Static Analysis with Set-closure in Secrecy

Woosuk Lee, Hyunsook Hong, Kwangkeun Yi, and Jung Hee Cheon

Seoul National University

Abstract. We report that the homomorphic encryption scheme can unleash the possibility of static analysis of encrypted programs. Static analysis in cipher-world is desirable in the static-analysis-as-a-service setting, because it allows the program owners to encrypt and upload their programs to the static analysis service while the service provider can still analyze the encrypted programs without decrypting them. Only the owner of the decryption key (the program owner) is able to decrypt the analysis result. As a concrete example, we describe how to perform a pointer analysis in secrecy. In our method, a somewhat homomorphic encryption scheme of depth $O(\log m)$ is able to evaluate a simple pointer analysis with $O(\log m)$ homomorphic matrix multiplications, for the number m of pointer variables when the maximal pointer level is bounded. We also demonstrate the viability of our method by implementing the pointer analysis in secrecy.

1 Introduction

In order for a *static-analysis-as-a-service* system[1] to be popular, we need to solve the users' copy-right concerns. Users are reluctant to upload their source to analysis server.

For more widespread use of such service, we explored a method of performing static analysis on encrypted programs. Fig. 1 depicts the system.

Challenge Our work is based on *homomorphic encryption* (HE). A HE scheme enables computation of arbitrary functions on encrypted data. In other words, a HE scheme provides the functions $f_\oplus$ and $f_\wedge$ that satisfy the following homomorphic properties for plaintexts $x, y \in \{0, 1\}$ without any secrets:

$$\mathsf{Enc}(x \oplus y) = f_\oplus(\mathsf{Enc}(x), \mathsf{Enc}(y)), \qquad \mathsf{Enc}(x \wedge y) = f_\wedge(\mathsf{Enc}(x), \mathsf{Enc}(y))$$

A HE scheme was first shown in the work of Gentry [14]. Since then, although there have been many efforts to improve the efficiency [3, 4, 9, 21], the cost is still too large for immediate applications into daily computations.

Due to the high complexity of HE operation, practical deployments of HE require *application-specific* techniques. Application-specific techniques are often demonstrated in other fields. Kim et al. [8] introduced an optimization technique to reduce the depth of an arithmetic circuit computing edit distance on encrypted DNA sequences. In addition, methods of bubble sort and insertion sort on encrypted data have been proposed [6]. Also, private database query protocol using somewhat homomorphic encryption has been proposed [2].

Our Results As a first step, we propose a pointer analysis in secrecy. As many analyses depends on the pointer information, we expect our work to have significant implications along the way to static analysis in secrecy.

We first describe a basic approach. We design an arithmetic circuit of the pointer analysis algorithm only using operations that a HE scheme supports. Program owner encrypts some numbers representing his program under the HE scheme. On the encrypted data, a server performs a series of corresponding homomorphic operations referring to the arithmetic circuit and outputs encrypted pointer analysis results. This basic approach is simple but very costly.

To decrease the cost of the basic approach, we apply two optimization techniques. One is to exploit the *ciphertext packing* technique not only for performance boost but also for decreasing the huge number of ciphertexts required for the basic scheme. The basic approach makes ciphertexts size grow by the square to the number of pointer variables in a program, which is far from practical. Ciphertext packing makes total ciphertexts size be linear to the number of variables. The other technique is *level-by-level analysis*. We analyze the pointers of the same level together from the highest to lowest. With this technique, the depth of the arithmetic circuit for the pointer analysis significantly decreases: from $O(m^2 \log m)$ to $O(n \log m)$ for the number m of pointer variables and the maximal pointer level n. By decreasing the depth, which is the most important in performance of HE schemes, the technique decreases both ciphertexts size and the cost of each homomorphic operation.

The improvement by the two optimizations is summarized in Table 1.

	Multiplicative depth	# Ctxt
Basic	$O(m^2 \log m)$	$4m^2$
Improved	$O(n \log m)$	$(2n+2)m$

m : the number of pointer variables in the target program

n : the maximum level of pointer in the program, which does not exceed 5 in usual

Table 1: The comparison between the basic and the improved scheme

Although our interest in this paper is limited to a pointer analysis, we expect other analyses in the same family will be performed in a similar manner to our method. Analyses in the family essentially compute a transitive closure of a graph subject to dynamic changes; new edges may be added during the analysis. Our method computes an encrypted transitive closure of a graph when both edge insertion queries and all the edges are encrypted. Thus, we expect only a few modifications to our method will make other similar analyses (*e.g.,* 0-CFA) be in secrecy.

2 Background

In this section, we introduce the concept of homomorphic encryption, and describe the security model of our static analysis in secrecy.

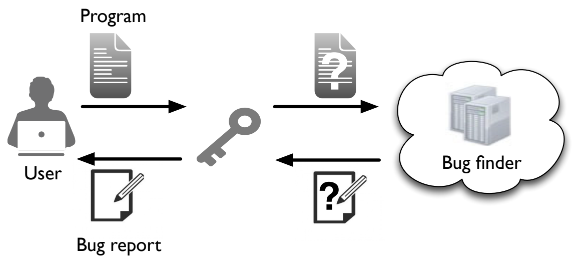

Fig. 1: Secure static analysis is performed in 3 steps: 1) target program encryption 2) analysis in secrecy, and 3) analysis result decryption

2.1 Homomorphic Encryption

A homomorphic encryption (HE) scheme HE=(KG, Enc, Dec, Eval) is a quadruple of probabilistic polynomial-time algorithm as follows:

- $(\mathsf{pk}, \mathsf{evk}; \mathsf{sk}) \leftarrow \mathsf{HE.KG}(1^\lambda)$: The algorithm takes the security parameter λ as input and outputs a public encryption key pk, a public evaluation key evk, and a secret decryption key sk.
- $\bar{c} \leftarrow \mathsf{HE.Enc}_{\mathsf{pk}}(\mu, r)$: The algorithm takes the public key pk, a single bit message $\mu \in \{0, 1\}$,[1] and a randomizer r. It outputs a ciphertext $\bar{c}$. If we have no confusion, we omit the randomizer r.
- $\mu \leftarrow \mathsf{HE.Dec}_{\mathsf{sk}}(\bar{c})$: The algorithm takes the secret key sk and a ciphertext $\bar{c} = \mathsf{HE.Enc}_{\mathsf{pk}}(\mu)$ and outputs a message $\mu \in \{0, 1\}$
- $\bar{c}_f \leftarrow \mathsf{HE.Eval}_{\mathsf{evk}}(f; \bar{c}_1, \ldots, \bar{c}_l)$: The algorithm takes the evaluation key evk, a function $f : \{0, 1\}^l \to \{0, 1\}$ represented by an arithmetic circuit over $\mathbb{Z}_2 = \{0, 1\}$ with the addition and multiplication gates, and a set of l ciphertexts $\{\bar{c}_i = \mathsf{HE.Enc}(\mu_i)\}_{i=1}^{l}$, and outputs a ciphertext $\bar{c}_f = \mathsf{HE.Enc}(f(\mu_1, \cdots, \mu_l))$.

We say that a scheme HE=(KG, Enc, Dec, Eval) is *f-homomorphic* if for any set of inputs $(\mu_1, \cdots, \mu_l)$, and all sufficiently large λ, it holds that

$$\Pr\left[\mathsf{HE.Dec}_{\mathsf{sk}}\left(\mathsf{HE.Eval}_{\mathsf{evk}}(f; \bar{c}_1, \cdots, \bar{c}_l)\right) \neq f(\mu_1, \cdots, \mu_l)\right] = \mathrm{negl}(\lambda),$$

where negl is a negligible function, $(\mathsf{pk}, \mathsf{evk}; \mathsf{sk}) \leftarrow \mathsf{HE.KG}(1^\lambda)$, and $\bar{c}_i \leftarrow \mathsf{HE.Enc}_{\mathsf{pk}}(\mu_i)$.

If a HE scheme can evaluate all functions represented by arithmetic circuits over $\mathbb{Z}_2$ (equivalently, boolean circuits with AND and XOR gates[2]), the HE scheme is called *fully homomorphic*.

To facilitate understanding of HE schemes, we introduce a simple symmetric version of the HE scheme [11] based on approximate common divisor problems [19]:

[1] For simplicity, we assume that the plaintext space is $\mathbb{Z}_2 = \{0, 1\}$, but extension to larger plaintext space is immediate.

[2] AND and XOR gates are sufficient to simulate all binary circuits.

– $\mathsf{sk} \leftarrow \mathsf{KG}(1^\lambda)$: Choose an integer p and outputs the secret key $\mathsf{sk} = p$.
– $\bar{c} \leftarrow \mathsf{Enc}(\mu \in \{0,1\})$: Choose a random integer q and a random noise integer r with $|r| \ll |p|$. It outputs $\bar{c} = pq + 2r + \mu$.
– $\mu \leftarrow \mathsf{Dec}_{\mathsf{sk}}(\bar{c})$: Outputs $\mu = ((\bar{c} \bmod p) \bmod 2)$.
– $\bar{c}_{\mathsf{add}} \leftarrow \mathsf{Add}(\bar{c}_1, \bar{c}_2)$: Outputs $\bar{c}_{\mathsf{add}} = \bar{c}_1 + \bar{c}_2$.
– $\bar{c}_{\mathsf{mult}} \leftarrow \mathsf{Mult}(\bar{c}_1, \bar{c}_2)$: Outputs $\bar{c}_{\mathsf{mult}} = \bar{c}_1 \times \bar{c}_2$.

For ciphertexts $\bar{c}_1 \leftarrow \mathsf{Enc}(\mu_1)$ and $\bar{c}_2 \leftarrow \mathsf{Enc}(\mu_2)$, we know each $\bar{c}_i$ is of the form $\bar{c}_i = pq_i + 2r_i + \mu_i$ for some integer q_i and noise r_i. Hence $((\bar{c}_i \bmod p) \bmod 2) = \mu_i$, if $|2r_i + \mu_i| < p/2$. Then, the following equations hold:

$$\bar{c}_1 + \bar{c}_2 = p(q_1 + q_2) + \underbrace{2(r_1 + r_2) + \mu_1 + \mu_2}_{\text{noise}},$$
$$\bar{c}_1 \times \bar{c}_2 = p(pq_1q_2 + \cdots) + \underbrace{2(2r_1r_2 + r_1\mu_2 + r_2\mu_1) + \mu_1 \cdot \mu_2}_{\text{noise}}$$

Based on these properties,

$$\mathsf{Dec}_{\mathsf{sk}}(\bar{c}_1 + \bar{c}_2) = \mu_1 + \mu_2 \text{ and } \mathsf{Dec}_{\mathsf{sk}}(\bar{c}_1 \times \bar{c}_2) = \mu_1 \cdot \mu_2$$

if the absolute value of $2(2r_1r_2 + r_1\mu_2 + r_2\mu_1) + \mu_1\mu_2$ is less than $p/2$. The noise in the resulting ciphertext increases during homomorphic addition and multiplication (twice and quadratically as much noise as before respectively). If the noise becomes larger than $p/2$, the decryption result of the above scheme will be spoiled. As long as the noise is managed, the scheme is able to potentially evaluate all boolean circuits as the addition and multiplication in $\mathbb{Z}_2$ corresponds to the XOR and AND operations.

We consider somewhat homomorphic encryption (SWHE) schemes that adopt the modulus-switching [4, 5, 10, 15] for the noise-management. The modulus-switching reduces the noise by scaling the factor of the modulus in the ciphertext space. SWHE schemes support a limited number of homomorphic operations on each ciphertext, as opposed to fully homomorphic encryption schemes [7, 11, 14, 23] which are based on a different noise-management technique. But SWHE schemes are more efficient to support low-degree homomorphic computations.

In this paper, we will measure the efficiency of homomorphic evaluation by the *multiplicative depth* of an underlying circuit. The multiplicative depth is defined as the number of multiplication gates encountered along the longest path from input to output. When it comes to the depth of a circuit computing a function f, we discuss the circuit of the minimal depth among any circuits computing f. For example, if a somewhat homomorphic encryption scheme can evaluate circuits of depth L, we may maximally perform 2^L multiplications on the ciphertexts maintaining the correctness of the result. We do not consider the number of addition gates in counting the depth of a circuit because the noise increase by additions is negligible compared with the noise increase by multiplications. The multiplicative depth of a circuit is the most important factor in the performance of homomorphic evaluation of the circuit in the view of both the size of ciphertexts and the cost of per-gate homomorphic computation. Thus, minimizing the depth is the most important in performance.

2.2 The BGV-type cryptosystem

Our underlying HE scheme is a variant of the Brakerski-Gentry-Vaikuntanathan (BGV)-type cryptosystem [4, 15]. In this section, we only provide a brief review of the cryptosystem [4]. For more details, please refer to [4, 15]. Let $\Phi(X)$ be an irreducible polynomial over $\mathbb{Z}$. The implementation of the scheme is based on the polynomial operations in ring $R = \mathbb{Z}[X]/(\Phi(X))$ which is the set of integer polynomials of degree less than $\deg(\Phi)$. Let $R_p := R/pR$ be the message space for a prime p and $R_q \times R_q$ be the ciphertext space for an integer q. Now, we describe the BGV cryptosystem as follows:

- $((a, b); \mathbf{s}) \leftarrow \mathsf{BGV.KG}(1^\lambda, \sigma, q)$: Choose a secret key $\mathbf{s}$ and a noise polynomial e from a discrete Gaussian distribution over R with standard deviation σ. Choose a random polynomial a from R_q and generate the public key $(a, b = a \cdot \mathbf{s} + p \cdot e) \in R_q \times R_q$. Output the public key $\mathsf{pk} = (a, b)$ and the secret key $\mathsf{sk} = \mathbf{s}$.
- $\bar{\mathbf{c}} \leftarrow \mathsf{BGV.Enc}_{\mathsf{pk}}(\mu)$: To encrypt a message $\mu \in R_p$, choose a random polynomial v whose coefficients are in $\{0, \pm 1\}$ and two noise polynomials e_0, e_1. Output the ciphertext $\mathbf{c} = (c_0, c_1) = (bv + pe_0 + \mu, av + pe_1) \bmod (q, \Phi(X))$.
- $\mu \leftarrow \mathsf{BGV.Dec}_{\mathsf{sk}}(\bar{\mathbf{c}})$: Given a ciphertext $\bar{\mathbf{c}} = (c_0, c_1)$, it outputs $\mu = (((c_0 - c_1 \cdot \mathbf{s}) \bmod q) \bmod p)$.
- $\bar{\mathbf{c}}_{\mathsf{add}} \leftarrow \mathsf{BGV.Add}_{\mathsf{pk}}(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2; \mathsf{evk})$: Given ciphertexts $\bar{\mathbf{c}}_1 = \mathsf{BGV.Enc}(\mu_1)$ and $\bar{\mathbf{c}}_2 = \mathsf{BGV.Enc}(\mu_2)$, it outputs the ciphertext $\bar{\mathbf{c}}_{\mathsf{add}} = \mathsf{BGV.Enc}(\mu_1 + \mu_2)$.
- $\bar{\mathbf{c}}_{\mathsf{mult}} \leftarrow \mathsf{BGV.Mult}_{\mathsf{pk}}(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2; \mathsf{evk})$: Given ciphertexts $\bar{\mathbf{c}}_1 = \mathsf{BGV.Enc}(\mu_1)$ and $\bar{\mathbf{c}}_2 = \mathsf{BGV.Enc}(\mu_2)$, it outputs the ciphertext $\bar{\mathbf{c}}_{\mathsf{mult}} = \mathsf{BGV.Enc}(\mu_1 \cdot \mu_2)$.

2.3 Security Model

We assume that program owners and analyzer servers are semi-honest. In this model, the analyzer runs the protocol exactly as specified, but may try to learn as much as possible about the program information. However, in our method, since programs are encrypted under the BGV-type cryptosystem which is secure under the hardness of the ring learning with errors (RLWE) problem (see [4] for the details), analyzers cannot learn no more information than the program size.

3 A Basic Construction of a Pointer Analysis in Secrecy

In this section, we explain how to perform a pointer analysis in secrecy.

3.1 A Brief Review of a Pointer Analysis

We begin with a brief review of a pointer analysis. We consider flow- and context-insensitive pointer analyses. To simplify our presentation, we consider a tiny language consisting of primitive assignments involving just the operations `*` and `&`. A program P is a finite set of assignments `A`:

$$\mathtt{A} \rightarrow \mathtt{x} = \&\mathtt{y} \mid \mathtt{x} = \mathtt{y} \mid *\mathtt{x} = \mathtt{y} \mid \mathtt{x} = *\mathtt{y}$$

We present a pointer analysis algorithm with simple resolution rules in a similar manner to [18]. Given some program P, we construct resolution rules as specified in Table 2. In the first rule, the side condition "if $\texttt{x} = \&\texttt{y}$ in P" indicates that there is an instance of this rule for each occurrence of an assignment of the form $\texttt{x} = \&\texttt{y}$ in P. The side conditions in the other rules are similarly interpreted. Intuitively, an edge $\texttt{x} \longrightarrow \&\texttt{y}$ indicates that $\texttt{x}$ can point to $\texttt{y}$. An edge $\texttt{x} \longrightarrow \texttt{y}$ indicates that for any variable $\texttt{v}$, if $\texttt{y}$ may point to $\texttt{v}$ then $\texttt{x}$ may point to $\texttt{v}$. The pointer analysis is applying the resolution rules until reaching a fixpoint.

$$\frac{\quad}{\texttt{x} \longrightarrow \&\texttt{y}} \ (\text{if } \texttt{x} = \&\texttt{y} \text{ in } P) \quad (\mathsf{New}) \qquad\qquad \frac{\quad}{\texttt{x} \longrightarrow \texttt{y}} \ (\text{if } \texttt{x} = \texttt{y} \text{ in } P) \quad (\mathsf{Copy})$$

$$\frac{\texttt{x} \longrightarrow \&\texttt{z}}{\texttt{y} \longrightarrow \texttt{z}} \ (\text{if } \texttt{y} = *\texttt{x} \text{ in } P) \quad (\mathsf{Load}) \qquad\qquad \frac{\texttt{x} \longrightarrow \&\texttt{z}}{\texttt{z} \longrightarrow \texttt{y}} \ (\text{if } *\texttt{x} = \texttt{y} \text{ in } P) \quad (\mathsf{Store})$$

$$\frac{\texttt{x} \longrightarrow \texttt{z} \quad \texttt{z} \longrightarrow \&\texttt{y}}{\texttt{x} \longrightarrow \&\texttt{y}} \quad (\mathsf{Trans})$$

Table 2: Resolution rules for pointer analysis.

3.2 The Pointer Analysis in Secrecy

The analysis in secrecy will be performed in the following 3 steps. First, a program owner derives numbers that represent his program and encrypt them under a HE scheme. The encrypted numbers will be given to an analysis server. Next, the server performs homomorphic evaluation of an underlying arithmetic circuit representing the pointer analysis with the inputs from the program owner. Finally, the program owner obtains an encrypted analysis result and recovers a set of points-to relations by decryption.

Before beginning, we define some notations. We assume a program owner assigns a number to every variable using some numbering scheme. In the rest of the paper, we will denote a variable numbered i by $\texttt{x}_\texttt{i}$. In addition, to express the arithmetic circuit of the pointer analysis algorithm, we define the notations $\delta_{i,j}$ and $\eta_{i,j}$ in $\mathbb{Z}$ for $i, j = 1, \cdots, m$ by

$$\begin{array}{lll} \delta_{i,j} \neq 0 & \text{iff} & \text{An edge } \texttt{x}_\texttt{i} \longrightarrow \&\texttt{x}_\texttt{j} \text{ is derived by the resolution rules.} \\ \eta_{i,j} \neq 0 & \text{iff} & \text{An edge } \texttt{x}_\texttt{i} \longrightarrow \texttt{x}_\texttt{j} \text{ is derived by the resolution rules.} \end{array}$$

for variables $\texttt{x}_\texttt{i}$ and $\texttt{x}_\texttt{j}$, and the number m of pointer variables.

Inputs from Client A client (program owner) derives the following numbers that represent his program P (here, m is the number of variables):

$$\{(\delta_{i,j}, \eta_{i,j}, u_{i,j}, v_{i,j}) \in \mathbb{Z} \times \mathbb{Z} \times \{0,1\} \times \{0,1\} \mid 1 \leq i, j \leq m\}$$

which are initially assigned as follows:

$$\delta_{i,j} \leftarrow \begin{cases} 1 & \text{if } \exists \mathtt{x_i} = \&\mathtt{x_j} \\ 0 & \text{otherwise} \end{cases} \qquad \eta_{i,j} \leftarrow \begin{cases} 1 & \text{if } \exists \mathtt{x_i} = \mathtt{x_j} \text{ or } i = j \\ 0 & \text{otherwise} \end{cases}$$

$$u_{i,j} \leftarrow \begin{cases} 1 & \text{if } \exists \mathtt{x_j} = *\mathtt{x_i} \\ 0 & \text{otherwise} \end{cases} \qquad v_{i,j} \leftarrow \begin{cases} 1 & \text{if } \exists *\mathtt{x_j} = \mathtt{x_i} \\ 0 & \text{otherwise} \end{cases}$$

In the assignment of $\delta_{i,j}$, the side condition $\exists \mathtt{x_i} = \&\mathtt{x_j}$ indicates that there is the assignment $\mathtt{x_i} = \&\mathtt{x_j}$ in the program P. The other side conditions are similarly interpreted.

The program owner encrypts the numbers using a HE scheme and provides them to the server. We denote the encryption of $\delta_{i,j}$, $\eta_{i,j}$, $u_{i,j}$, and $v_{i,j}$ by $\bar{\delta}_{i,j}$, $\bar{\eta}_{i,j}$, $\bar{u}_{i,j}$, and $\bar{v}_{i,j}$, respectively. Therefore, the program owner generates $4m^2$ ciphertexts where m is the number of pointer variables.

Server's Analysis Provided the set of the ciphertexts from the program owner, the server homomorphically applies the resolution rules. With a slight abuse of notation, we will denote $+$ and $\cdot$ as homomorphic addition and multiplication respectively to simplify the presentation.

We begin with applying the Trans rule in Table 2. For $i, j = 1, \cdots, m$, the server updates $\bar{\delta}_{i,j}$ as follows:

$$\bar{\delta}_{i,j} \leftarrow \textstyle\sum_{k=1}^{m} \bar{\eta}_{i,k} \cdot \bar{\delta}_{k,j}$$

If edges $\mathtt{x_i} \longrightarrow \mathtt{x_k}$ and $\mathtt{x_k} \longrightarrow \&\mathtt{x_j}$ are derived by the resolution rules for some variable $\mathtt{x_k}$, then the edge $\mathtt{x_i} \longrightarrow \&\mathtt{x_j}$ will be derived by the Trans rule and the value $\delta_{i,j}$ will have a positive integer. If there is no variable $\mathtt{x_k}$ that satisfies the conditions for all $k = 1, \cdots, m$, there will be no update on $\delta_{i,j}$ ($\because \eta_{i,i} = 1$).

Next, we describe applying the Load rule.

$$\bar{\eta}_{i,j} \leftarrow \bar{\eta}_{i,j} + \textstyle\sum_{k=1}^{m} \bar{u}_{i,k} \cdot \bar{\delta}_{k,j}$$

If an edge $\mathtt{x_k} \longrightarrow \&\mathtt{x_j}$ is derived and the program P has a command $\mathtt{x_i} := *\mathtt{x_k}$ and for some integer k, then the edge $\mathtt{x_i} \longrightarrow \mathtt{x_j}$ will be derived and $\eta_{i,j}$ will have a positive value. If none of variables $\mathtt{x_k}$ satisfies the conditions, there will be no update on $\eta_{i,j}$.

Finally, to apply the Store rule, the server performs the following operations:

$$\bar{\eta}_{i,j} \leftarrow \bar{\eta}_{i,j} + \textstyle\sum_{k=1}^{m} \bar{v}_{j,k} \cdot \bar{\delta}_{k,i}$$

If an edge $\mathtt{x_k} \longrightarrow \&\mathtt{x_i}$ is derived and the program P has a command $*\mathtt{x_k} := \mathtt{x_j}$ for some variable $\mathtt{x_k}$, then an edge $\mathtt{x_i} \longrightarrow \mathtt{x_j}$ will be derived and $\eta_{i,j}$ will have a non-zero value.

Note that the server must repeat applying the rules as if in the worst case since the server cannot know whether a fixpoint is reached during the operations. The server may obtain a fixpoint by repeating the following two steps in turn m^2 times:

1. Applying the Trans rule m times
2. Applying the Load and Store rules

The reason for doing step 1 is that we may have a m-length path through edges as the longest one in the worst case. The reason for repeating the two steps m^2 times is that we may have a new edge by applying the Load and Store rules, and we may have at most m^2 edges at termination of the analysis.

We need $O(m^2 \log m)$ multiplicative depth in total. Because performing the step 1 entails m homomorphic multiplications on each $\bar{\delta}_{i,j}$, and repeating the two steps m^2 times performs about m^{m^2} homomorphic multiplications on each $\bar{\delta}_{i,j}$.

Output Determination The client receives the updated $\{\bar{\delta}_{i,j} \mid 1 \leq i, j \leq m\}$ from the server and recovers a set of points-to relations as follows:

$$\{\mathtt{x_i} \longrightarrow \&\mathtt{x_j} \mid \mathsf{HE.Dec_{sk}}(\bar{\delta}_{i,j}) \neq 0 \text{ and } 1 \leq i, j \leq m\}$$

Why do we not represent the algorithm by a Boolean circuit? One may wonder why we represent the pointer analysis algorithm by an arithmetic circuit rather than a Boolean circuit. As an example of applying the Trans rule, we might update $\delta_{i,j}$ by $\delta_{i,j} \leftarrow \bigvee_{1 \leq k \leq m} \eta_{i,k} \wedge \delta_{k,j}$. However, this representation causes more multiplicative depth than our current approach. The OR operation consists of the XOR and AND operations as follows: $x \vee y \overset{\mathsf{def}}{=} (x \wedge y) \oplus x \oplus y$. Note that the addition and multiplication in $\mathbb{Z}_2$ correspond to the XOR and AND operations, respectively. Since the OR operation requires a single multiplication over ciphertexts, this method requires m more multiplications than our current method to update $\delta_{i,j}$ once.

sectionImprovement of the Pointer Analysis in Secrecy In this section, we present three techniques to reduce the cost of the basic approach described in the section 3.2. We begin with problems of the basic approach followed by our solutions.

3.3 Problems of the Basic Approach

The basic scheme has the following problems that make the scheme impractical.

- Huge # of homomorphic multiplications: The scheme described in the section 3.2 can be implemented with a SWHE scheme of the depth $O(m^2 \log m)$. Homomorphic evaluation of a circuit over the hundreds depth is regarded unrealistic in usual. The depth of the arithmetic circuit described in the section 3.2 exceeds 300 even if a program has only 10 variables.
- Huge # of ciphertexts: The basic approach requires $4m^2$ ciphertexts, where m is the number of pointer variables. When a program has 1000 variables, 4 million ciphertexts are necessary. For instance, the size of a single ciphertext in the BGV cryptosystem is about 2MB when the depth is 20. In this case, the scheme requires 7.6 TB memory space for all the ciphertexts.

– Decryption error may happen: In our underlying HE scheme, the message space is the polynomial ring over modulus p. During the operations, $\delta_{i,j}$ and $\eta_{i,j}$ increase and may become p which is congruent to 0 modulo p. Since we are interested in whether each value is zero or not, incorrect results may be derived if the values become congruent to 0 modulo p by accident.

3.4 Overview of Improvement

For the number m of pointer variables and the maximal pointer level n, the followings are our solutions.

– **Level-by-level Analysis**: We analyze pointers of the same level together from the highest to lowest in order to decrease the depth of the arithmetic circuit described in the section 3.2. To apply the technique, program owners are required to reveal an upper bound of the maximal pointer level. By this compromise, the depth of the arithmetic circuit significantly decreases: from $O(m^2 \log m)$ to $O(n \log m)$. We expect this information leak is not much compromise because the maximal pointer level is well known to be a small number in usual cases.
– **Ciphertext Packing**: We adopt ciphertext packing not only for performance boost but also for decreasing the huge number of ciphertexts required for the basic scheme. The technique makes total ciphertext sizes be linear to the number of variables.
– **Randomization of Ciphertexts**: We randomize ciphertexts to balance the probability of incorrect results and ciphertext size. We may obtain correct results with the probability of $(1 - \frac{1}{p-1})^{n(\lceil \log m \rceil + 3)}$.

The following table summarizes the improvement.

	Depth	# Ctxt
Basic	$O(m^2 \log m)$	$4m^2$
Improved	$O(n \log m)$	$(2n+2)m$

3.5 Level-by-level Analysis

We significantly decrease the multiplicative depth by doing the analysis in a level by level manner in terms of *level of pointers*. The level of a pointer is the maximum level of possible indirect accesses from the pointer, *e.g.,* the pointer level of `p` in the definition "`int** p`" is 2. From this point, we denote the level of a pointer variable `x` by ptl(`x`).

We assume that type-casting a pointer value to a lower or higher-level pointer is absent in programs. For example, we do not consider a program that has type-casting from `void*` to `int**` because the pointer level increases from 1 to 2.

On the assumption, we analyze the pointers of the same level together from the highest to lowest. The correctness is guaranteed because lower-level pointers cannot affect pointer values of higher-level pointers during the analysis. For example, pointer values of x initialized by assignments of the form $\mathtt{x} = \&\mathtt{y}$ may change by assignments of the form $\mathtt{x} = \mathtt{y}$, $\mathtt{x} = *\mathtt{y}$, or $*\mathtt{p} = \mathtt{y}$ ($\because$ `p` may point to `x`) during the analysis. The following table presents pointer levels of involved variables in the assignments that affects pointer values of `x`.

Assignment	Levels
$\mathtt{x} = \mathtt{y}$	$\mathsf{ptl}(\mathtt{x}) = \mathsf{ptl}(\mathtt{y})$
$\mathtt{x} = *\mathtt{y}$	$\mathsf{ptl}(\mathtt{y}) = \mathsf{ptl}(\mathtt{x}) + 1$
$*\mathtt{p} = \mathtt{y}$	$\mathsf{ptl}(\mathtt{p}) = \mathsf{ptl}(\mathtt{x}) + 1 \land \mathsf{ptl}(\mathtt{y}) = \mathsf{ptl}(\mathtt{x})$

Note that all the variables affect pointer values of `x` have higher or equal pointer level compared to `x`.

Now we describe the level-by-level analysis in secrecy similarly to the basic scheme. Before beginning, we define the notations $\delta_{i,j}^{(\ell)}$ and $\eta_{i,j}^{(\ell)}$ in $\mathbb{Z}$ for $i, j = 1, \cdots, m$ by

$$\begin{aligned} \delta_{i,j}^{(\ell)} \neq 0 \quad &\text{iff} \quad \text{An edge } \mathtt{x_i} \longrightarrow \&\mathtt{x_j} \text{ is derived and } \mathsf{ptl}(\mathtt{x_i}) = \ell \\ \eta_{i,j}^{(\ell)} \neq 0 \quad &\text{iff} \quad \text{An edge } \mathtt{x_i} \longrightarrow \mathtt{x_j} \text{ is derived and } \mathsf{ptl}(\mathtt{x_i}) = \ell. \end{aligned}$$

Inputs from Client For the level-by-level analysis, a program owner derives the following numbers that represent his program P (n is the maximal level of pointer in the program):

$$\{(\delta_{i,j}^{(\ell)}, \eta_{i,j}^{(\ell)}) \mid 1 \leq i, j \leq m, 1 \leq \ell \leq n\} \cup \{(u_{i,j}, v_{i,j}) \mid 1 \leq i, j \leq m\}$$

where $\delta_{i,j}^{(\ell)}$ and $\eta_{i,j}^{(\ell)}$ are defined as follows.

$$\delta_{i,j}^{(\ell)} = \begin{cases} 1 & \text{if } \exists \mathtt{x_i} = \&\mathtt{x_j},\ \mathsf{ptl}(\mathtt{x_i}) = \ell \\ 0 & \text{o.w.} \end{cases} \qquad \eta_{i,j}^{(\ell)} = \begin{cases} 1 & \text{if } (\exists \mathtt{x_i} = \mathtt{x_j} \text{ or } i = j),\ \mathsf{ptl}(\mathtt{x_i}) = \ell \\ 0 & \text{o.w.} \end{cases}$$

The definitions of $u_{i,j}$ and $v_{i,j}$ are the same as in the section 3.2. We denote the encryption of $\delta_{i,j}^{(\ell)}$ and $\eta_{i,j}^{(\ell)}$ by $\bar{\delta}_{i,j}^{(\ell)}$, $\bar{\eta}_{i,j}^{(\ell)}$, respectively.

Server's Analysis Server's analysis begins with propagating pointer values of the maximal level n by applying the Trans rule as much as possible. In other words, for $i, j = 1, \cdots, m$, the server repeats the following update m times:

$$\bar{\delta}_{i,j}^{(n)} \leftarrow \textstyle\sum_{k=1}^{m} \bar{\eta}_{i,k}^{(n)} \cdot \bar{\delta}_{k,j}^{(n)}$$

Next, from the level $n-1$ down to 1, the analysis at a level ℓ is carried out in the following steps:

1. applying the Load rule: $\bar{\eta}_{i,j}^{(\ell)} \leftarrow \bar{\eta}_{i,j}^{(\ell)} + \sum_{k=1}^{m} \bar{u}_{i,k} \cdot \bar{\delta}_{k,j}^{(\ell+1)}$
2. applying the Store rule: $\bar{\eta}_{i,j}^{(\ell)} \leftarrow \bar{\eta}_{i,j}^{(\ell)} + \sum_{k=1}^{m} \bar{v}_{j,k} \cdot \bar{\delta}_{k,i}^{(\ell+1)}$

3. applying the Trans rule: repeating the following update m times

$$\bar{\delta}_{i,j}^{(\ell)} \leftarrow \textstyle\sum_{k=1}^{m} \bar{\eta}_{i,k}^{(\ell)} \cdot \bar{\delta}_{k,j}^{(\ell)}$$

Through step 1 and 2, edges of the form $x_i \longrightarrow x_j$ are derived where either x_i or x_j is determined by pointer values of the immediate higher level $\ell+1$. In step 3, pointer values of a current level ℓ are propagated as much as possible.

We need $O(n \log m)$ multiplicative depth in total because repeating the above 3 steps n times entails maximally m^n homomorphic multiplications on a single ciphertext.

Output Determination The client receives the updated $\{\bar{\delta}_{i,j}^{(\ell)} \mid 1 \le i,j \le m, 1 \le \ell \le n\}$ from the server and recovers a set of points-to relations as follows:

$$\{\mathtt{x_i} \longrightarrow \&\mathtt{x_j} \mid \mathsf{HE.Dec_{sk}}(\bar{\delta}_{i,j}^{(\ell)}) \neq 0,\ 1 \le i,j \le m,\ \text{and}\ 1 \le \ell \le n\}$$

3.6 Ciphertext Packing

Our use of ciphertext packing aims to decrease total ciphertext size by using fewer ciphertexts than the basic scheme. Thanks to ciphertext packing, a single ciphertext can hold multiple plaintexts rather than a single value. For given a vector of plaintexts $(\mu_1, \cdots, \mu_m)$, the BGV cryptosystem allows to obtain a ciphertext $\bar{\mathbf{c}} \leftarrow \mathsf{BGV.Enc}(\mu_1, \cdots, \mu_m)$.

Furthermore, as each ciphertext holds a vector of multiple plaintexts, homomorphic operations between such ciphertexts are performed component-wise. For given ciphetexts $\bar{\mathbf{c}}_1 = \mathsf{BGV.Enc}(\mu_{1,1}, \cdots, \mu_{1,m})$ and $\bar{\mathbf{c}}_2 = \mathsf{BGV.Enc}(\mu_{2,1}, \cdots, \mu_{2,m})$, the homomorphic addition and multiplication in the BGV scheme satisfy the following properties:

$$\mathsf{BGV.Add}(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2) \text{ returns a ciphertext } \mathsf{BGV.Enc}(\mu_{1,1} + \mu_{2,1}, \cdots, \mu_{1,m} + \mu_{2,m})$$

$$\mathsf{BGV.Mult}(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2) \text{ returns a ciphertext } \mathsf{BGV.Enc}(\mu_{1,1} \cdot \mu_{2,1}, \cdots, \mu_{1,m} \cdot \mu_{2,m})$$

The BGV scheme provides other homomorphic operations such as cyclic rotation. For example, we can perform cyclic rotation of vector by any amount on ciphertexts (*e.g.,* $\mathsf{BGV.Enc}(\mu_m, \mu_1, \cdots, \mu_{m-1})$ from $\mathsf{BGV.Enc}(\mu_1, \mu_2, \cdots, \mu_m)$). Using the homomorphic addition, multiplication, and other operations, we can perform the matrix addition, multiplication and transposition operations on encrypted matrices.

In this subsection, we describe ciphertext packing and the homomorphic matrix operations in more detail.

Principle of Ciphertext Packing We begin with some notations. For an integer q, $\mathbb{Z}_q \overset{\mathsf{def}}{=} [-q/2, q/2) \cap \mathbb{Z}$ and $x \bmod q$ denotes a number in $[-q/2, q/2) \cap \mathbb{Z}$ which is equivalent to x modulo q. Recall that the message space of the BGV

Rule	Integer form	Matrix form
Trans	$\delta_{i,j}^{(\ell)} \leftarrow \sum_{k=1}^{m} \eta_{i,k}^{(\ell)} \cdot \delta_{k,j}^{(\ell)}$	$\Delta_\ell \leftarrow H_\ell \cdot \Delta_\ell$
Load	$\eta_{i,j}^{(\ell)} \leftarrow \eta_{i,j}^{(\ell)} + \sum_{k=1}^{m} u_{i,k} \cdot \delta_{k,j}^{(\ell+1)}$	$H_\ell \leftarrow H_\ell + U \cdot \Delta_{\ell+1}$
Store	$\eta_{i,j}^{(\ell)} \leftarrow \eta_{i,j}^{(\ell)} + \sum_{k=1}^{m} v_{j,k} \cdot \delta_{k,i}^{(\ell+1)}$	$H_\ell \leftarrow H_\ell + (V \cdot \Delta_{\ell+1})^T$

Table 3: Circuit expression of the level-by-level analysis

cryptosystem is $R_p = \mathbb{Z}[X]/(p, \Phi(X))$ for a prime p and an irreducible polynomial $\Phi(X)$. We identify the polynomial ring R_p with $\{a_0 + a_1X + \cdots + a_{\deg\Phi-1}X^{\deg\Phi-1} \mid a_i \in \mathbb{Z}_p \text{ and } 0 \le i < \deg\Phi\}$.

In the basic approach, although the message space of the BGV scheme is the polynomial ring R_p, we have used only constant polynomials (*i.e.,* numbers) for plaintexts. Thus, if a vector of plaintexts is represented as a single non-constant polynomial, a single ciphertext can hold multiple plaintexts rather than a single value. Therefore we can save the total memory space by using fewer ciphertexts than the basic scheme. Suppose the factorization of $\Phi(X)$ modulo p is $\Phi(X) = \prod_{i=1}^{m} F_i(X) \bmod p$ where each F_i is an irreducible polynomial in $\mathbb{Z}_p[X]$. Then a polynomial $\mu(X) \in R_p$ can be viewed as a vector of m different small polynomials, $(\mu_1(X), \cdots, \mu_m(X))$ such that $\mu_i(X) = (\mu(X) \text{ modulo } F_i(X))$ for $i = 1, \cdots, m$.

From this observation, we can encrypt a vector $\boldsymbol{\mu} = (\mu_1, \cdots, \mu_m)$ of plaintexts in $\prod_{i=1}^{m} \mathbb{Z}_p$ into a single ciphertext by the following transitions:

$$\begin{array}{ccccccc} \mathbb{Z}_p \times \cdots \times \mathbb{Z}_p & \longrightarrow & \prod_{i=1}^{m} \mathbb{Z}_p[X]/(F_i(X)) & \longrightarrow & \mathbb{Z}_p[X]/(\Phi(X)) & \longrightarrow & R_q \\ (\mu_1, \cdots, \mu_m) & \xmapsto{\mathsf{id}} & (\mu_1(X), \cdots, \mu_m(X)) & \xmapsto{\text{CRT}} & \mu(X) & \xmapsto{\mathsf{BGV.Enc}} & \bar{\mathbf{c}} \end{array}$$

First, we view a component μ_i in a vector $\boldsymbol{\mu} = (\mu_1, \cdots, \mu_m)$ as a contant polynomial $\mu_i \in \mathbb{Z}_p[X]/(F_i(X))$ for $i = 1, \cdots, m$. Then, we can compute the unique polynomial $\mu(X) \in R_p$ satisfying $\mu(X) = \mu_i \bmod (p, F_i(X))$ for $i = 1, \cdots, m$ by the Chinese Remainder Theorem (CRT) of polynomials. Finally, to encrypt a vector $\boldsymbol{\mu} = (\mu_1, \cdots, \mu_m)$ in $\prod_{i=1}^{m} \mathbb{Z}_p$, we encrypt the polynomial $\mu(X) \in R_p$ into a ciphertext $\bar{\mathbf{c}}$ which is denoted by $\mathsf{BGV.Enc}(\mu_1, \cdots, \mu_m)$. For more details to the ciphertext packing, we suggest that readers see the paper [22].

Homomorphic Matrix Operations Applying the resolution rules in the level-by-level analysis in the section 3.5 can be re-written in a matrix form as shown in Table 3. In Table 3, $\Delta_\ell = [\delta_{i,j}^{(\ell)}], H_\ell = [\eta_{i,j}^{(\ell)}], U = [u_{i,j}],$ and $V = [v_{i,j}]$ are $m \times m$ integer matrices. Let the i-th row of Δ_ℓ and H_ℓ be $\boldsymbol{\delta}_i^{(\ell)}$ and $\boldsymbol{\eta}_i^{(\ell)}$ respectively. And we denote the encryptions as $\bar{\boldsymbol{\delta}}_i^{(\ell)} = \mathsf{BGV.Enc}(\boldsymbol{\delta}_i^{(\ell)})$ and $\bar{\boldsymbol{\eta}}_i^{(\ell)} = \mathsf{BGV.Enc}(\boldsymbol{\eta}_i^{(\ell)})$.

We follow the methods in [16] to perform multiplication between encrypted matrices. We use the $\mathsf{Replicate}$ homomorphic operation supported by the BGV scheme [16]. For a given ciphertext $\bar{\mathbf{c}} = \mathsf{BGV.Enc}(\mu_1, \cdots, \mu_m)$, the operation $\mathsf{Replicate}(\bar{\mathbf{c}}, i)$ generates a ciphertext $\mathsf{BGV.Enc}(\mu_i, \cdots, \mu_i)$ for $i = 1, \cdots, m$. Using the operation, we can generate an encryption of the i-th row of $(H_\ell \cdot \Delta_\ell)$ as follows:

$$\mathsf{BGV.Mult}\left(\mathsf{Replicate}(\bar{\boldsymbol{\eta}}_i^{(\ell)}, 1), \bar{\boldsymbol{\delta}}_1^{(\ell)}\right) + \cdots + \mathsf{BGV.Mult}\left(\mathsf{Replicate}(\bar{\boldsymbol{\eta}}_i^{(\ell)}, m), \bar{\boldsymbol{\delta}}_m^{(\ell)}\right).$$

Note that this method does not affect the asymptotic notation of the multiplicative depth since the operation Replicate entails only a single multiplication.

To compute a transpose of an encrypted matrix, we use the masking and cyclic rotation techniques described in [16]. Algorithms for the homomorphic operations on encrypted matrices are described in Fig. 3 in Appendix A.

3.7 Randomization of Ciphertexts

During the matrix multiplications, components of resulting matrices may become p by coincidence, which is congruent to 0 in $\mathbb{Z}_p$. In this case, incorrect results may happen. We randomize intermediate results to decrease the failure probability.

To multiply the matrices $H_\ell = [\eta_{i,j}^{(\ell)}]$ and $\Delta_\ell = [\delta_{i,j}^{(\ell)}]$, we choose non-zero random elements $\{r_{i,j}\}$ in $\mathbb{Z}_p$ for $i, j = 1, \cdots, m$ and compute $H'_\ell = [r_{i,j} \cdot \eta_{i,j}^{(\ell)}]$. Then, each component of a resulting matrix of the matrix multiplication $(H'_\ell \cdot \Delta_\ell)$ is almost uniformly distributed over $\mathbb{Z}_p$.

Thanks to the randomization, the probability for each component of $H' \cdot \Delta$ of being congruent to zero modulo p is in inverse proportion to p. We may obtain a correct component with the probability of $(1 - \frac{1}{p-1})$. Because we perform in total $n(\lceil \log m \rceil + 3) - 2$ matrix multiplications for the analysis, the probability for a component of being correct is greater than $(1 - \frac{1}{p-1})^{n(\lceil \log m \rceil + 3)}$. For example, in the case where $n = 2, m = 1000$ and $p = 503$, the success probability for a component is about 95%.

Putting up altogether, we present the final protocol in Fig. 2 in Appendix A.

4 Experimental Result

In this section, we demonstrate the performance of the pointer analysis in secrecy. In our experiment, we use HElib library [16], an implementation of the BGV cryptosystem. We test on 4 small C example programs including tiny linux packages. The experiment was done on a Linux 3.13 system running on 8 cores of Intel 3.2 GHz box with 24GB of main memory. Our implementation runs in parallel on 8 cores using shared memory.

Table 4 shows the result. We set the security parameter 72 which is usually considered large enough. It means a ciphertext can be broken in a worst case time proportional to 2^{72}. In all the programs, the maximum pointer level is 2.

Why "Basic" Algorithm? Many optimization techniques to scale the pointer analysis to larger programs [12, 13, 17, 18, 20] cannot be applied into our setting without exposing much information of the program. Two key optimizations are the cycle elimination and the difference propagation. But neither method is applicable. The cycle elimination [12, 17, 18, 20] aims to prevent redundant computation of transitive closure by collapsing each cycle's components into a single node. The method cannot be applied into our setting because cycles cannot be

Table 4: Experimental Result

Program	**LOC**	**# Var**	**Enc**	**Propagation**	**Edge addition**	**Total**	**Depth**
toy	10	9	26s	28m 49s	5m 58s	35m 13s	37
buthead-1.0	46	17	1m 26s	5h 41m 36s	56m 19s	6h 39m 21s	43
wysihtml-0.13	202	32	2m 59s	18h 11m 50s	2h 59m 38s	21h 14m 27s	49
cd-discid-1.1	259	41	3m 49s	32h 22m 33s	5h 22m 35s	37h 48m 57s	49

Enc : time for program encryption, **Depth** : the depth required for the analysis
Propagation : time for homomorphic applications of the Trans rule
Edge addition : time for homomorphic applications of the Load and Store rules

detected and collapsed as all the program information and intermediate analysis results are encrypted. The other technique, difference propagation [13, 20], only propagates new reachability facts. Also, we cannot consider the technique because analysis server cannot determine which reachability fact is new as intermediate analysis results are encrypted.

5 Discussion

By combining language and cryptographic primitives, we confirm that the homomorphic encryption scheme can unleash the possibility of static analysis of encrypted programs. As a representative example, we show the feasibility of the pointer analysis in secrecy.

Although there is still a long way to go toward practical use, the experimental result is indicative of the viability of our idea. If the performance issue is properly handled in future, this idea can be used in many real-world cases. Besides depending on developments and advances in HE that are constantly being made, clients can help to improve the performance by encrypting only sensitive sub-parts of programs. The other parts are provided in plaintexts. In this case, analysis operations with the mixture of ciphertexts and plaintexts should be devised. This kind of operations are far cheaper than operations between ciphertexts because they lead to smaller noise increases.

A major future direction is adapting other kinds of static analysis operations(*e.g.,* arbitrary $\sqcup$, $\sqsubseteq$, and semantic operations) into HE schemes. For now, we expect other analyses similar to the pointer analysis (such as 0-CFA) will be performed in a similar manner.

Acknowledgment The authors would like to thank the anonymous reviewers for their valuable comments. The first and third authors were supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning(MSIP) / National Research Foundation of Korea(NRF) (Grant NRF-2008-0062609), and Samsung Electronics Software R&D Center (No. 0421-20140012). The second and last authors were supported by Samsung Electronics Software R&D Center (No. 0421-20140013).

References

1. Software clinic service. `http://rosaec.snu.ac.kr/clinic`.
2. D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. Wu. Private Database Queries Using Somewhat Homomorphic Encryption. In *ACNS*. 2013.
3. Z. Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *CRYPTO*. 2012.
4. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *ITCS*, 2012.
5. Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS*, 2011.
6. A. Chatterjee, M. Kaushal, and I. Sengupta. Accelerating Sorting of Fully Homomorphic Encrypted Data. In *INDOCRYPT*. 2013.
7. J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, and A. Yun. Batch Fully Homomorphic Encryption over the Integers. In *EUROCRYPT 2013*.
8. J. H. Cheon, M. Kim, and K. Lauter. Homomorphic Computation of Edit Distance. *IACR Cryptology ePrint Archive*, 2015:132, 2015. To appear in WAHC 2015.
9. J. H. Cheon and D. Stehlé. Fully homomorphic encryption over the integers revisited. In *EUROCRYPT*, 2015. To appear.
10. J.-S. Coron, D. Naccache, and M. Tibouchi. Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers. In *EUROCRYPT 2012*.
11. M. v. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *EUROCRYPT 2010*.
12. M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI 1998*.
13. C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *Nord. J. Comput.*, 5(4):304–329, 1998.
14. C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `http://crypto.stanford.edu/craig`.
15. C. Gentry, S. Halevi, and N. P. Smart. Homomorphic Evaluation of the AES Circuit. In *CRYPTO 2012*.
16. S. Halevi and V. Shoup. Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106, 2014. `http://eprint.iacr.org/`.
17. B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *PLDI 2007*.
18. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *PLDI 2001*.
19. N. Howgrave-Graham. Approximate Integer Common Divisors. In *CaLC*, 2001.
20. D. Pearce, P. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *SCAM*, 2003.
21. N. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography*, 71(1):57–81, 2014.
22. N. P. Smart and F. Vercauteren. Fully Homomorphic SIMD Operations. *IACR Cryptology ePrint Archive*, 2011:133.
23. N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *PKC*. 2010.

A Algorithms

Fig. 2 describes the protocol. Fig. 3 describes the homomorphic matrix operations and necessary sub algorithms.

Main Protocol

Client Input: There are m pointer variables in the client's program with the maximal pointer level n. The sets $\left\{(\delta_{i,j}^{(\ell)}, \eta_{i,j}^{(\ell)}) \mid 1 \le i,j \le m, 1 \le \ell \le n\right\}$ and $\{(u_{i,j}, v_{i,j}) \mid 1 \le i,j \le m\}$ are initialized as described in Section 3.2 and 3.5. For a security parameter λ, the client generates the parameters $(\mathsf{pk}, \mathsf{evk}; \mathsf{sk}) \leftarrow \mathsf{BGV.KG}(1^\lambda)$ of the BGV scheme.
Sub-algorithms: In this protocol, we use the sub-algorithms in Fig. 3.

– **Program Encryption** (Client's work)

```
1. for $\ell = 1$ to $n$ and for $i = 1$ to $m$ do
2.     $\bar{\boldsymbol{\delta}}_i^{(\ell)} \leftarrow \mathsf{BGV.Enc}(\delta_{i,1}^{(\ell)}, \cdots, \delta_{i,m}^{(\ell)})$, $\bar{\boldsymbol{\eta}}_i^{(\ell)} \leftarrow \mathsf{BGV.Enc}(\eta_{i,1}^{(\ell)}, \cdots, \eta_{i,m}^{(\ell)})$
3.     $\bar{\mathbf{u}}_i \leftarrow \mathsf{BGV.Enc}(u_{i,1}, \cdots, u_{i,m})$, $\bar{\mathbf{v}}_i \leftarrow \mathsf{BGV.Enc}(v_{i,1}, \cdots, v_{i,m})$
4. for $\ell = 1$ to $n$ do
5.     $\bar{\Delta}_\ell \leftarrow \left\langle \bar{\boldsymbol{\delta}}_1^{(\ell)} | \cdots | \bar{\boldsymbol{\delta}}_m^{(\ell)} \right\rangle^T$, $\bar{H}_\ell \leftarrow \left\langle \bar{\boldsymbol{\eta}}_1^{(\ell)} | \cdots | \bar{\boldsymbol{\eta}}_m^{(\ell)} \right\rangle^T$ // the $i$-th row of $\bar{\Delta}_\ell$ is $\bar{\boldsymbol{\delta}}_i^{(\ell)}$.
6. $\bar{U} \leftarrow \langle \bar{\mathbf{u}}_1 | \cdots | \bar{\mathbf{u}}_m \rangle^T$, $\bar{V} \leftarrow \langle \bar{\mathbf{v}}_1 | \cdots | \bar{\mathbf{v}}_m \rangle^T$ // the $i$-th row of $\bar{U}$ is $\bar{\mathbf{u}}_i$.
7. Client sends the sets $\left\{(\bar{\Delta}_\ell, \bar{H}_\ell) \mid 1 \le \ell \le n\right\}$ and $\left\{(\bar{U}, \bar{V})\right\}$ to server.
```

– **Analysis in Secrecy** (Server's work)

```
1. $\bar{\Delta}_n \leftarrow \mathsf{HE.MatMult}\left(\mathsf{HE.MatPower}(\bar{H}_n, m), \bar{\Delta}_n\right)$
2. for $\ell = n - 1$ to 1 do
3.     $\bar{A} \leftarrow \mathsf{HE.MatMult}(\bar{U}, \bar{\Delta}_{\ell+1})$, $\bar{B} \leftarrow \mathsf{HE.MatTrans}\left(\mathsf{HE.MatMult}(\bar{V}, \bar{\Delta}_{\ell+1})\right)$
4.     $\bar{H}_\ell \leftarrow \mathsf{HE.MatAdd}\left(\mathsf{HE.MatAdd}(\bar{H}_\ell, \bar{A}), \bar{B}\right)$ // apply Load and Store rules
5.     $\bar{\Delta}_\ell \leftarrow \mathsf{HE.MatMult}\left(\mathsf{HE.MatPower}(\bar{H}_\ell, m), \bar{\Delta}_\ell\right)$ // apply Trans rule
6. Server sends the ciphertext set $\left\{\bar{\boldsymbol{\delta}}_i^{(\ell)} \mid 1 \le \ell \le n \text{ and } 1 \le i \le m\right\}$ to client.
```

– **Output Determination** (Client's work)

```
1. for $i = 1$ to $m$ and for $\ell = 1$ to $n$ do
2.     Client computes $(\delta_{i,1}^{(\ell)}, \cdots, \delta_{i,m}^{(\ell)}) \leftarrow \mathsf{BGV.Dec}(\bar{\boldsymbol{\delta}}_i^{(\ell)})$.
3. Client determines the set $\left\{\mathtt{x_i} \longrightarrow \&\mathtt{x_j} \mid \delta_{i,j}^{(\ell)} \neq 0, 1 \le i,j \le m, 1 \le \ell \le n\right\}$.
```

Fig. 2: The Pointer Analysis in Secrecy

```
// We assume that $m$ is the same as the number of plaintext slots in the BGV scheme.
// A prime $p$ is the modulus of message space in the BGV-type cryptosystem.
// We denote the encryption of the matrix $A = [a_{i,j}] \in \mathbb{Z}_p^{m \times m}$ by $\bar{A}$.
// The $i$-th row $\bar{\mathbf{a}}_i$ of $\bar{A}$ is the ciphertext $\mathsf{BGV.Enc}(a_{i,1}, \cdots, a_{i,m})$ for $i = 1, \cdots, m$.
// For ciphertexts $\bar{\mathbf{c}}_1, \cdots, \bar{\mathbf{c}}_m$, we denote the matrix whose rows are $\bar{\mathbf{c}}_i$ by $\langle \bar{\mathbf{c}}_1 | \cdots | \bar{\mathbf{c}}_m \rangle^T$

HE.MatAdd($\bar{A}, \bar{B}$)
 // Input : $\bar{A}, \bar{B}$ are encryptions of $A = [a_{i,j}], B = [b_{i,j}]$.
 // Output : $\overline{A+B}$ is an encryption of $A + B = [a_{i,j} + b_{i,j}]$.
   1 for $i = 1$ to $m$ do      $\bar{\mathbf{z}}_i \leftarrow \mathsf{BGV.Add}(\bar{\mathbf{a}}_i, \bar{\mathbf{b}}_j)$
   2 return $\bar{Z} \leftarrow \langle \bar{\mathbf{z}}_1 | \bar{\mathbf{z}}_2 | \cdots | \bar{\mathbf{z}}_m \rangle^T$  // the $i$-th row of $\bar{Z}$ is $\bar{\mathbf{z}}_i$

HE.MatMult($\bar{A}, \bar{B}$)
 // Input : $\bar{A}, \bar{B}$ are encryptions of $A = [a_{i,j}], B = [b_{i,j}]$.
 // Output : $\overline{R_A \cdot B}$ is an encryption of $R_A \cdot B = \left[\sum_{k=1}^{m} r_{i,k} \cdot (a_{i,k} b_{k,j})\right]$,
 // where $r_{i,j} \xleftarrow{\$} [-p/2, p/2) \cap \mathbb{Z}$ with $r_{i,j} \neq 0$.
   1 $\bar{R} \leftarrow \mathsf{HE.MatRandomize}(\bar{A})$
   2 for $i = 1$ to $m$ do      $\bar{\mathbf{z}}_i \leftarrow \sum_{j=1}^{m} \mathsf{BGV.Mult}\left(\mathsf{HE.Replicate}(\bar{\mathbf{r}}_i, j), \bar{\mathbf{b}}_j\right)$  // ciphertext additions
   3 return $\bar{Z} \leftarrow \langle \bar{\mathbf{z}}_1 | \bar{\mathbf{z}}_2 | \cdots | \bar{\mathbf{z}}_m \rangle^T$  // the $i$-th row of $\bar{Z}$ is $\bar{\mathbf{z}}_i$

HE.MatPower($\bar{A}, k$)
 // Input : $\bar{A}$ is an encryption of $A$.
 // Output : $\overline{A^w}$ is an encryption of $A^w$, where $w = 2^{\lceil \log k \rceil}$.
   1 $\bar{Z} \leftarrow \bar{A}$
   2 for $i = 1$ to $\lceil \log k \rceil$ do      $\bar{Z} \leftarrow \mathsf{HE.MatrixMult}\left(\bar{Z}, \bar{Z}\right)$
   3 return $\bar{Z}$

HE.MatTrans($\bar{A}$)
 // Input : $\bar{A}$ is an encryption of $A = [a_{i,j}]$.
 // Output : $\overline{A^T}$ is an encryption of $A^T = [a_{j,i}]$.
   1 for $i = 1$ to $m$ do
   2       for $j = 1$ to $m$ do      $\bar{\mathbf{z}}_{i,j} \leftarrow \mathsf{HE.Masking}(\bar{\mathbf{a}}_j, i)$
   3       $\bar{\mathbf{z}}_i \leftarrow \sum_{j=1}^{i-1} \mathsf{HE.Rotate}(\bar{\mathbf{z}}_{i,j}, j - i + m) + \sum_{j=i}^{m} \mathsf{HE.Rotate}(\bar{\mathbf{z}}_{i,j}, j - i)$ // ciphertext additions
   4 return $\bar{Z} \leftarrow \langle \bar{\mathbf{z}}_1 | \bar{\mathbf{z}}_2 | \cdots | \bar{\mathbf{z}}_m \rangle^T$  // the $i$-th row of $\bar{Z}$ is $\bar{\mathbf{z}}_i$

HE.MatRandomize($\bar{A}$)
 // Input : $\bar{A}$ is an encryption of $A = [a_{i,j}]$.
 // Output : $\overline{R_A}$ is an encryption of $R_A = [r_{i,j} \cdot a_{i,j}]$, where $r_{i,j} \xleftarrow{\$} \mathbb{Z}_p$ with $r_{i,j} \neq 0$.
   1 for $i = 1$ to $m$ do
   2       Choose a vector $\mathbf{r}_i = (r_{i,1}, \cdots, r_{i,m}) \xleftarrow{\$} \mathbb{Z}_p^m$ with $r_{i,j} \neq 0 \bmod p$.
   3       $\bar{\mathbf{z}}_i \leftarrow \mathsf{BGV.multByConst}(\mathbf{r}_i, \bar{\mathbf{a}}_i)$
   4 return $\bar{Z} \leftarrow \langle \bar{\mathbf{z}}_1 | \bar{\mathbf{z}}_2 | \cdots | \bar{\mathbf{z}}_m \rangle^T$  // the $i$-th row of $\bar{Z}$ is $\bar{\mathbf{z}}_i$


// The following algorithms are in the library HElib.
// Here, we only give preview of the algorithms.

HE.Replicate($\bar{\mathbf{c}}, k$)
 // The ciphertext $\bar{\mathbf{c}}$ is the encryption of $(\mu_1, \cdots, \mu_m)$
  return the ciphertext $\mathsf{BGV.Enc}(\mu_k, \cdots, \mu_k)$

HE.Masking($\bar{\mathbf{c}}, k$)
 // The ciphertext $\bar{\mathbf{c}}$ is the encryption of $(\mu_1, \cdots, \mu_m)$
  return the ciphertext $\mathsf{BGV.Enc}(0, \cdots, 0, \mu_k, 0 \cdots, 0)$ // $\mu_k$ is the $k$-th plaintext slot.

HE.Rotate($\bar{\mathbf{c}}, k$)
 // The ciphertext $\bar{\mathbf{c}}$ is the encryption of $(\mu_1, \cdots, \mu_m)$
 // This operation is the right rotation as a linear array
  return the ciphertext $\mathsf{BGV.Enc}(\mu_{m-k+2}, \cdots, \mu_m, \mu_1, \cdots, \mu_{m-k+1})$

BGV.multByConst($\mathbf{r}, \bar{\mathbf{c}}$)
 // The operation of the multiply-by-constant induces "moderate" noise-growth,
 // while a multiplication of ciphertexts induces "expensive" noise-growth.
 // The constant vector $\mathbf{r} = (r_1, \cdots, r_m) \in \mathbb{Z}_p \times \cdots \times \mathbb{Z}_p$
 // The ciphertext $\bar{\mathbf{c}}$ is the encryption of $(\mu_1, \cdots, \mu_m)$
  return the ciphertext $\mathsf{BGV.Enc}(r_1\mu_1, \cdots, r_m\mu_m)$
```

Fig. 3: Pseudocode for the Homomorphic Matrix Operations