ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 NRF ERC

ROSAEC MEMO
2015-01
March 19, 2015

# Static Analysis with Set-closure in Secrecy

Woosuk Lee Hyunsook Hong Kwangkeun Yi Jung Hee Cheon

March 19, 2015

**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 inclusion-based 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

We have built a *static-analysis-as-a-service* system [2]. The analyzer is Sparrow [3], an industrial-strength static analyzer for C programs, implemented through our general techniques for improving the precision and scalability [22, 23, 24]. But the service has not been popular because of privacy concerns. Users are reluctant to upload their source to our analysis server.

For more widespread use of our system, 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 [16]. Since then, although there have been many efforts to improve the efficiency [5, 6, 11, 26], 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. [10] 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 [8]. Also, private database query protocol using somewhat homomorphic encryption has been proposed [4].

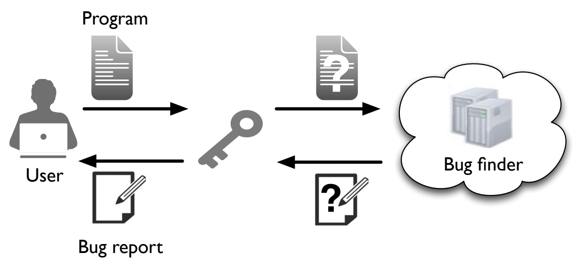

Figure 1: Secure static analysis is performed in 3 steps: 1) target program encryption 2) analysis in secrecy, and 3) analysis result decryption

**Our Results** As a first step, we propose an inclusion-based 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 inclusion-based 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.

### 2.1 Homomorphic Encryption

A homomorphic encryption (HE) scheme $\mathsf{HE}$=($\mathsf{KG}$, $\mathsf{Enc}$, $\mathsf{Dec}$, $\mathsf{Eval}$) 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 $\lambda$ as input and outputs a public encryption key $\mathsf{pk}$, a public evaluation key $\mathsf{evk}$, and a secret decryption key $\mathsf{sk}$.
- $\bar{c} \leftarrow \mathsf{HE.Enc}_{\mathsf{pk}}(\mu, r)$: The algorithm takes the public key $\mathsf{pk}$, a single 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 $\mathsf{sk}$ 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 $\mathsf{evk}$, a function $f : \{0,1\}^l \rightarrow \{0,1\}$ 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 $\mathsf{HE}$=($\mathsf{KG}$, $\mathsf{Enc}$, $\mathsf{Dec}$, $\mathsf{Eval}$) is *$f$-homomorphic* if for any set of inputs $(\mu_1, \cdots, \mu_l)$, and all sufficiently large $\lambda$, 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 [13] based on approximate common divisor problems [21]:

- $\mathsf{sk} \leftarrow \mathsf{KG}(1^\lambda)$: 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:

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

[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.

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 [6, 7, 12, 17] 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 [9, 13, 16, 27] 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 [6, 17]. In this section, we only provide a brief review of the cryptosystem [6]. For more details, please refer to the Appendix A.

Let $\Phi(X)$ be an irreducible polynomial over $\mathbb{Z}$. The implementation of the scheme is based on the polynomial operations in ring $R = \mathbb{Z}[X]/\left(\Phi(X)\right)$ which is the set of integer polynomials of degree less than $\deg(\Phi)$. Let $R_p \overset{\mathsf{def}}{=} R/pR$ 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 $\sigma$. 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 Appendix A 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 an inclusion-based pointer analysis in secrecy.

### 3.1 A Brief Review of Inclusion-based Pointer Analysis

We begin with a brief review of inclusion-based 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 inclusion-based pointer analysis algorithm with simple resolution rules in a similar manner to [20]. Given some program $P$, we construct resolution rules as specified in Table 2. In the first rule, the side condition "if $\mathtt{x} = \&\mathtt{y}$ in $P$" indicates that there is an instance of this rule for each occurrence of an assignment of the form $\mathtt{x} = \&\mathtt{y}$ in $P$. The side conditions in the other rules are similarly interpreted. Intuitively, an edge $\mathtt{x} \longrightarrow \&\mathtt{y}$ indicates that `x` can point to `y`. An edge $\mathtt{x} \longrightarrow \mathtt{y}$ indicates that for any variable `v`, if `y` may point to `v` then `x` may point to `v`. The pointer analysis is applying the resolution rules until reaching a fixpoint.

| | | |
|---|---|---|
| $\dfrac{}{\mathtt{x} \longrightarrow \&\mathtt{y}}$ | (if $\mathtt{x} = \&\mathtt{y}$ in $P$) | (New) |
| $\dfrac{}{\mathtt{x} \longrightarrow \mathtt{y}}$ | (if $\mathtt{x} = \mathtt{y}$ in $P$) | (Copy) |
| $\dfrac{\mathtt{x} \longrightarrow \&\mathtt{z}}{\mathtt{y} \longrightarrow \mathtt{z}}$ | (if $\mathtt{y} = *\mathtt{x}$ in $P$) | (Load) |
| $\dfrac{\mathtt{x} \longrightarrow \&\mathtt{z}}{\mathtt{z} \longrightarrow \mathtt{y}}$ | (if $*\mathtt{x} = \mathtt{y}$ in $P$) | (Store) |
| $\dfrac{\mathtt{x} \longrightarrow \mathtt{z} \quad \mathtt{z} \longrightarrow \&\mathtt{y}}{\mathtt{x} \longrightarrow \&\mathtt{y}}$ | | (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 inclusion-based 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 $\mathtt{x_i}$. 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

| | | |
|---|---|---|
| $\delta_{i,j} \neq 0$ | iff | An edge $\mathtt{x_i} \longrightarrow \&\mathtt{x_j}$ is derived by the resolution rules. |
| $\eta_{i,j} \neq 0$ | iff | An edge $\mathtt{x_i} \longrightarrow \mathtt{x_j}$ is derived by the resolution rules. |

for variables $\mathtt{x_i}$ and $\mathtt{x_j}$, and the number $m$ of pointer variables.

### 3.2.1 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 $\delta_{ij}$, 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.

### 3.2.2 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 $\mathsf{Trans}$ 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 $\mathsf{Trans}$ 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 $\mathsf{Load}$ 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 $\eta_{j,k}$.

Finally, to apply the $\mathsf{Store}$ 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 $\texttt{x}_\texttt{k} \longrightarrow \&\texttt{x}_\texttt{i}$ is derived and the program $P$ has a command $*\texttt{x}_\texttt{k} := \texttt{x}_\texttt{j}$ for some variable $\texttt{x}_\texttt{k}$, then an edge $\texttt{x}_\texttt{i} \longrightarrow \texttt{x}_\texttt{j}$ 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 $\bar{\delta}_{ij}$, and repeating the two steps $m^2$ times performs about $m^{m^2}$ homomorphic multiplications on each $\bar{\delta}_{ij}$.

### 3.2.3 Output Determination

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

$$\{\texttt{x}_\texttt{i} \longrightarrow \&\texttt{x}_\texttt{j} \mid \mathsf{HE.Dec}_{\mathsf{sk}}(\bar{\delta}_{i,j}) \neq 0 \text{ and } 1 \le i,j \le m\}$$

### 3.2.4 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 the following method:

$$\delta_{i,j} \leftarrow \bigvee_{1\le k\le m} \eta_{i,k} \wedge \delta_{k,j}$$

However, this representation causes more multiplicative depth than our current approach. The OR operation consists of the XOR and AND operations as follows:

$$x \vee y \overset{\texttt{def}}{=} (x \wedge y) \oplus x \oplus y$$

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.

## 4 Improvement 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.

## 4.1 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.

## 4.2 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 cipertexts 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 | $O(n \log m)$ | $(2n+2)m$ |

## 4.3 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 $\mathsf{ptl}(\mathtt{x})$.

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 |
|---|---|
| $\mathtt{x} = \mathtt{y}$ | $\mathsf{ptl}(\mathtt{x}) = \mathsf{ptl}(\mathtt{y})$ |
| $\mathtt{x} = *\mathtt{y}$ | $\mathsf{ptl}(\mathtt{y}) = \mathsf{ptl}(\mathtt{x}) + 1$ |
| $*\mathtt{p} = \mathtt{y}$ | $\mathsf{ptl}(\mathtt{p}) = \mathsf{ptl}(\mathtt{x}) + 1 \wedge \mathsf{ptl}(\mathtt{y}) = \mathsf{ptl}(\mathtt{x})$ |

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

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

### 4.3.1 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):

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

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

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

The definitions of $u_{ij}$ and $v_{ij}$ 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.

### 4.3.2 Server's Analysis

Server's analysis begins with propagating pointer values of the maximal level $n$ by applying the $\mathsf{Trans}$ 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 $\ell$ is carried out in the following steps:

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

### 4.3.3 Output Determination

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

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

## 4.4 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 $\bar{\mathbf{c}} \leftarrow \mathsf{BGV.Enc}(\mu_1, \cdots, \mu_m)$.

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 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 $\{a_0 + a_1 X + \cdots + a_{\deg \Phi - 1} X^{\deg \Phi - 1} \mid a_i \in \mathbb{Z}_p \text{ and } 0 \leq i < \deg \Phi\}$.

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:

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

First, we view a component $\mu_i$ 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)

| 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

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 [28].

### 4.4.1 Homomorphic Matrix Operations

Applying the resolution rules in the level-by-level analysis in the section 4.3 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 $\Delta_\ell$ and $H_\ell$ 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 [18] to perform multiplication between encrypted matrices. We use the Replicate homomorphic operation supported by the BGV scheme [18]. 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 [18]. Algorithms for the homomorphic operations on encrypted matrices are described in Fig. 3 and 4 in Appendix B.

## 4.5 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 B.

Table 4: Experimental Result

| **Program** | **LOC** | **# Var** | **Enc** | **Propagation** | **Edge addition** | **Total** | **Depth** |
|---|---|---|---|---|---|---|---|
| toy | 10 | 9 | 26s | 32m 24s | 5m 40s | 38m 29s | 37 |
| buthead-1.0 | 46 | 17 | 1m 26s | 7h 21m 58s | 1h 4m 52s | 8h 28m 17s | 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 | 40h 6m 42s | 5h 43m 21s | 45h 53m 52s | 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

## 5 Experimental Result

In this section, we demonstrate the performance of the pointer analysis in secrecy. In our experiment, we use HElib library [18], 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.

## 6 Discussion

### Why "Basic" Algorithm?

Many optimization techniques to scale inclusion-based pointer analysis to larger programs [14, 15, 19, 20, 25] cannot be applied into our setting without exposing much information of the program. Two key optimizations for inclusion-based pointer analysis are the cycle elimination and the difference propagation. But neither method is applicable. The cycle elimination [14, 19, 20, 25] 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 detected and collapsed as all the program information and intermediate analysis results are encrypted. The other technique, difference propagation [15, 25], 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.

### Privacy Preserving App Reviews

Our method may be used for app store review systems. App review systems (*e.g.,* Apple App Store, Samsung Apps) aim to filter malicious apps before deployments. In app review systems, a server-side analysis in secrecy may help for the following reasons:

- Analysis cannot be performed on the client-side because they may tamper with the analysis results.
- Revealed analysis mechanism may be used to avoid the detection.
- App source codes often require privacy for copyright protection.

A prerequisite for the realization of this scenario is a threshold cryptosystem. In threshold cryptosystems, two parties must cooperate in the decryption protocol. In our setting, the secret key is shared between analysis server and program owner. This decryption mechanism is for preventing the program owner from doing the decryption by himself and tampering with the result. Another prerequisite is a zero-knowledge protocol by which that the program owner did not maliciously change his original program is proved.

## 7 Conclusion

We report that the homomorphic encryption scheme can unleash the possibility of static analysis of encrypted programs. As a representative example, we have described an inclusion-based pointer analysis in secrecy. In our method, a somewhat homomorphic encryption scheme of depth $O(\log m)$ is able to evaluate the pointer analysis with $O(\log m)$ homomorphic matrix multiplications.

We also show the viability of our work by implementing the pointer analysis in secrecy. We expect our method will scale to larger programs thanks to new developments and advances in HE that are constantly being made.

## References

[1] On Ideal Lattices and Learning with Errors over Rings. In *EUROCRYPT 2010*.

[2] Software clinic service. `http://rosaec.snu.ac.kr/clinic`.

[3] Sparrow. `http://ropas.snu.ac.kr/sparrow`.

[4] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. Wu. Private Database Queries Using Somewhat Homomorphic Encryption. In M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *Applied Cryptography and Network Security*, volume 7954 of *Lecture Notes in Computer Science*, pages 102–118. Springer Berlin Heidelberg, 2013.

[5] Z. Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *CRYPTO 2012*.

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *ITCS '12 Proceedings of the 3rd Innovations in Theoretical Computer Science*, pages 309–325. ACM, 2012.

[7] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, FOCS'11, pages 97–106. IEEE Computer Society, 2011.

[8] A. Chatterjee, M. Kaushal, and I. Sengupta. Accelerating Sorting of Fully Homomorphic Encrypted Data. In G. Paul and S. Vaudenay, editors, *Progress in Cryptology - INDOCRYPT 2013*, volume 8250 of *Lecture Notes in Computer Science*, pages 262–273. Springer International Publishing, 2013.

[9] 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*.

[10] 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.

[11] J. H. Cheon and D. Stehlé. Fully homomorphic encryption over the integers revisited. In *EUROCRYPT 2015*. To appear.

[12] J.-S. Coron, D. Naccache, and M. Tibouchi. Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers. In *EUROCRYPT 2012*.

[13] M. v. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *EUROCRYPT 2010*.

[14] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI '98*.

[15] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *Nord. J. Comput.*, 5(4):304–329, 1998.

[16] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `http://crypto.stanford.edu/craig`.

[17] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic Evaluation of the AES Circuit. In *CRYPTO 2012*.

[18] S. Halevi and V. Shoup. Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106, 2014. `http://eprint.iacr.org/`.

[19] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *PLDI '07*.

[20] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *PLDI '01*.

[21] N. Howgrave-Graham. Approximate Integer Common Divisors. In *CaLC*, pages 51–66, 2001.

[22] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI 2012*.

[23] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. In *PLDI '14*.

[24] H. Oh and K. Yi. Access-based abstract memory localization in static analysis. *Science of Computer Programming*, 78(9):1701–1727, 2013.

[25] D. Pearce, P. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 3–12, 2003.

[26] N. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography*, 71(1):57–81, 2014.

[27] N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *PKC 2010*. 2010.

[28] N. P. Smart and F. Vercauteren. Fully Homomorphic SIMD Operations. *IACR Cryptology ePrint Archive*, 2011:133, 2011.

## A The BGV-type Cryptosystem

**Notations.**

For an integer $q$, we denote the ring of integers modulo $q$ by $\mathbb{Z}_q$. Let $\Phi(X)$ be an irreducible polynomial over $\mathbb{Z}$. Our implementation is based on the operations in polynomial ring $R = \mathbb{Z}[X]/\left(\Phi(X)\right)$ which is the set of integer polynomials of degree less than $\deg\Phi$. We identify the quotient ring $R_q := R/qR$ with the set of integer polynomials of degree up to $\deg\Phi - 1$ reduced modulo $q$ for the integer $q$ (*i.e.*, $R_q = \{a_0 + a_1X + \cdots + a_{\deg\Phi-1}X^{\deg\Phi-1} \mid a_i \in \mathbb{Z}_p \text{ and } 0 \leq i < \phi(N)\}$).

**The BGV Scheme.**

Our solutions are implemented with the efficient variant of the Brakerski-Gentry-Vaikuntanathan (BGV) cryptosystem using a modulus switching technique. We recall that the BGV cryptosystem [6] based on the hardness of the "ring learning with errors " (RLWE) problem [1]. The RLWE problem is to distinguish pair $(a_i, b_i = a_i \cdot \mathbf{s} + e_i) \in R_q \times R_q$ from uniformly random pairs, where $\mathbf{s} \in R_q$ is a random "secret" polynomial which remains fixed over all pairs, the $a_i \in R_q$ are uniformly random and independent, and the "noise" terms $e_i \in R$ are sampled from a noise distribution that outputs polynomials whose coefficients much "smaller" than $q$ (an example is a discrete Gaussian distribution over $R$ with small standard deviation).

For a polynomial ring $R = \mathbb{Z}[X]/\left(\Phi(X)\right)$, we set the message space to $R_p$ for some fixed prime $p \geq 2$ and the ciphertext space to $R_q \times R_q$ for an integer $q$. Then all the ciphertexts are treated as vectors of elements in $R_q$. Now, we describe the BGV cryptosystem as follows:

- $((a, b), \mathsf{evk}; \mathbf{s}) \leftarrow \mathsf{BGV.KG}(1^\lambda, w, \sigma, q)$: Chooses a weight $w$ secret key $\mathbf{s}$ and generates a RLWE instance $(a, b)$ relative to the secret key $\mathbf{s}$. Compute a evaluation key for a homomorphic evaluation of ciphertexts. Output the public key $\mathsf{pk} = (a, b)$, the evaluation key $\mathsf{evk}$, and the secret key $\mathsf{sk} = \mathbf{s}$.

- $\bar{\mathbf{c}} \leftarrow \mathsf{BGV.Enc}_{\mathsf{pk}}(\mu)$: To encrypt a message $\mu \in R_t$, choose a random polynomial $v$ whose coefficients are in $\{0, \pm 1\}$ and two noise polynomials $e_0, e_1$ from a discrete Gaussian distribution over $R$ with standard deviation $\sigma$. Outputs the ciphertext $\mathbf{c} = (c_0, c_1) = (bv + pe_0 + \mu, av + pe_1) \bmod q$.

- $\mu \leftarrow \mathsf{BGV.Dec}_{\mathsf{sk}}(\bar{\mathbf{c}})$: 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 ciphertext $\bar{\mathbf{c}}_1 = \mathsf{BGV.Enc}(\mu_1)$ 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 ciphertext $\bar{\mathbf{c}}_1 = \mathsf{BGV.Enc}(\mu_1)$ and $\bar{\mathbf{c}}_2 = \mathsf{BGV.Enc}(\mu_2)$, it outputs the ciphertext $\bar{\mathbf{c}}_{\mathsf{mult}} = \mathsf{BGV.Enc}(\mu_1 \cdot \mu_2)$.

In the BGV scheme, homomorphic addition is done by simple component-wise addition of the ciphertexts and homomorphic multiplication is by tensor product over $R_q$. Since the norm of the noise and the degree of the ciphertext are increased after operations of ciphertexts, modulus and key switching operation should be performed to reduce the norm of the noise and the degree of the ciphertext. For more details to the homomorphic operations on the BGV-type cryptosystem such as the key switching and modulus switching, please refer to [6, 17].

## B Algorithms

In this section, we describe algorithms for the andersen's analysis in secrecy. Fig. 2 describes the protocol. Fig. 3 and 4 describe the homomorphic matrix operations and sub algorithms necessary for the evaluation of the protocol respectively.

**Main Protocol**

**Client Input:** There $m$ pointer variables in the client's program with the maximal pointer level $n$. The sets $\left\{(\delta_{i,j}^{(\ell)}, \eta_{i,j}^{(\ell)}) \mid 1 \leq i, j \leq m, 1 \leq \ell \leq n\right\}$ and $\{(u_{i,j}, v_{i,j}) \mid 1 \leq i, j \leq m\}$ are initialized by the manner in the section 3.2 and 4.3. For a security parameter $\lambda$, 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 sub-algorithms in Fig. 3 and 4.

– **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.     $\bar{\boldsymbol{u}}_i \leftarrow \mathsf{BGV.Enc}(u_{i,1}, \cdots, u_{i,m})$, $\bar{\boldsymbol{v}}_i \leftarrow \mathsf{BGV.Enc}(v_{i,1}, \cdots, v_{i,m})$
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 $\left\{(\bar{\Delta}_\ell, \bar{H}_\ell) \mid 1 \leq \ell \leq n\right\}$ 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 $\left\{\bar{\boldsymbol{\delta}}_i^{(\ell)} \mid 1 \leq \ell \leq n \text{ and } 1 \leq i \leq m\right\}$ 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 $\left\{\mathtt{x_i} \longrightarrow \&\mathtt{x_j} \mid \delta_{i,j}^{(\ell)} \neq 0, 1 \leq i, j \leq m, 1 \leq \ell \leq n\right\}$.
```

Figure 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
  2          $\bar{\mathbf{z}}_i \leftarrow \mathsf{BGV.Add}(\bar{\mathbf{a}}_i, \bar{\mathbf{b}}_j)$
  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.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 $R_A \cdot B = [\sum_{k=1}^{m} r_{i,k} \cdot (a_{i,k} b_{k,j})]$,
 // 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
  3          $\bar{\mathbf{z}}_i \leftarrow \sum_{j=1}^{m} \mathsf{BGV.Mult}\left(\mathsf{HE.Replicate}(\bar{\mathbf{r}}_i, j), \bar{\mathbf{b}}_j\right)$  // ciphertext additions
  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.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
  3          $\bar{Z} \leftarrow \mathsf{HE.MatrixMult}\left(\bar{Z}, \bar{Z}\right)$
  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
  3                  $\bar{\mathbf{z}}_{i,j} \leftarrow \mathsf{HE.Masking}(\bar{\mathbf{a}}_j, i)$
  5          $\bar{\mathbf{z}}_i \leftarrow \sum_{j=1}^{i-1} \mathsf{HE.Rotate}(\bar{\mathbf{z}}_{i,j}, j - i + m) + \sum_{j=i}^{m} \mathsf{HE.Rotate}(\bar{\mathbf{z}}_{i,j}, j - i)$
              // $\triangle$ ciphertext additions
  6   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          $\bar{\mathbf{z}}_i \leftarrow \mathsf{BGV.multByConst}(\mathbf{r}_i, \bar{\mathbf{a}}_i)$
  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 $\bar{\mathbf{z}}_i$
```

Figure 3: Pseudocode for the Homomorphic Matrix Operations

```
// 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 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 BGV.Enc($0, \cdots, 0, \mu_k, 0 \cdots, 0$) // $k$-th of plaintext slots is $\mu_k$

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 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 BGV.Enc($r_1\mu_1, \cdots, r_m\mu_m$)
```

Figure 4: Pseudocode for Some Homomorphic Algorithms