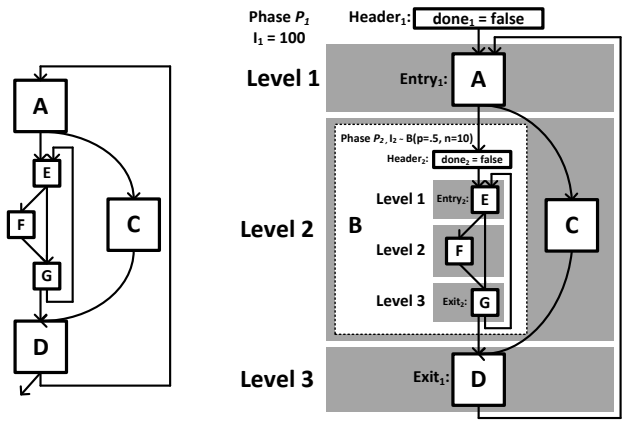

Figure 3: A hierarchical two-level loop that is mapped to two phases, before transformations (left) and after (right). Values for I_1 and I_2 are explained in Section 5.1.3.

blocks and (possibly) other phases. The fundamental difference between executing basic blocks within a single phase and executing across multiple phases is that with multiple phases, *the server loses track of PC' when the server breaks out of a phase prematurely.* This happens when $I_k < I'_k$ for some phase $\mathcal{P}_k$ and can be prevented in program-specific circumstances[4]. In general, however, the server cannot detect if $I_k < I'_k$ for some k because PC' is encrypted. The server only knows when it decides to transition from one phase to another.

Because of ambiguity in each phase's iteration count, we compute on hierarchical programs with the following goals in mind:

1. **Safety and recovery.** If the server breaks out of a phase $\mathcal{P}_k$ too early, any useful computation performed up to that point should not be clobbered (*safety*). Furthermore, if some time after breaking out of $\mathcal{P}_k$ the server decides to run $\mathcal{P}_k$ for more iterations, $\mathcal{P}_k$ should be able to pick up where it left off (*recovery*).

2. **Efficiency.** While the server is executing within a phase, the system should get the efficiency advantage described in Section 4.4.

With this in mind, a hierarchical program is evaluated using the state update function (Eqns. 3 and 4) from Section 4.2, after a change is made to each equation:

Eqn. 4: evaluating phases in levels. Suppose we start executing in some phase $\mathcal{P}_k$. Each level in $\mathcal{P}_k$ is now made up of basic blocks and possibly other phases. To evaluate a basic block within a level, we evaluate the basic block's term in Eqn. 4 as before. To evaluate a phase $\mathcal{P}_j$ within a level, the server enters $\mathcal{P}_j$ and performs I_j iterations worth of work in $\mathcal{P}_j$. The c_j term (multiplied to $\mathcal{P}_j$) is re-defined to

$$\text{if } \mathrm{PC}(\mathcal{P}_j) \stackrel{?}{=} PC' \vee \mathrm{PC}_{\mathrm{entry}}(\mathcal{P}_j) \stackrel{?}{=} PC' : \; c_j = 1 \qquad (5)$$
$$\text{else} : c_j = 0$$

where $\mathrm{PC}(\mathcal{P}_j)$ is the PC of the *header* basic block in $\mathcal{P}_j$ and $\mathrm{PC}_{\mathrm{entry}}(\mathcal{P}_j)$ is the PC of the *entry* basic block in $\mathcal{P}_j$ (e.g., after the *done* flag is initialized to $false$). If $\mathrm{PC}(\mathcal{P}_j) \stackrel{?}{=} PC'$ holds, it was time to evaluate $\mathcal{P}_j$. If $\mathrm{PC}_{\mathrm{entry}}(\mathcal{P}_j) \stackrel{?}{=} PC'$ holds, $\mathcal{P}_j$ was run at some point in the past but the server transitioned out of $\mathcal{P}_j$ too soon (i.e., $I_j < I'_j$ and $done_j \stackrel{?}{=} false$ holds). In this case, $\mathcal{P}_j$ can continue executing where it left off. Note that by "stepping into" $\mathcal{P}_j$ at step s, $\mathcal{M}_s$ must be saved while $\mathcal{P}_j(\mathcal{M}_s)$ is being evaluated. After evaluating $\mathcal{P}_j$, PC' is updated using the *transition check* as described in Section 4.3.

Eqn. 3: dominators in multi-phase programs. Suppose we enter an arbitrary phase $\mathcal{P}_j$ (whose levels are given by $\mathcal{L}$) at some step s and have run $s' \geq 1$ steps in $\mathcal{P}_j$ so far; for simplicity, we assume that each level in $\mathcal{P}_j$ takes a single step to evaluate, regardless of whether it has inner phases. The multiplication optimization given by Eqn. 3 can only be applied in level $l = s' \bmod |\mathcal{L}|$ if $\mathrm{dominator}(\mathcal{L}_{l,1})$ holds and no *uncertain* inner phase was evaluated in any level from step $s \ldots s+s'-1$. (By *uncertain*, we mean a phase whose iteration count was not run for a guaranteed upper bound number of iterations by the server). We can make this optimization whenever we enter a phase (or inner phase) $\mathcal{P}_j$ because we evaluate $c_j * \mathcal{P}_j$—i.e., we condition on c_j and can therefore assume that PC' points to the start of $\mathcal{P}_j$. We cannot make this optimization in $\mathcal{P}_j$ after an uncertain inner phase within $\mathcal{P}_j$ has been evaluated because the server cannot assume that the inner phase completed, and therefore cannot assume any value for PC' while in $\mathcal{P}_j$.

Noise while evaluating phases. Suppose that we are currently evaluating $\mathcal{P}_1$ (from Figure 3) and will now evaluate $c_2 * \mathcal{P}_2(\mathcal{M}_s)$ where s is arbitrary. The noise on c_2 is proportional to the noise on PC' after evaluating basic block A (call this PC'_A). Thus, the noise on variables modified within $\mathcal{P}_2$ will increase proportionally to the noise on PC'_A after $c_2 * \mathcal{P}_2(\mathcal{M}_s)$ completes. If $\mathcal{P}_2$ modifies a large amount of state, PC'_A may need to be explicitly refreshed before applying the $c_2 * \mathcal{P}_2$ operation. After evaluating $c_2 * \mathcal{P}_2$, the noise on PC'_A increases in a manner proportional to the noise on done_2 (i.e., the *done* flag from $\mathcal{P}_2$) because of the *transition check*.

5.1.2 Computation Stalls

While executing within a phase $\mathcal{P}_j$, efficiency is maximized when $I_j = I'_j$ (I'_j is the actual iteration count). If $I_j > I'_j$, the server performs extra work in $\mathcal{P}_j$ that does not make forward progress but $\mathcal{P}_j$ completes. If $I_j < I'_j$, the server *stalls* in $\mathcal{P}_j$ (e.g., PC' gets stuck at $\mathrm{PC}_{\mathrm{entry}}(\mathcal{P}_j)$) and doesn't make forward progress until $\mathcal{P}_j$ is run for additional iterations (we call this a *recovery*).

The way we re-define c_j for phases, in Eqn. 5, guarantees safe execution but only limited recovery if the server stalls. For example, consider a small change to the transformed two-level loop from Figure 3 where $\mathcal{P}_1$ is itself an inner-loop in some phase $\mathcal{P}_3$. If the server underestimates I_2 during the *final* iteration of $\mathcal{P}_1$, PC' gets stuck at $PC(E)$, the PC for basic block E, and $\mathrm{done}_2 \stackrel{?}{=} false$ holds. Since this happened in the last iteration of $\mathcal{P}_1$, $\mathrm{done}_1 \stackrel{?}{=} false$ will hold as well. Now, when $\mathcal{P}_3$ tries to evaluate $\mathcal{P}_1$ again (to attempt to recover), c_1 in $c_1 * \mathcal{P}_1(\mathcal{M})$ is guaranteed to equal 0 because $PC' = PC(E)$. Thus, the server is stalled and cannot recover.

5.1.3 Probabilistic Efficiency Through Recovery

The server can trade-off confidence that it has not stalled with efficiency by using probabilistic techniques and the recovery mechanism from Section 5.1.1. In the previous section (5.1.2), the server stalled because it didn't know I_2 precisely. Suppose that the server knows I_1, the outer phase's iteration count, and can model I_2 as a

[4]For example, if the server can determine an upper-bound on the size of a and b in GCD$\star$ based on ciphertext length, it may be able to run for a conservative number of iterations.

random variable with a known probability distribution. The server wants to know how many iterations of $\mathcal{P}_1$ (call this value I_1') it will have to perform if it picks a fixed value for I_2 and wants to stall with very small probability.

Consider the following scenario. The server knows that $I_1 = 100$ and that $I_2 \sim \text{Binomial}\,(p = .5, n = 10)$. Thus, the upper bound on the number of iterations $\mathcal{P}_2$ has to perform overall is $100 * 10 = 1000$; if the server runs for this long, it is guaranteed to not stall. To decrease absolute computation, the server decides to run $\mathcal{P}_2$ for 5 iterations always (even though it knows that I_2 is a random variable where $Pr(I_2 > 5) \approx .37$). Because of the recovery mechanism, the server will finish $\mathcal{P}_2$ after one (if $I_2 \leq 5$) or two (if $I_2 > 5$) tries. The server can model how many times it will need to try twice as $\mathbf{F} \sim \text{Binomial}\,(p = .37, n = 100)$. From this, the server can calculate that $Pr(\mathbf{F} > 60) \approx 9 \times 10^{-7} = \epsilon$; the server has chosen 60 to make the probability very small. Thus, to finish $\mathcal{P}_1$ with $1 - \epsilon$ probability, the server needs to to run $\mathcal{P}_1$ for $I_1' = 2 \times 60 + (100 - 60) = 160$ iterations. Even though the server's view of I_2 was fuzzy, the server established with high confidence that it could finish $\mathcal{P}_1$ while performing $160 * 5 = 800$ iterations of $\mathcal{P}_2$ instead of 1000.[5]

5.2 Data-Dependent Recursion

If $\mathcal{A}$ has data-dependent recursion, we say that the recursive calls in $\mathcal{A}$ will form a *call stack* $\mathcal{F}$, made of up to T *call frames* (for some T), where each frame is denoted $\mathcal{F}_t$ for $t = 1 \ldots T$. Call frames are pushed onto ($\mathcal{F}_t$ transitions to $\mathcal{F}_{t+1}$) and popped off ($\mathcal{F}_t \rightarrow \mathcal{F}_{t-1}$) of the call stack based on data-dependent conditions. Data-dependent recursion is problematic under encryption because (a) whether a $\mathcal{F}_t \rightarrow \mathcal{F}_{t+1}$ or $\mathcal{F}_t \rightarrow \mathcal{F}_{t-1}$ transition occurs depends on encrypted data and (b) when $\mathcal{F}_t \rightarrow \mathcal{F}_{t-1}$ transitions occur, control flow may return to multiple places based on where the recursive call took place in the program.

The key idea that we use to efficiently path levelize recursive programs is to *decouple the call stack from the operation performed in every call frame*. Formally, if $\mathcal{A}$ denotes the recursive algorithm and $\mathcal{F}_s$ is the call frame that will be computed on at step s (assume that each frame is evaluated in 1 step for simplicity), this transformation creates the following frame update function:

$$\mathcal{F}_{s+1} = \mathcal{A}(\mathcal{F}_s)$$

where $\mathcal{F}_{s+1}$ can be used as the call frame for the next call to $\mathcal{A}$ at step $s + 1$. Note that if $\mathcal{F}_s$ corresponds to $\mathcal{F}_t$ (the t^{th} call frame), $\mathcal{F}_{s+1}$ will correspond to $\mathcal{F}_{t+t'}$ where $(-t < t' \leq 1) \wedge (t' \neq 0) \wedge (t + t' \leq T)$. $t' < -1$ is allowed so that the program can optimize tail calls.

5.3 Example: 3-Satisfiability

With recursion and phase hierarchy in hand, we are ready to transform a recursive backtracking 3-Satisfiability (3SAT) algorithm and show how to run it under encryption. This algorithm (shown in Algorithm 2) is given a 3CNF formula (a set of clauses F) and finds the satisfying assignment if one exists or reports that no satisfying assignment exists (by returning $false$). Throughout this section, N represents the number of unique variables in F.

For simplicity, we consider a simple backtracking algorithm that assigns variables in a fixed (and public) order O. A stack data-structure st, which implicitly maintains a head-of-stack pointer denoted $st.h$ ($st.h = 0$ means the stack is empty), keeps track of partial assignments to the 3CNF. O determines which variable is added to/removed from the stack when a push/pop occurs. This version of 3SAT uses the following helper functions:

$\text{stackFull}(N, st)$: Returns $true$ if $N \stackrel{?}{=} st.h$ holds, and $false$ otherwise.

$\text{push}(st, true/false)$: Conceptually pushes $true/false$ onto st. This operation writes $true/false$ onto st at position $st.h$; $st.h$ then gets incremented and the new st is returned.

$\text{canBeSAT}(F, st)$: Returns $true$ if F can possibly be SAT given st, and $false$ otherwise.

For the rest of this section, we assume that $\text{canBeSAT}()$ is implemented as a loop over the clauses in F, where each iteration can perform random accesses to st. Note that we do not need a $\text{pop}()$ function to manage st—this is done implicitly through recursive calls.

Conceptually, the $\text{canBeSAT}()$ routine tries to prune subtrees when possible, making the search tree that gets traversed depend on F. This makes evaluation under encryption difficult because the program call stack's behavior depends on F; note that when 3SAT returns, control may be at line 7 or 11 in Algorithm 2 (3SAT, top). Levelizing 3SAT using only techniques from Section 4 doesn't help, as each recursive call may take an exponential number of steps to return. In fact, using the methodology of Section 3.5 would be more efficient.

5.3.1 3SAT Transformation

To transform 3SAT, we first inline helper functions and transform inner loops into phases. Inlining $\text{stackFull}()$ and $\text{push}()$ is straightforward—both are straight-line code. $\text{canBeSAT}()$ is a loop over F, but has a guaranteed upper bound on its iteration count—a maximum 3CNF size, F'. To properly manage the stack st under recursion, we also require a bound N' on the number of unique variables in the 3CNF. F' and N' are public and must be agreed upon by the user and the server. When $\text{canBeSAT}()$'s current iteration reaches $|F|$, the *done* flag is toggled to $true$. Past that point, $\text{canBeSAT}()$ is in fixed point mode. With this transformation in hand, we will treat $\text{canBeSAT}()$ as a single basic block for the rest of the section.

Next, to handle the data-dependent recursive calls in 3SAT, we *decouple the call stack from the operation performed at every node in the SAT search tree* (the high-level idea is shown at the top of Figure 4). The transformed algorithm, 3SAT$\star$, is shown in psuedocode in Algorithm 2 and in control-flow form in Figure 4. d implements the *done* flag and r is a flag to indicate whether the formula is SAT or not SAT. 3SAT$\star$ uses the following helper functions, in addition to those introduced in Section 5.3:

$\text{pop}(st)$: Conceptually pops an element off of st. This operation decrements $st.h$ and then returns the element from st at the new position $st.h$.

$\text{allFalse}(st)$: Returns $true$ if every variable assignment in st is $false$ and returns $false$ otherwise.

$\text{backtrack}(st, n)$: Sets $st.h = n$ and then evaluates and returns $\text{push}(st, false)$. In Algorithm 2, this function conceptually pops off elements from st until it sees a $true$, and then it replaces that $true$ with a $false$.

The call stack for 3SAT$\star$ consists of st (which is the same as st in Section 5.3) and an independent stack ra which maintains the position of each assignment to $true$ in st. The $\text{backtrack}()$ function

[5]Note that I_1' holds even if basic block A transitions to basic block C for some of the iterations in $\mathcal{P}_1$. This is because evaluating C will always take one step (or "try.")

performs the recursive function returns that were handled implicitly in the original algorithm. ra is used to efficiently implement the backtrack() routine—this can be seen as a form of tail call optimization.

Algorithm 2 The original (recursive) and transformed 3SAT algorithms. Explanations of helper functions are given in Sections 5.3 and 5.3.1. The main loop in $\mathcal{A}$ is shown between lines 4 and 23 in 3SAT⋆.

3SAT(N, F, st)**:**
if stackFull$(N, st) \wedge$ canBeSAT(F, st) **then**
  return st // the satisfying assignment
else if stackFull(N, st) **then**
  return $false$
end if
SAT = 3SAT$(N, F,$ push$(st, true))$
if SAT $\neq false$ **then**
  return SAT
else
  return 3SAT$(N, F,$ push$(st, false))$
end if

3SAT⋆(N, F, st, ra)**:**
d = $false$
r = $false$
for i = 0; i < I; i++ **do**
  SAT = canBeSAT(F, st)
  if SAT $\wedge$ stackFull(N, st) **then**
    d = $true$
    r = $true$
  end if
  if allFalse$(st) \wedge$ SAT $\stackrel{?}{=} false$ **then**
    d = $true$
    r = $false$
  end if
  if d $\stackrel{?}{=} false$ **then**
    if SAT **then**
      ra = push$(ra, st.h)$
      st = push$(st, true)$
    else
      n = pop(ra)
      st = backtrack(st, n)
    end if
  end if
end for
return d, r, st, ra

If 3SAT⋆ is run for I iterations and we can determine if F is SAT in I' iterations, we have the following cases:

$I \geq I'$: If F was SAT: $d = true$, $r = true$, and st is the satisfying assignment. Otherwise: $d = true$, $r = false$, and $st =$ invalid.

$I < I'$: $d = false$. $r =$ invalid. st is the last variable assignment that was tested.

5.3.2 Transformed 3SAT to FHE

To evaluate 3SAT⋆ under encryption, the user initially sends a stack st (containing a single element, initialized to $true$), ra (which contains a single element that points to the first element in st), F, N and the initial PC' as the encrypted $\mathcal{M}_0$. The server

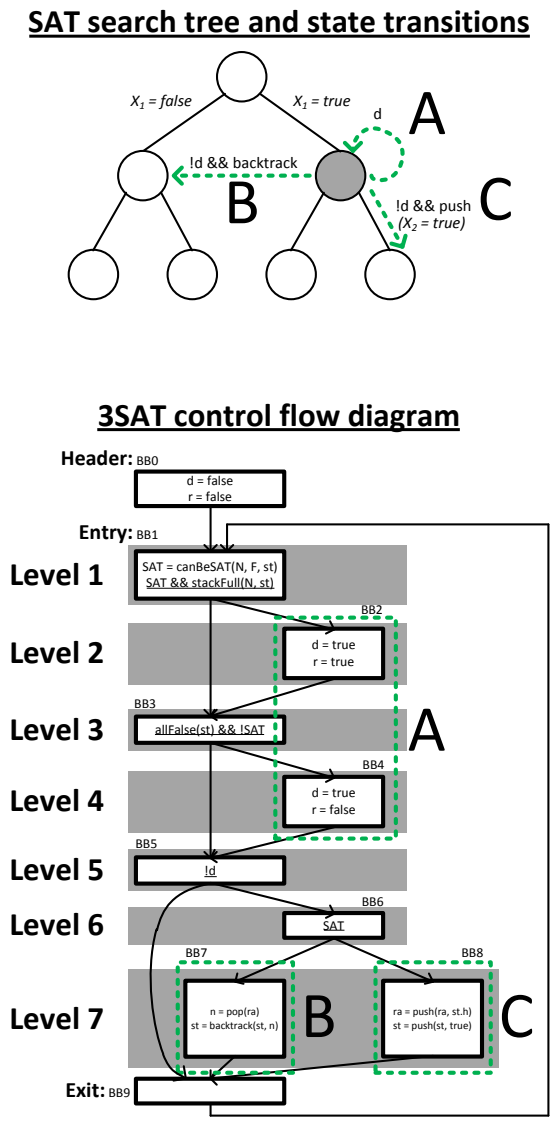

Figure 4: The top shows the SAT search tree overlaid with the work done by one iteration of the 3SAT⋆ main loop. The bottom shows the control flow graph for 3SAT⋆. In the top part, each loop iteration performs work on one node of the search tree (the shaded circle, or current state). Each iteration, state transitions to other nodes (dashed green arrows) are evaluated and one is taken. Each state transition is labeled **A**, **B** or **C** to show correspondence to the basic blocks in the bottom diagram that perform the same operation. $BB9$ is not counted as a level and is only shown to conceptually give the 3SAT⋆ loop a single exit point.

synthesizes

$$\mathcal{L}_{3SAT} = [\{BB1\}^{\bullet}, \{BB2\}, \{BB3\}^{\bullet}, \{BB4\}, \{BB5\}^{\bullet}, \{BB6\}, \{BB7, BB8\}]$$

where '$^{\bullet}$' annotations follow the meaning from Section 4.2.

To start the computation, the server evaluates $BB0$ to initialize d. After evaluating $\mathcal{L}_{3SAT}$ I times, the server performs the *transi-*

tion check (Section 4.3) to finalize PC' and then returns encrypted values for the current st, r, and PC'.

As before, the user decrypts PC' and checks $PC' \stackrel{?}{=} PC_{\text{done}}$. If this condition holds, the user decrypts r. If r is $true$, the user decrypts st—which is the satisfying assignment. If r is $false$, the user knows that F was not SAT. If $PC' \stackrel{?}{=} PC_{\text{done}}$ did not hold, the user has received an intermediate result.

Broadly speaking, the program transformations for 3SAT are non-obvious and significantly more sophisticated than the transformation applied to GCD. The benefit is that $\mathcal{L}_{3SAT}$ is independent F and that the largest level in $\mathcal{L}_{3SAT}$ ($\{BB7, BB8\}$, with an additional identity operation that is not shown) is significantly smaller than the number of basic blocks in the program.

Any program within the scope of programs we have considered can theoretically be written in path-levelized and fixed point form, though it may be quite difficult to *automatically* transform arbitrary program descriptions into such a form. In cases when automatic transformation of a program or part of a program is difficult, one can revert to the mechanical, albeit less-efficient, form introduced in Section 3.5.

6. PRIVACY LEAKAGE OF APPROACH

We show that the privacy leakage in our approach is optimal. We start by considering the two-interactive protocol described in Section 3.1. We prove that if the server is able to figure out when the computation is finished (such that the user always receives the final result), then the underlying FHE can be broken in that the server obtains a means to decrypt any message of its liking. This means that intermediate results cannot be avoided, unless the user can guarantee an upper bound on the number of steps required for the computation. Without this guarantee, to obtain the final result, the user may need to interact more than two times by asking for more computation and this necessarily leaks some additional privacy. At the end of this section, we discuss multi-interactive protocols.

6.1 Privacy of Two-Interactive Protocols

In the two-interactive protocol Π of Section 3.1, we assume that $\mathcal{A}$ imposes a known relation on the sizes (the length of the bit representations) of x and y, i.e., $|y| = f(|x|)$ for some invertible function f. E.g., $|y| = 0$ if the server does not supply any inputs, or y represents some part of a database over which the user wishes to search, or $|y|$ indicates a number of random bits, proportional to $|x|$, supplied by the server. The server only receives the pair $(\mathcal{M}, S)$ from the user and no other messages that depend on some intermediate computations performed by the server. This means that the only information about $\mathcal{A}(x, y)$ given to the server in Π comes from the pair $(\mathcal{M}, S)$ and its own input y. The view of the server $\texttt{view}^{\Pi}_{\texttt{server}}(x, y)$ is defined as the triple $(\mathcal{M}, S, y)$.

Privacy w.r.t. semi-honest server behavior is defined as follows [16]. We say that Π privately computes (an intermediate result of) $\mathcal{A}$ if there exists a probabilistic polynomial time (ppt) algorithm, denoted P, such that

$$\{P(y)\}_{x,y\in\{0,1\}^*} \stackrel{S}{\equiv} \{\texttt{view}^{\Pi}_{\texttt{server}}(x, y)\}_{x,y\in\{0,1\}^*} \quad (6)$$

where $|y| = f(|x|)$.

Since the used FHE scheme is assumed to be secure, $\mathcal{M}$ cannot be distinguished from any other ciphertext of size $|\mathcal{M}|$. Hence, P can simulate $\mathcal{M}$ by encrypting a random bit string of size $|x| = f^{-1}(|y|)$. Only if the user in Π computes the number of steps S by using some ppt algorithm $\mathcal{A}'$ (based on $\mathcal{A}$) on inputs $\mathcal{M}$ and y, P is able to simulate S. We conclude that Π privately computes $\mathcal{A}$ if Π uses a ppt algorithm $\mathcal{A}'$ to compute the number of steps $S \leftarrow \mathcal{A}'(\mathcal{M}, y)$ based on inputs $\mathcal{M}$ and y. E.g., S can be a function of $|x| = f^{-1}(|y|)$.

In the GCD example the user wants to outsource the computation of the greatest common divisor of a and b while keeping as much information about a and b hidden. If each bit of a and b is encrypted separately, the sizes of a and b leak. This does not mean that the computation under encryption is not private; the computation itself is private in that no extra privacy is leaked beyond the encryptions of a and b. To limit privacy leakage through its inputs, the user may arbitrarily increase the sizes of a and b by adding encryptions of dummy zeroes to represent extra most significant bits.

Suppose that a and b are a-priori set to say 100-bit representations such that the sizes of compact bit representations do not leak, and therefore no information about the ranges of a and b leaks. The GCD is privately computed if the number of steps does not depend on the actual integer values of a and b. E.g., the user knows that $O(\log b)$ iterations are enough to complete the GCD algorithm. If the user's strategy (which is known to the server) is to ask the server to compute for $S = O(\log b)$ cycles, then the server knows that b is in the range 2^S. Even though the encryption of the 100-bit representation of b keeps the range of b private, S reveals the range of b.

Similarly, for the 3SAT problem, the user may decide on a fixed strategy, where he always asks for exactly 1 hour of computation. If the user receives an intermediate result, he concludes that the 3SAT problem is likely not satisfiable.

6.2 Teaching the Server when to Stop

We may want to improve the two-interactive protocol such that the user is always guaranteed to receive the final result $\mathcal{A}(x, y)$. This is possible if there exists an upperbound $u(|x|, y)$ on the total number of iterations required to compute $\mathcal{A}(x, y)$ under encryption; the user chooses the number of steps S such that the server executes at least $u(|x|, y)$ iterations. In order to compute $u(|x|, y)$, the user may need to ask the server (assumed to be semi-honest) information about y. Since the number of steps is a function of $|x|$ and y, no additional privacy is leaked.

For the 3SAT problem, such an upperbound is exponential in $|x|$. On average the 3SAT problem may be solved much sooner (much less than, e.g., the 1 hour computation the user is willing to pay for). For this reason we want to give the server some algorithm $\mathcal{D}$ to decide whether the computation under encryption reached the final state. However, we will show that an honest-but-curious server can use such an algorithm $\mathcal{D}$ to decrypt any ciphertext of its liking.

Let $I(x, y)$ denote the exact number of iterations needed by $\mathcal{A}(x, y)$ to finish its computation under encryption. We assume that x and y represent inputs for which $I(x, y) < S$. I.e., the computation of $\mathcal{A}(x, y)$ finishes in less the maximum number of steps the user is willing to pay for. In other words x and y represent an instance for which $\mathcal{D}$ reduces the cost for the user.

We first consider the case which supposes that

$$\exists_{k>0}\ Prob_{x"\ \text{s.t.}\ |x"|=|x|}[I(x", y) \neq I(x, y)] > 1/|x|^k. \quad (7)$$

Suppose that the server executes $\mathcal{A}$ under encryption on inputs $\mathcal{M}$ and y and uses the encrypted state as input to $\mathcal{D}$ to decide whether the computation is finished or not. Suppose that the computation finishes after exactly $i = I(x, y) < S$ iterations within the allotted number of steps. Then, the server is able to learn i by using $\mathcal{D}$.

Suppose the server constructs an encryption e of 1. With a slight abuse of notation, let $x = (x_1, \ldots, x_m) \in \{0, 1\}^m$ and let $\mathcal{M} = (\mathcal{M}_1, \ldots, \mathcal{M}_m)$ be a bit by bit encryption of x, i.e., $\mathcal{M}_j$ is a ciphertext of bit x_j. By adding e to arbitrary positions

in $\mathcal{M}$, an encryption of some random bit string x" is constructed: $\mathcal{M}+(x"-x)e$ is the bit by bit encryption of $x+(x"-x)\cdot 1 = x"$. See Eqn. 7—the server only needs to try a polynomial number $|x|^k$ possible values $x"$ in order to find one with $I(x", y) \neq I(x, y)$.

If $I(x", y) \neq I(x, y)$, then the computation based on the encryption of $x"$ and y does not finish after exactly $i = I(x, y)$ iterations. As soon as this happens, any ciphertext w can be decrypted as follows. The server adds w instead of e to the selected positions in $\mathcal{M}$. This results in $\mathcal{M}+(x"-x)w$ which is a bit by bit encryption of $x + (x" - x) \cdot 0 = x$ if w is an encryption of 0. It results in the encryption of the bit string $x + (x" - x) \cdot 1 = x"$ if w is an encryption of 1. So, if the computation of $\mathcal{A}$ on the encrypted input $\mathcal{M} + (x" - x)w$ takes exactly i iterations, then w is an encryption of 0. If this computation takes less or more iterations, then w is an encryption of 1.

We conclude that either algorithm $\mathcal{D}$ enables the server to decrypt any ciphertext of its liking or Eqn. 7 is not true.

Notice that if Eqn. 7 is not true, then $I(x", y) = I(x, y)$ with probability $> 1 - negl(|x|)$. This means that, except for a negligible fraction of inputs x, the user knows a precise tight upper bound $u(|x|, y)$ that can be used to choose the number of steps. This means that algorithm $\mathcal{D}$ does not help in improving the protocol's performance.

We conclude that either algorithm $\mathcal{D}$ enables the server to decrypt any ciphertext of its liking or does not help in improving the protocol's performance. A two-interactive protocol for computing $\mathcal{A}$ cannot guarantee for general $\mathcal{A}$ without additional privacy leakage that the user receives the final result and not an intermediate one.

6.3 Multiple Interactions

If a user learns an intermediate result after two interactions, it may decide to ask the server to continue its computation for another number of steps. This allows the user to obtain the final result after multiple interactions. The disadvantage is that the decision to continue the computation or not is based on an intermediate result and therefore leaks some privacy about the final result.

In general, the server learns the number of client-interactions j needed for computing $\mathcal{A}(x, y)$. Since the server knows the protocol, $\mathcal{A}$ and its own input y, the server is able to simulate for arbitrary $x"$ the number of interactions that would be needed for computing $\mathcal{A}(x", y)$ under encryption in the protocol. If this is not equal to j, then the server knows that $x \neq x"$. This may lead to undesired privacy leakage: The set of typical (most likely) user inputs can be partitioned into bins B_j where $x \in B_j$ if the computation of $\mathcal{A}(x, y)$ takes j interactions. The bin structure is revealed to the server.

In protocols with multiple interactions, besides asking the server to continue its computation, the client may ask the server to use the last state of its computation as input to a second computation. Even though this likely leads to efficiency improvements, the disadvantage is more privacy leakage; by simulating the protocol, the server is able to partition the user inputs not only according to the number of interactions but also according to the requested computations for each interaction.

7. MALICIOUS SERVERS

In the semi-honest model we assume the server to be honest in executing the protocol and curious in that the server tries to learn as much about the final result as possible.

If the server is malicious, then we need to take care of additional problems:

Verification of Computation: The user needs to verify whether algorithm $\mathcal{A}$ was executed by the server (and evaluated on the correct inputs). For NP-complete problems the user is able to verify a positive answer in polynomial time. If the final result cannot be verified in polynomial time, then check pointing may help: the server commits to all intermediate results (by transmitting to the user the root of a Merkle tree over all intermediate results) after which the client asks for a few intermediate results, checks the commitment and verifies their consistency.

Another approach is to encrypt algorithm $\mathcal{A}$ and use a universal Turing machine approach to evaluate $\mathcal{A}$ on its encrypted inputs. The user may ask the server to execute a number of encrypted algorithms, one of them being $\mathcal{A}$. If the user knows the final results for the other algorithms, then the user is able to check these. This will gain trust in the correct evaluation of $\mathcal{A}$.

Rather than asking the server to execute multiple encrypted algorithms, the user may construct a new algorithm $\mathcal{A}''$ that interleaves (in an unpredictable way based on its input) the execution of $\mathcal{A}$ with some other algorithm $\mathcal{A}'$, which is a way of verifying the execution of $\mathcal{A}$. If the server evaluates the encryption of $\mathcal{A}''$ by using a universal Turing machine approach, then the server does not know which iteration corresponds to the execution of $\mathcal{A}$. Therefore, if the server tampers with one of the iterations, there is a significant probability that an iteration of $\mathcal{A}'$ is tampered with and this can be detected by the user.

Guaranteeing Fairness: If the user pays for the outsourcing of $\mathcal{A}$, then the server needs to guarantee fairness in that the number of performed steps can be verified by the user. We need to bind an iteration counter to the state of $\mathcal{A}$. E.g., algorithm $\mathcal{A}'$ (see above) can be a simple counter through which the user is able to obtain an estimate of the total number of completed steps (e.g., if the interleaving is on average 1 iteration of $\mathcal{A}$ on 1 iteration of $\mathcal{A}'$, then twice the final counter value of $\mathcal{A}'$ leads to a good estimate of the total number of completed steps).

Notice that using a universal Turing machine approach leads to circuit privacy; since algorithm $\mathcal{A}$ is encrypted and used as input to the universal Turing machine, the working of $\mathcal{A}$ remains private. However, using a universal Turing machine to obfuscate the workings of $\mathcal{A}$ costs much more overhead.

Rather than encrypting $\mathcal{A}$ and using it as input to a universal Turing machine, we may partially obfuscate and encrypt $\mathcal{A}$. E.g., the user may use control bits to indicate whether gates represent addition or multiplication. The user transmits the encryption of each control bit to the server. This means that the interleaving of $\mathcal{A}$ with other algorithms (used for counting steps etc.) becomes obfuscated. If obfuscated sufficiently, the computation can be verified, fairness can be checked and the algorithm itself remains partially private.

8. CONCLUSION

In this paper we take a first step toward building a compiler for encrypted computation of general programs. This problem has not received much attention to date, with most encrypted computation examples being restricted to fixed-iteration loops or Boolean circuits (e.g., AES).

We have addressed issues arising from complex control flow in programs and provided a mechanical, albeit inefficient, method for encrypted computation. We have shown how to improve efficiency

by reducing the *PC ambiguity* seen by the server while maintaining optimal leakage in a two-interactive protocol. The two techniques of path levelization and fixed point computation are broadly applicable to programs. Using two examples, GCD and 3SAT, we have shown how these techniques significantly reduce the absolute computation required in encrypted computation.

Future work involves building an optimizing compiler front-end for automatic transformation of programs for efficient encrypted computation. While we have treated FHE as a black box, it is worthwhile to investigate how a compiler back-end that assumes a particular FHE scheme can be built to improve efficiency further.

9. REFERENCES

[1] Alex Bain, John Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. A domain-specific language for computing on encrypted data. In *FSTTCS 2011*. LIPIcs, December 2011. Invited paper.

[2] Z. Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. IACR eprint archive, 2012.

[3] Z. Brakerski, G. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS 2012*, pages 309–325, 2012.

[4] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *FOCS'11*, 2011.

[5] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *CRYPTO'11*, 2011.

[6] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi. Fully homomorphic encryption over the integers with shorter public-keys. In *Crypto'11*, 2011.

[7] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *EUROCRYPT'10, LNCS 6110, Springer*, pages 24–43, 2010.

[8] J. Fan and F. Vercauteren. Somewhat Practical Fully Homomorphic Encryption. IACR eprint archive, 2012.

[9] C. Gentry. A fully homomorphic encryption scheme. PhD thesis, Stanford University, 2009.

[10] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC'09*, pages 169–178, 2009.

[11] C. Gentry and S. Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. In *FOCS'11*, 2011.

[12] C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *EUROCRYPT'11, LNCS 6632, Springer*, pages 129–148, 2011.

[13] C. Gentry, S. Halevi, and N.P. Smart. Fully homomorphic encryption with polylog overhead. IACR eprint archive, 2012.

[14] C. Gentry, S. Halevi, and N.P. Smart. Homomorphic Evaluation of the AES Circuit. IACR eprint archive, 2012.

[15] W. F. Gilreath and P. A. Laplante. *Computer Architecture: A Minimalist Approach*. Springer, 2003.

[16] O. Goldreich. *Foundations of Cryptography: Volume II (Basic Applications)*. University Press, 2004.

[17] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[18] R. Lindner and C. Peikert. Better Key Sizes (and Attacks) for LWE-Based Encryption. In *Topics in Cryptology - CT-RSA 2011, LNCS 6558, Springer*, pages 319–339, 2011.

[19] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT'10, LNCS 6110, Springer*, pages 1–23, 2010.

[20] D. Malkhi, N. Nisan, B. Pinkas, and Yaron Sella. Fairplay: a secure two-party computation system. In *Proceedings of the 13th Usenix Security Symposium*, pages 287–302, 2004.

[21] J.C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman. Information-flow control for programming on encrypted data. Cryptology ePrint Archive, Report 2012/205, 2012. http://eprint.iacr.org/.

[22] M. Naehrig, K. Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *CCSW 2011, ACM*, pages 113–124, 2011.

[23] R. Rivest, L. Adleman, and M.L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, pages 169–180, 1978.

[24] N.P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *PKC'10, LNCS 6056, Springer*, pages 420–443, 2010.

[25] N.P. Smart and F. Vercauteren. Fully Homomorphic SIMD Operations. IACR eprint archive, 2011.

[26] D. Stehle and R. Steinfeld. Faster Fully Homomorphic Encryption. In *Advances in Cryptology - ASIACRYPT 2010, LNCS 6477, Springer*, pages 377–394, 2010.

[27] A.C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

APPENDIX

A. TURING MACHINE UNDER ENCRYPTION

We formally show how a Turing machine operation can be transformed into an arithmetic circuit that can be evaluated under encryption. Since the state of the Turing machine is encrypted, the server needs to assume that any state is possible. For this reason, all possible state transitions need to be considered in the arithmetic circuit.

For completeness, we first explain Turing machines. Next we show how a Turing machine operation can be transformed into an arithmetic circuit and we perform an asymptotical analysis of its run time under encryption (based on ring-LWE FHE).

A.1 The Turing Machine

The following definition and discussion is taken from [17].

A Turing machine consists of a finite-state control unit and a tape. Communication between the two is provided by a single head, which reads symbols from the tape and is also used to change the symbols on tape. The control unit operates in discrete steps; at each step it performs two functions in a way dependent on its current state and the tape symbol currently scanned by the read/write head:

1. Put the control unit in the initial state.
2. Either:
   (a) Write a symbol in the tape square currently scanned, replacing the one already there; or
   (b) Move the read/write head one tape square to the left or right.

The tape has a left end, but it extends indefinitely to the right. Initially, an input string is inscribed on tape squares at the left end of the tape and the rest of the tape contains blank symbols, denoted by $\#$. There is a special halt state, denoted by h, that is used to signal the end of a computation. At the end of a computation the tape represents the output or answer. The distinct symbols L and R are used to denote movement of the head to the left or right.

A formal definition of a Turing machine M is as follows: A Turing machine is a quadruple (K, Σ, δ, s_I), where

- K is a finite set of states, not containing the halt state h,
- $s_I \in K$ is the initial state,
- Σ is an alphabet, containing the blank symbol $\#$, but not containing the symbols L and R,
- δ is a function from $K \times \Sigma$ to $K^+ \times \Sigma^+$, where K^+ denotes the set $K \times \Sigma$ and Σ^+ denotes the set $\Sigma \cup \{L, R\}$.

If $q \in K$, $a \in \Sigma$, and $\delta(q, a) = (p, b)$, then M, when in state q and scanning symbol a, will enter state p, and (1) if b is a symbol in Σ, rewrite a as b, or (2) if b is L or R, move its head in the direction of b. Since δ is a function, the operation of M is deterministic and will stop only when M enters the halt state or attempts to move left off the end of the tape.

A.2 Turing Machines as Arithmetic Circuits

Our goal is to show how general computations, like those defined by Turing machines, can be evaluated under encryption by using FHE. Since FHE is designed to evaluate arithmetic circuits under encryption, we first explain how a Turing machine that operates on a *finite tape* can be represented as an arithmetic circuit such that it can be used to evaluate encrypted inputs.

The Turing machine is fully described by its state $s \in K^+$, the position of its head, and the content of the tape. For a finite length tape bounded to n squares, its content from left to right is a sequence of values t^1, t^2, ..., t^n. Let z be the position of the head indicating that the Turing machine will read t^z during its next cycle.

We represent each symbol as a bit string: In order to achieve a flat arithmetic circuit (of constant depth, not depending on n) we represent integer z by $n+2$ bits: for $0 \le i \le n+1$, $z_i = 1$ if and only if the position of the head is equal to i, all other z_j's are equal to 0. For example, $z - 1$ is computed as a shift of the bit representation of z in the direction of the LSB. It is possible for the head to drop off the tape; positions $z = 0$ and $z = n + 1$ indicate that the head dropped off to the left and right respectively.

In a similar way, we represent s by a string of $|K^+|$ bits and each t^i by a string of $|\Sigma|$ bits. For example, the bit of s indexed by $q \in K^+$ equals $s_q = 1$ if and only if s represents the state q in K^+. In particular, s_h denotes the bit corresponding to the halt state.

If the head points at a position on the tape, then the following rules explain the operation of the Turing machine. For $z \in \{1, \ldots, n\}$:

U: If $\delta(s, t^z) = (s', t')$ with $t' \notin \{L, R\}$, then after the next cycle the new state is equal to s', the position of the head z stays the same, and the tape is only modified in position z where t^z is updated to t'. We write $s \leftarrow s'$ and $t^z \leftarrow t'$ for short.

L: If $\delta(s, t^z) = (s', L)$, then $s \leftarrow s'$ and $z \leftarrow z - 1$.

R: If $\delta(s, t^z) = (s', R)$, then $s \leftarrow s'$ and $z \leftarrow z + 1$.

We add an extra rule in case the position of the head drops off the tape. The position of the head drops off if either z has been decreased to 0 or z has been increased to $n + 1$. If this happens, then the following rule "freezes" the Turing machine giving the possibility to the user to "unfreeze" the Turing machine by increasing the tape length such that the computation can be picked up again at a later time.

F: If $z \in \{0, n + 1\}$, then nothing changes, i.e., there are no updates to the state, position of the head, or tape.

We will now translate rules U, L, R and F into an arithmetic network.

Tape updates: Only rule U updates the squares on the tape: There is exactly one index q for which $s_q = 1$, meaning that the state of the Turing machine is q. There is exactly one index i for which $z_i = 1$, meaning that the head is positioned above the ith square. There is exactly one index $a \in \Sigma$ for which $t^i_a = 1$, meaning that the ith square has value a. We first observe that this square is overwritten if (q, a) is in set

$$[K \times \Sigma]_b \triangleq \{(q, a) \in K \times \Sigma \,:\, \delta(q, a) = (., b)\} \text{ with } b \in \Sigma,$$

in which case $t^i_b \leftarrow 1$ and all other bits in t^i are set to 0. Notice that $(q, a) \in [K \times \Sigma]_b$ if and only if $\sum_{(q,a)\in[K\times\Sigma]_b} s_q t^i_a = 1$. So, the same update rule

$$t^i_b \leftarrow \sum_{(q,a)\in[K\times\Sigma]_b} s_q t^i_a \tag{8}$$

applies to all bit positions b in t^i.

Second, if $(q, a) \notin [K \times \Sigma]_b$ for any $b \in \Sigma$, then the tape is not updated at all. This condition is satisfied if and only if either q equals the halting state h (i.e., $s_h = 1$) or q is in the set

$$[K]_a \triangleq \{q \in K \,:\, \delta(q, a) = (., b') \text{ with } b' \in \{L, R\}\}.$$

Summarizing, the tape is not updated at all if q is in

$$[K^+]_a \triangleq [K]_a \cup \{h\}.$$

If $q \in [K^+]_a$, or equivalently $\sum_{q\in[K^+]_a} s_q = 1$, then t_a^i remains equal to 1. Since $1 = t_a^i \sum_{q\in[K^+]_a} s_q$, this can be expressed as $t_a^i \leftarrow t_a^i \sum_{q\in[K^+]_a} s_q$. For all the other $b \in \Sigma - \{a\}$, t_b^i remains equal to 0. Since $t_b^i = 0$, the same update rule

$$t_b^i \leftarrow t_b^i \sum_{q\in[K^+]_b} s_q \tag{9}$$

applies.

The first and second observation only hold for i with $z_i = 1$. If $z_i = 0$, then the bits in t^i are not updated. This leads to $t_b^i \leftarrow z_i\{(8) + (9)\} + (1 - z_i)t_b^i$ which expresses how tape values are updated: For $i \in \{1, \ldots, n\}$ and $b \in \Sigma$,

$$t_b^i \leftarrow z_i \left\{ \sum_{(q,a)\in[K\times\Sigma]_b} s_q t_a^i + t_b^i \sum_{q\in[K^+]_b} s_q \right\} + (1-z_i)t_b^i, \tag{10}$$

where addition is over bit strings without carry, i.e., addition is XOR.

State updates: State s is updated in rules U, L and R, and stays the same if rule F applies or if $s_h = 1$: Now we define

$$[K\times\Sigma^+]_p \triangleq \{(q,a) \in K\times\Sigma^+ \,:\, \delta(q,a) = (p,.)\} \text{ with } p \in K^+$$

and notice that if $z_i = 1$ and $\sum_{(q,a)\in[K\times\Sigma^+]_p} s_q t_a^i = 1$, then the state of the Turing machine changes into p. The state does not change if the head has dropped off the tape or if it is in the halting state. Let the indicator function $I(B)$ equal 1 if the Boolean statement B is true and 0 if B is false. Then, summarizing, for $p \in K^+$,

$$\begin{aligned} s_p \;\leftarrow\; & (z_0 + z_{n+1})s_p + \\ & \sum_{i=1}^{n} z_i \left\{ I(p = h)s_h + \sum_{(q,a)\in[K\times\Sigma^+]_p} s_q t_a^i \right\}, \end{aligned} \tag{11}$$

where all additions are over bit strings without carry.

Head position updates: The position of the head z only changes if rules R or L apply. Rule R increases z by 1 and rule L decreases z by 1. In our representation, if $z_i = 1$, then by rule R $z_i \leftarrow 0$ and $z_{i+1} \leftarrow 1$, and by rule L $z_i \leftarrow 0$ and $z_{i-1} \leftarrow 1$: We define

$$[K \times \Sigma]_L \triangleq \{(q,a) \in K \times \Sigma \,:\, \delta(q,a) = (.,L)\}$$

and

$$[K \times \Sigma]_R \triangleq \{(q,a) \in K \times \Sigma \,:\, \delta(q,a) = (.,R)\}.$$

Rule L is enforced if, for some $(q,a) \in [K \times \Sigma]_L$, $s_q = 1$ and $t_a^{i+1} = 1$ with $z_{i+1} = 1$, i.e., if the head reads out a and the Turing machine is in state q which implies that the head moves to the left to position i. Similarly, rule R is enforced if, for some $(q,a) \in [K \times \Sigma]_R$, $s_q = 1$ and $t_a^{i-1} = 1$ with $z_{i-1} = 1$ in which case the head moves to the right to position i. The head position z remains unchanged for all other cases: $s_h = 1$, z_0 or $z_{n+1} = 1$, or (q,a) with $s_q = 1$ and $t_a^i = 1$ and $z_i = 1$ is in

$$[K \times \Sigma]_\Sigma \triangleq \cup_{b\in\Sigma}[K \times \Sigma]_b.$$

Summarizing, for $i \in \{0, \ldots, n+1\}$,

$$\begin{aligned} z_i \;\leftarrow\; & I(i \le n-1)z_{i+1} \sum_{(q,a)\in[K\times\Sigma]_L} s_q t_a^{i+1} + \\ & I(i \ge 2)z_{i-1} \sum_{(q,a)[K\times\Sigma]_R} s_q t_a^{i-1} + \\ & I(1 \le i \le n)z_i \sum_{(q,a)\in[K\times\Sigma]_\Sigma} s_q t_a^i + \\ & (I(i=0) + I(i=n+1))z_i + I(1 \le i \le n)s_h, \end{aligned} \tag{12}$$

where all additions are over bit strings without carry.

Arithmetic network: An step of the Turing machine is fully described by the arithmetic network (10-12) with bit strings s, t^i for $1 \le i \le n$, and z as input. Notice that the network represents a network of (constant) depth 3: each bit is updated by a sum over $z_i s_q t_a^j$, which is a multi-variate polynomial of degree 3. This means that if we use FHE to evaluate the network under encryption, then each iteration cubes the "encryption noise" of each encrypted bit.

A.3 Evaluation under Encryption

We want to compute the overhead per iteration by the Turing Machine which is defined as the ratio between the computation needed to evaluate a single iteration under encryption and the computation needed to evaluate a single iteration by the Turing machine itself (without encryption).

We rely on the results in [3] where an FHE scheme is introduced with its security based on the hardness of the ring-LWE problem. The RWLE scheme of [3] with security parameter λ has the following properties:

- For the L-leveled RWLE FHE scheme (which is able to evaluate circuits up to depth L), the per-gate computation is $\tilde{O}(\lambda L^3)$ (Theorem 3).
- Bootstrapping "refreshes" a ciphertext by running the decryption function on it homomorphically, using an encrypted secret (given in the public key), resulting in reduced noise. The RWLE scheme has a decryption function that requires $\tilde{O}(\lambda)$ computation and depth $O(\log \lambda)$ (Section 5.2.1).
- Since bootstrapping is rather expensive, the best performance is achieved by bootstrapping lazily once every $\theta(\log \lambda)$ levels (Section 5.2.2).
- Bootstrapping can be batched by packing all ciphertexts together, decrypting, and unpacking all refreshed ciphertexts. For appropriately chosen parameters, batched bootstrapping requires the same computation and depth as non-batched bootstrapping (Theorem 4).
- Finally, bootstrapping permits short ciphertexts as input: E.g., each input can be encrypted using AES, to be de-compressed to longer ciphertexts that permit homomorphic operations[6] (Section 5.2 + paper).

To evaluate multiple iterations of the Turing machine by using the RLWE scheme we propose the following sequence:

1. The user initializes the tape of the Turing machine (the tape records its input). Next, the user encrypts the first n tape symbols using AES. Here, the n tape symbols can be represented

[6] A ciphertext $AES_K(x)$ can be encrypted using FHE.Enc with the FHE public key. As part of the public key the AES key K is encrypted using FHE.Enc. Evaluating AES decryption under encryption by using FHE.Add and FHE.Mult computes a ciphertext FHE.Enc(x).

by $\log |\Sigma|$ bits each, multiple symbols can be batched together into a single AES encryption. Finally, the user transmits the encrypted tape together with the Turing machine to the server. The user also asks (pays) the server to perform c iterations of the arithmetic network.

2. The server bootstraps the Turing machine by decrypting the tape homomorphically using the RLWE scheme. This gives the FHE encryption of each of the bits that represent the symbols on the input tape. The server executes a Boolean network that transforms this representation (under encryption) into the one discussed in the previous subsection. The server initializes the state and head position by setting the bit in s corresponding to the initial state to 1 and by setting $z_1 = 1$ and all other bits to 0.

3. The server performs $\theta(\log \lambda)$ iterations of the arithmetic network corresponding to the Turing machine. Since the arithmetic network has constant depth, a $L = \theta(\log \lambda)$-leveled RWLE scheme suffices. The per-gate computation is $\tilde{O}(\lambda L^3)$. Let $m = n + |K| + |\Sigma|$ be the number of bits over which the arithmetic network computes. Then a single iteration costs $\tilde{O}(m\lambda L^3) = \tilde{O}(m\lambda)$ computation. A single iteration of the Turing machine itself costs about $\log m$ computation, hence, the per-iteration computation overhead is $\tilde{O}(m\lambda/\log m) = \tilde{O}(m\lambda)$.

4. The server packs all ciphertexts, decrypts them homomorphically (bootstrapping), and unpacks them. Since the server uses a $L = \theta(\log \lambda)$-leveled RWLE scheme and bootstrapping requires $O(\log \lambda)$ levels, this can all be done under encryption. The batched bootstrapping costs $\tilde{O}(\lambda)$ computation if it is not done under encryption. Since it is performed under encryption, the cost is multiplied by the per-gate computation $\tilde{O}(\lambda L^3)$. Amortized over $\theta(\log \lambda)$ iterations, this gives an additional cost of $\tilde{O}(\lambda(\lambda L^3)/\log \lambda) = \tilde{O}(\lambda^2)$ per iteration.

5. The server repeats steps 3 and 4 till the total number of iterations equals c. Once c iterations have been completed the server transmits the encrypted tape, state and head position to the user (the server may first compress their representation to $m \log m$ bits).

6. The user decrypts the state. If $s = h$, the user decrypts the part of the tape which represents the output of the computation. If $s \neq h$, the user decrypts the head position. If $z = 0$, then the Turing machine was not well defined. If $z \in \{1, \ldots, n\}$, then the user may ask (pay) the server in an additional interaction to perform another number of iterations (as discussed before, this strategy leaks privacy).

   If $z = n + 1$, then the head dropped off the tape. In order to continue the computation, the user needs to add more tape. For example, he may redefine the arithmetic network to work over the tape squares that correspond to positions in $[n/2, 3n/2]$. This policy leads to an amortized per-iteration computation overhead of $\tilde{O}((\lambda + m)\lambda)$ (but again, this strategy leaks privacy because of the feedback to the server of variable $z = n + 1$).

   A second strategy is to allow an infinite length tape such that the head never drops of the tape. Since the head can only move as many steps as the number of iterations performed so far (in particular, $n \leq c$), this policy leads to an amortized per-iteration computation overhead of $\tilde{O}((\lambda + |K| + |\Sigma|)/c + c)\lambda)$. This overhead is about λ times the cost of evaluating the Turing machine itself (without encryption).

We conclude that evaluating a Turing machine under encryption by this procedure costs a factor $\tilde{O}((\lambda + n + |K| + |\Sigma|)\lambda)$ more computation as compared to a plain Turing machine computation. If we forget log terms, then this factor seems asymptotically close to optimal since the server needs to perform a computation over all $O(n + |K| + |\Sigma|)$ variables during each iteration, otherwise, information about the state of the Turing machine is revealed.