공학박사학위논문

정적 분석기 사용자 편의성 증대에 관한 연구

Improving the Usability of Static Analyzers

2016년 2월

서울대학교 대학원

전기컴퓨터공학부

이 우 석

정적 분석기 사용자 편의성 증대에 관한 연구

지도교수 이 광 근

이 논문을 공학박사학위논문으로 제출함
2015년 10월

서울대학교 대학원
전기컴퓨터공학부
이 우 석

이 우 석의 박사학위논문을 인준함
2015년 12월

Abstract

Improving the Usability of Static Analyzers

Woosuk Lee
Department of Electrical Engineering & Computer Science
College of Engineering
Seoul National University

As programs become larger and more complex, users of static analyzers often encounter three usability issues. Firstly, static analyzers often produce a large number of both true and false alarms that are tedious to classify manually. Secondly, users cannot but wait long time without any progress information during analysis. Lastly, copy-right concerns over software sources hinder extensive uses of static analyzers.

In this dissertation, we present our solutions to the three usability issues. To reduce users' alarm-classification efforts, we propose a sound method for clustering static analysis alarms. Our method clusters alarms by discovering sound dependencies between them such that if the dominant alarms of a cluster turns out to be false, all the other alarms in the same cluster are guaranteed to be false. Once clusters are found, users only need to investigate their dominant alarms. Next, we present a progress indicator of static analyzers. Our technique first combines a semantic-based pre-analysis and a statistical method to approximate how a main analysis progresses in terms of lattice height of the abstract domain. Then, we use this information during the main analysis and estimate the analysis' current progress. Lastly, we present a static analysis of encrypted programs to resolve users' copy-right concerns over software sources. Users have purchased expensive commercial static analyzers or outsource static analyses on their programs to analysis servers taking the risk of loss of copy-right. Our method allows 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.

We have implemented all the methods on top of a realistic static analyzer for C programs and empirically proved that our methods effectively improve the usability.

Keywords	:	Usability, Progress Estimation, Static Analysis, False Alarms, Homomorphic Encryption, Abstract Interpretation, Non-statistical Clustering, Pointer Analysis, Buffer-overflow Detection
Student Number	:	2009-20866

Contents

List of Tables

List of Figures

Chapter 1

Overview

1.1 Problems

Static program analysis (or simply "static analysis") offers static compile-time techniques that safely estimate the program properties observable at run-time. Sound static analysis captures all possible run-time behaviors of programs. Thus, it can be used to prove the absence of run-time errors in programs.

As programs become larger and more complex, users of static analyzers often encounter the following major usability issues:

- **Many false alarms** : Users of sound static analyzers often encounter false positives that cause tedious alarm-investigation efforts. To capture all possible run-time behaviors of programs, any sound static analysis should inevitably be incomplete in the sense that it can only provide over-approximate answers. These over-approximations are often imprecise as target programs become more complex. For example, to guarantee the termination of fixpoint computation in complex control-flow cycles in a program, extrapolation (*e.g.,* widening) is often applied. Such extrapolations often lead to imprecise analysis results.

- **Missing progress indicator** : Other than almost syntactic properties, once the target property becomes slight deep in semantics static analyzers usually take a long time to analyze real-world complex software.

  However, users cannot but wait long time without any progress information. For instance, Sparrow [31, 49, 50, 51, 52, 61], our realistic static analyzer for full C, takes more than 10 hours to analyze 400K lines of C code. ASTRÉE, a domain-aware analyzer for the primary flight-control software, has also been reported to take over 20 hours to analyze programs of size over 500KLOC [16]. Nonetheless, such static analyzers are silent during their operation.

  It is because estimating static analysis progress at real-time has been considered challenging in general. Static analyzers take most of their time in fixpoint computation, but estimating the progress of fixpoint algorithms has been unknown. One challenge is that the analysis time is generally not proportional to the size of the program to analyze. For instance, Sparrow takes 4 hours in analyzing one million lines but require 10 hours to analyze programs of sizes around 400KLOC [49]. Similar observations have been made for ASTRÉE as well: ASTRÉE takes 1.5 hours for 70KLOC but takes 40 minutes for 120KLOC [16].

- **Copy-right concerns** : Copy-right concerns over software sources have been the major cause that prevents extensive uses of commercial static analyzers. Target programs often require copy-right protections. Also, to handle large complex programs, static analyzers should be equipped with sophisticated approximation methods [45, 56] and cost-reduction techniques for better scalability [49, 50, 51, 52]. These techniques necessitate the copy-right protection of commercial static analyzers.

  So far, users should either purchase expensive static analyzers or outsource static analyses on their programs to analysis servers taking risk of the loss of copy-right.

1.2 Solutions

- **Sound non-statistical alarm clustering** [39] : To reduce users' alarm-investigation burden caused by many false alarms, we propose a method of clustering alarms according to their sound dependence information. We say that alarm A has sound dependence on alarm B if alarm B turns out to be false, then so does alarm A as a logical consequence. When we find a set of alarms depending on the same alarm, which we call a dominant alarm, we can cluster them together. Once we find clusters of alarms, users only need to check whether their dominant alarms are false.

  Our method automatically discovers sound dependencies among alarms. Our basic approach is to perform a post-analysis assuming some dominant alarm candidates to be false and see if the other existing alarms are suppressed by the assumption.

  We prove the effectiveness of our clustering method with a realistic static analyzer for buffer-overflow detection. On 14 open-source benchmarks, our clustering method identified 45% of alarms to be non-dominating. This result amounts to 45% reduction in the number of investigated alarms if the other 55% turns out to be false.

  Our framework is general and applicable to any semantic-based static analyzers. It is orthogonal to other methods for reducing alarm-investigation burden such as refining approaches [24, 25, 33, 55] or statistical ranking schemes [31, 35, 36].

- **A progress bar for static analyzers** [40] : We propose a progress bar for static analyzers. We estimate analysis progress by calculating lattice heights of intermediate analysis results and comparing them with the height of the final analysis result. To this end, we employ a semantic-based pre-analysis and a machine learning technique. First, we use the pre-analysis to approximate the height of the fixpoint. This estimated height is then fine-tuned with the statistical method. Second, because this height progress usually does not indicate the actual progress (speed), we normalize the progress using the pre-analysis.

We show that our technique effectively estimates static analysis progress in a realistic setting. We have implemented our idea on top of Sparrow. In our experiments with various open-source benchmarks, the proposed technique is found to be useful to estimate the progress of three different kinds of analyses (interval, octagon, and pointer analysis). The pre-analysis overheads are 3.8%, 7.3%, and 36.6% on average in each kind of analyses (interval, pointer, and octagon analysis), respectively.

- **Static analysis with set-closure in secrecy** [38] : To resolve the copyright concerns, we report that the homomorphic encryption (HE) scheme can unleash the possibility of static analysis of encrypted programs. A HE scheme enables computation of arbitrary functions on encrypted data. Our method allows 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.

  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.

  To overcome the high complexity of HE operations, we propose two optimization techniques. One is *level-by-level analysis*. We analyze the pointers of the same level together from the highest to lowest. The technique decreases both each ciphertexts size and the cost of each homomorphic operation. The other is to exploit the *ciphertext packing* technique in cryptography not only for performance boost but also for decreasing the total size of ciphertexts required for the program encryption.

1.3 Outline

The rest of this dissertation is organized as follows:

- Chapter 2 defines preliminary concepts and static analyses used in our methods.
- Chapter 3 presents our sound alarm clustering method.
- Chapter 4 presents our progress bar for static analyzers.
- Chapter 5 presents our methods for static analysis with set-closure in secrecy.
- Chapter 6 discusses related works and Chapter 7 concludes the dissertation.

Chapter 2

Preliminaries

2.1 Concepts

In this section, we define preliminary concepts.

Programs We represent a program P as a transition system $(\mathbb{S}, \rightarrow, \mathbb{S}_\iota)$ where $\mathbb{S}$ is the set of states of the program, $(\rightarrow) \subseteq \mathbb{S} \times \mathbb{S}$ is the transition relation of the possible, elementary execution steps, and $\mathbb{S}_\iota \subseteq \mathbb{S}$ denotes the set of initial states.

Collecting Semantics We write $\mathbb{S}^+$ for the set of all finite non-empty sequences of states. If $\sigma \in \mathbb{S}^+$ is a finite sequence of states, σ_i denotes the $(i+1)$-th state of the sequence, σ_0 is the first state, and $\sigma_\dashv$ the last state. If τ is a prefix of σ, we write $\tau \preceq \sigma$.

We say a sequence σ is a *trace* if σ is a (partial) execution sequence, i.e., $\sigma_0 \in \mathbb{S}_\iota \wedge \forall k. \sigma_k \rightarrow \sigma_{k+1}$. The trace semantics of program P is defined as the set of all traces of the program:

$$[\![P]\!] = \{\sigma \in \mathbb{S}^+ \mid \sigma_0 \in \mathbb{S}_\iota \wedge \forall i. \sigma_i \rightarrow \sigma_{i+1}\}$$

Note that the set $[\![P]\!]$ is a least fixpoint of the following semantic function F_P:

$$\begin{array}{rcl} F_P & : & \wp(\mathbb{S}^+) \to \wp(\mathbb{S}^+) \\ F_P(E) & = & \{\langle s_\iota \rangle \mid s_\iota \in \mathbb{S}_\iota\} \\ & \cup & \{\langle s_0, \cdots, s_{n+1}\rangle \mid \langle s_0, \cdots, s_n\rangle \in E \wedge s_n \to s_{n+1}\}. \end{array}$$

That is, $[\![P]\!] = \mathsf{lfp}\ F_P$.

Abstract Semantics We obtain the static analyzer by abstracting the trace semantics in two steps. First, we abstract the set of traces (i.e. $\wp(\mathbb{S}^+)$) into partitioned sets of reachable-states (i.e. $\Phi \to \wp(\mathbb{S})$ where Φ is a pre-defined set of partitioning indices). Next, we abstract the set of states associated with each partitioning index into an abstract state ($\hat{\mathbb{S}}$), leading to the final abstract domain $\hat{\mathbb{D}} = \Phi \to \hat{\mathbb{S}}$. The overall abstraction is formalized by the following Galois-connection:

$$\wp(\mathbb{S}^+) \underset{\alpha_0}{\overset{\gamma_0}{\leftrightarrows}} \Phi \to \wp(\mathbb{S}) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \Phi \to \hat{\mathbb{S}}.$$

We call the first part *partitioning abstraction* and the second part *set of states abstraction.*

1. Partitioning abstraction: Suppose that we have a pre-defined set Φ of indices and a partitioning function

$$\delta : \Phi \to \wp(\mathbb{S}^+)$$

   which maps partitioning indices (Φ) to a set of traces. We assume that the partitioning function is well-formed in a sense that it covers all the traces, i.e., $\bigcup_{\varphi \in \Phi} \delta(\varphi) = \mathbb{S}^+$, and all the associated sets are disjoint, i.e., $\forall \varphi_1, \varphi_2.\ \varphi_1 \neq \varphi_2 \implies \delta(\varphi_1) \cap \delta(\varphi_2) = \emptyset$.

   Example 2.1.1. The most popular strategy is the so-called flow-sensitivity that partitions the set of traces based on the program point of the final state; when a state is a pair of program point ($\mathbb{C}$) and a memory state ($\mathbb{M}$), i.e.,

$\mathbb{S} = \mathbb{C} \times \mathbb{M}$, this final program point partitioning is defined by the partitioning function $\delta_p(c) = \{\sigma \mid \exists\rho.\sigma_\dashv = (c, \rho)\}$; the set $\mathbb{C}$ of program points forms the partitioning indices Φ and δ_p classifies the set of traces according to their final program points. Other conventional partitioning strategies such as context-sensitivity, path-sensitivity, loop-unrolling are also obtained by defining appropriate partitioning indices Φ and function δ.

With a given partitioning function δ, we first define the partitioned reachable-state domain $\Phi \to \wp(\mathbb{S})$, which is defined by the following Galois-connection:

$$\wp(\mathbb{S}^+) \underset{\alpha_0}{\overset{\gamma_0}{\leftrightarrows}} \Phi \to \wp(\mathbb{S})$$

where the abstraction function α_0 and the concretization function γ_0 are defined as follows:

$$\begin{array}{rcl} \alpha_0(\Sigma) & = & \lambda\varphi.\{\sigma_\dashv \mid \sigma \in \Sigma \cap \delta(\varphi)\} \\ \gamma_0(f) & = & \{\sigma \mid \forall\tau \preceq \sigma.\forall\varphi \in \Phi.\ \tau \in \delta(\varphi) \Rightarrow \tau_\dashv \in f(\varphi)\}. \end{array}$$

We write concrete semantics $[\![P]\!]$ modulo the partitioning function δ as $[\![P]\!]_{/\delta}$, i.e., $[\![P]\!]_{/\delta} \in \Phi \to \wp(\mathbb{S})$.

2. Set of states abstraction: We further abstract the partitioned reachable states by the following Galois-connection:

$$\Phi \to \wp(\mathbb{S}) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \Phi \to \hat{\mathbb{S}}.$$

The Galois-connection of (α_1, γ_1) is defined as pointwise lifting of Galois-connection (α_S, γ_S) of states abstraction $\wp(\mathbb{S}) \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \hat{\mathbb{S}}$.

From this point, we will denote α and γ as $\alpha_1 \circ \alpha_0$ and $\gamma_0 \circ \gamma_1$ respectively.

The abstract semantics of program P computed by the analyzer is a fixpoint $[\![\hat{P}]\!] = \mathsf{lfp}^\# \hat{F}$ where the function $\hat{F} : (\Phi \to \hat{\mathbb{S}}) \to (\Phi \to \hat{\mathbb{S}})$ is a monotone or an extensive abstract transfer function such that $\alpha \circ F_P \sqsubseteq \hat{F} \circ \alpha$ and $\mathsf{lfp}^{\#}$ is a sound, abstract post-fixpoint operator which is defined as follows:

$$\bigsqcup_{i \in \mathbb{N}} \hat{F}^i(\hat{\bot}) = \hat{F}^0(\hat{\bot}) \sqcup \hat{F}^1(\hat{\bot}) \sqcup \hat{F}^2(\hat{\bot}) \sqcup \cdots \tag{2.1}$$

where $\hat{F}^0(\hat{\bot}) = \hat{\bot}$ and $\hat{F}^{i+1}(\hat{\bot}) = \hat{F}(\hat{F}^i(\hat{\bot}))$ and $\hat{\bot} = \bot_{\Phi \to \hat{\mathbb{S}}}$. The analysis' job is to compute the above sequence until stabilized. When the chain is infinitely long, we can use a widening operator $\bigtriangledown : (\Phi \to \hat{\mathbb{S}}) \times (\Phi \to \hat{\mathbb{S}}) \to (\Phi \to \hat{\mathbb{S}})$ to accelerate the sequence.

The soundness of the static analysis follows from the fixpoint transfer theorem [18].

2.2 Static Analyses We Use

In this section, we define three underlying static analyses. Our methods are implemented on top of the analyses.

Programs as CFGs We consider programs that can be represented by control flow graphs. A program is a tuple $\langle \mathbb{C}, \hookrightarrow \rangle$ where $\mathbb{C}$ is a set of program points, $(\hookrightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ is a relation that denotes control flows: $c \hookrightarrow c'$ indicates that c' is a next program point of c. Each program point is associated with a command: $c(c)$ denotes the command associated with program point c.

2.2.1 Interval Analysis

Now we describe a baseline analyzer that is an interval domain-based flow-sensitive abstract interpreter [2]. The analyzer is a realistic buffer-overflow detector performing sound and inter-procedural analysis.

To simplify the presentation, we consider a simple language and a program property. Each variable has an integer value in the simple language. The target program property we consider is about size relationships between variables.

Program Commands Each command in a node (or program point) $c \in \mathbb{C}$ in the graph has one of the following command, denoted $\texttt{cmd}(c)$:

$$\begin{array}{lcl} \text{command} & c & \rightarrow & \texttt{x} := e \mid \{\!\{\texttt{x} \leq \texttt{n}\}\!\} \mid \texttt{x} := \texttt{unknown()} \\ \text{expression} & e & \rightarrow & \texttt{n} \mid \texttt{x} \mid e \ \texttt{+} \ e \end{array}$$

An (side-effect-free) expression is either constant integer ($\texttt{n}$), binary operation (e + e), or variable ($\texttt{x}$). The command $\texttt{x} := e$ assigns the value of e into $\texttt{x}$. The command $\{\!\{\texttt{x} \leq \texttt{n}\}\!\}$ makes the program continue only when the condition evaluates to true. The command $\texttt{x} := \texttt{unknown()}$ assigns an arbitrary integer into $\texttt{x}$.

Collecting Semantics Collecting semantics of a program P is an invariant $[\![P]\!]_{/\delta_p} : \mathbb{C} \rightarrow \wp(\mathbb{S})$ where δ_p is the final program point partitioning function described in §2.1. It represents a set of reachable states at each program point, where the concrete domain of states $\mathbb{S}$ is the set of finite maps from variables (Var) to integers ($\mathbb{Z}$).

Abstract Semantics In our analysis, the set of (possibly infinite) concrete memory states for each program point are abstracted by an abstract memory state ($\hat{\mathbb{S}}_{\mathbb{I}} = \mathit{Var} \xrightarrow{\text{fin}} \mathbb{I}$), a finite map from variables (Var) to interval values ($\mathbb{I}$) that abstract a set of integers:

$$\mathbb{I} \;=\; \{\bot\} \cup \{[l,u] \mid l \in \mathbb{Z} \cup \{-\infty\} \wedge u \in \mathbb{Z} \cup \{+\infty\} \wedge l \leq u\}.$$

The pair of functions $(\alpha_{\mathbb{I}}, \gamma_{\mathbb{I}})$ forms a Galois connection: $\wp(\mathbb{S}) \underset{\alpha_{\mathbb{I}}}{\overset{\gamma_{\mathbb{I}}}{\leftrightarrows}} \hat{\mathbb{S}}_{\mathbb{I}}$.

For each node, we define a transfer function $\hat{f}_{\mathbb{I}} : \mathbb{C} \rightarrow \hat{\mathbb{S}}_{\mathbb{I}} \rightarrow \hat{\mathbb{S}}_{\mathbb{I}}$ that, given an input memory state, computes the effect of the assignment in the node on the input state:

$$\hat{f}_{\mathbb{I}}\ c\ \hat{m} = \begin{cases} \hat{m}[x \mapsto \hat{\mathcal{V}}(e)(\hat{m})] & (\texttt{cmd}(c) = \texttt{x} := e) \\ \hat{m}[x \mapsto \hat{m}(x) \sqcap [-\infty, n]] & (\texttt{cmd}(c) = \{\!\{\texttt{x} \leq \texttt{n}\}\!\}) \\ \hat{m}[x \mapsto [-\infty, \infty]] & (\texttt{cmd}(c) = \texttt{x} := \texttt{unknown()}) \end{cases}$$

The effect of node $\{\!\{\texttt{x} \leq \texttt{n}\}\!\}$ is to confine the interval value of $\texttt{x}$ according to the condition. The effect of node $\texttt{x} := e$ is to assign the abstract value of e into variable x. The effect of node $\texttt{x} := \texttt{unknown()}$ is to assign the top interval value into variable x. Given expression e and abstract memory state $\hat{m}$, auxiliary function $\hat{\mathcal{V}}$ computes abstract values:

$$\begin{array}{rcl}
\hat{\mathcal{V}}(e) & : & \hat{\mathbb{S}}_{\mathbb{I}} \to \mathbb{I} \\
\\
\hat{\mathcal{V}}(\texttt{n})(\hat{m}) & = & [n, n] \\
\hat{\mathcal{V}}(e_1 \texttt{ + } e_2)(\hat{m}) & = & \hat{\mathcal{V}}(e_1)(\hat{m}) \hat{+} \hat{\mathcal{V}}(e_2)(\hat{m}) \\
\hat{\mathcal{V}}(\texttt{x})(\hat{m}) & = & \hat{m}(x)
\end{array}$$

We skip the conventional definition of the abstract binary ($\hat{+}$) and join ($\sqcup$) operations in interval domain.

The analyzer computes a fixpoint table $[\![\hat{P}]\!]^{\mathbb{I}} \in \mathbb{C} \to \hat{\mathbb{S}}_{\mathbb{I}}$ that maps each node in the program to its output abstract memory state. The abstract memory state at each program point approximates all the concrete memory states occurring at the node in the concrete executions. The map is defined by the least fixpoint of the following function:

$$\begin{array}{c}
F_{\mathbb{I}} : (\mathbb{C} \to \hat{\mathbb{S}}_{\mathbb{I}}) \to (\mathbb{C} \to \hat{\mathbb{S}}_{\mathbb{I}}) \\
F_{\mathbb{I}}([\![\hat{P}]\!]) = \lambda c. \hat{f}_{\mathbb{I}}\ c\ (\bigsqcup_{c' \hookrightarrow c} [\![\hat{P}]\!](c'))
\end{array}$$

The fixpoint table $[\![\hat{P}]\!]^{\mathbb{I}}$ is a sound approximation of the collecting semantics of the program, i.e., $\forall c \in \mathbb{C}.\ \gamma_{\mathbb{I}}([\![\hat{P}]\!]^{\mathbb{I}}(c)) \sqsupseteq [\![P]\!]_{/\delta_p}(c)$

Fixpoint Computation with Widening As the interval domain has infinite height, we need a widening operator to approximate the least fixpoint of $F_{\mathbb{I}}$. The widening operator is applied at only headers of flow cycles [5]. Let $\mathbb{W} \subseteq \mathbb{C}$ be the set of widening points (all loop headers in the program) in the program.

Example 2.2.1. We use the following widening operator in our interval analysis:

$$[l, u] \triangledown [l', u'] = [\textit{if}\ \ (l' < l)\ \ \textit{then} - \infty\ \ \textit{else}\ \ l, \textit{if}\ \ (u' > u)\ \ \textit{then}\ \ + \infty\ \ \textit{else}\ \ u].$$

2.2.2 Octagon Analysis

Now we describe the octagon domain-based analysis. The octagon abstract domain [45] captures relational properties between variables. Our design of the octagon domain-based analysis is based on the same program representation and collecting semantics as in the interval analysis (§ 2.2.1).

Abstract Semantics Octagon domain $\hat{\mathbb{S}}_{\mathbb{O}}$ represents a set of octagonal constraints of the form $\pm x \pm y \leq k$ where $x, y \in \textit{Var}$ and $k \in \mathbb{Z} \cup \{+\infty\}$. For an octagon $o \in \hat{\mathbb{S}}_{\mathbb{O}}$, $o_{xy} = k$ denotes an octagonal constraint $y - x \leq k$. [1] The abstraction is characterized by the following abstraction function $\alpha_{\mathbb{O}}$:

$$\begin{array}{rcll} \alpha_{\mathbb{O}} & : & \wp(\mathbb{S}) \to \hat{\mathbb{S}}_{\mathbb{O}} & \\ \alpha_{\mathbb{O}}(S) & = & \perp_{\hat{\mathbb{S}}_{\mathbb{O}}} & \text{if } S = \emptyset \\ \big(\alpha_{\mathbb{O}}(S)\big)_{xy} & = & \max\{s(y) - s(x) \mid s \in S\} & \text{o.w} \end{array}$$

The abstract semantics is a fixpoint table $[\![\hat{P}]\!]^{\mathbb{O}} \in \mathbb{C} \to \hat{\mathbb{S}}_{\mathbb{O}}$ that maps each program point to a single octagon. The map is defined by the least fixpoint of the following function:

$$\begin{array}{c} F_{\mathbb{O}} : (\mathbb{C} \to \hat{\mathbb{S}}_{\mathbb{O}}) \to (\mathbb{C} \to \hat{\mathbb{S}}_{\mathbb{O}}) \\ F_{\mathbb{O}}([\![\hat{P}]\!]) = \lambda c.\hat{f}_{\mathbb{O}}\ c\ (\bigsqcup_{c' \hookrightarrow c} [\![\hat{P}]\!](c')) \end{array}$$

where $\hat{f}_{\mathbb{O}}$ functions as the standard octagon transfer function for the abstract assignment or the abstract test [45] according to an associated command.

As the octagon domain also has infinite height, we apply widening at loop headers in the program as in the interval analysis.

[1] For brevity, we only consider octagonal constraints of the following form: $x - y \leq k$.

2.2.3 Pointer Analysis

Now we describe a flow-sensitive and context-insensitive pointer analysis.

Program Commands Each command in a node (or program point) $c \in \mathbb{C}$ in the graph has one of the following command, denoted $\texttt{cmd}(c)$:

$$\text{command} \quad c \quad \rightarrow \quad \texttt{x} := \&\texttt{y} \mid \texttt{x} := \texttt{y} \mid *\texttt{x} := \texttt{y} \mid \texttt{x} := *\texttt{y}$$

Collecting Semantics We use the the final program point partitioning function described in § 3.2. The concrete domain of states $\mathbb{S}$ is the set of finite maps from variables (Var) to set of variables ($\wp(Var)$).

Abstract Semantics The abstract state is a map from program variables to its points-to set, i.e., $\hat{\mathbb{S}}_{\mathbb{P}} = Var \overset{\text{fin}}{\rightarrow} \wp(Var)$.

For each node, we define a transfer function $\hat{f}_{\mathbb{P}} : \mathbb{C} \rightarrow \hat{\mathbb{S}}_{\mathbb{P}} \rightarrow \hat{\mathbb{S}}_{\mathbb{P}}$ that, given an input memory state, computes the effect of the assignment in the node on the input state:

$$\hat{f}_{\mathbb{P}}\ c\ \hat{m} = \begin{cases} \hat{m}[x \mapsto \{y\}] & (\texttt{cmd}(c) = \texttt{x} := \&\texttt{y}) \\ \hat{m}[x \mapsto \hat{m}(y)] & (\texttt{cmd}(c) = \texttt{x} := \texttt{y}) \\ \hat{m}[x \mapsto \bigcup_{l \in \hat{m}(y)} \hat{m}(l)] & (\texttt{cmd}(c) = \texttt{x} := *\texttt{y}) \\ \hat{m}[l_1 \mapsto \hat{m}(l_1) \cup \hat{m}(y)] \cdots [l_n \mapsto \hat{m}(l_n) \cup \hat{m}(y)] & (\texttt{cmd}(c) = *\texttt{x} := \texttt{y}, \hat{m}(\texttt{x}) = \{\texttt{l}_1, \ldots, \texttt{l}_n\}) \end{cases}$$

The analyzer computes a fixpoint table $[\![\hat{P}]\!]^{\mathbb{P}} \in \mathbb{C} \rightarrow \hat{\mathbb{S}}_{\mathbb{P}}$ that maps each node in the program to its output abstract memory state. The abstract memory state at each program point approximates all the concrete memory states occurring at the node in the concrete executions. The map is defined by the least fixpoint of the following function:

$$F_{\mathbb{P}} : (\mathbb{C} \rightarrow \hat{\mathbb{S}}_{\mathbb{P}}) \rightarrow (\mathbb{C} \rightarrow \hat{\mathbb{S}}_{\mathbb{P}})$$
$$F_{\mathbb{P}}([\![\hat{P}]\!]) = \lambda c. \hat{f}_{\mathbb{P}}\ c\ (\textstyle\bigsqcup_{c' \hookrightarrow c} [\![\hat{P}]\!](c'))$$

Chapter 3

Method 1. Sound Non-statistical Alarm Clustering

3.1 Introduction

3.1.1 Problem

False alarms are the main obstacle to the wide adoption of sound static analysis tools that aim to prove safety properties about programs. Users of sound static analyzers suffer from a large number of false alarms, where false alarms often outnumber real errors. For instance, in a case of analyzing commercial software, we have found only one real error in 273 buffer-overflow alarms, after tedious and time-consuming alarm investigation efforts [31].

Statistical ranking schemes [31, 35, 36] have been proposed to find real errors quickly, but they do not fundamentally reduce alarm-investigation burdens especially in software verification. The ranking schemes alleviate the false alarm problem by showing alarms that are most likely to be real errors over those that are least likely. However, these ranking schemes cannot completely dismiss unlikely alarms. For example, we still need to examine all alarms to find the real ones in safety-critical softwares.

3.1.2 Our Solution

Our solution is to reduce alarm-investigation burden by clustering alarms according to their sound dependence information. We say that alarm A has (sound) dependence on alarm B if alarm B turns out to be false, then so does alarm A as a logical consequence. When we find a set of alarms depending on the same alarm, which we call a dominant alarm, we can cluster them together. Once we find clusters of alarms, we only need to check whether their dominant alarms are false.

In this paper, we present a sound alarm-clustering method for static analyzers. Our analysis automatically discovers sound dependencies among alarms. Combining such dependencies, our analysis finds clusters of alarms which have their own dominant alarms. If the dominant alarms turn out to be false (true resp.), we can assure that all the others in the same cluster are also false (true resp.).

3.1.3 Examples

Example 1 through 3 show examples of alarm dependencies and how they reduce alarm-investigation efforts. These examples are discovered automatically by our clustering algorithm.

Example 3.1.1 (*Beginning Example*). Our analyzer reports 5 buffer-overflow alarms for the following code excerpted from Nlkain-1.3 (alarms are underlined, and dominant alarms are double-underlined).

```
void residual(SYSTEM *sys, double *upad, double *r) {
  nx = 50;
  u = &upad[nx+2];
  ...
  for (k = 0; k < ny; k++) {
    u++;
    for(j = 0; j < nx; j++) {
      r[0] = ac[0]*u[0] - ax[0]*u[-1] - ax[1]*u[1] - ay[0]*u[-nx-2]
        - ay[nx]*u[nx+2] - q[0];
      r++; u++; q++; ac++; ax++; ay++;
    }
12      u++; ax++;
13    }
14  }
```

Note the following two facts in this example:

1. If the buffer access `u[-nx-2]` at line 8 overflows the buffer, so do the others since `-nx-2` is the lowest index among the indices of all the buffer accesses on `u`.
2. If the buffer access `u[nx+2]` at line 9 does not overflow the buffer, neither do the others since `nx+2` is the highest index among the indices of all the buffer accesses on `u`.

Using these two facts, we can cluster alarms in the following way: we can find a false alarm cluster which consists of all the alarms in the example and the dominant alarm is the one of the buffer access `u[nx+2]` at line 9. We can also find a true alarm cluster in the same way; the buffer access `u[-nx-2]` at line 8 is the dominant alarm of the cluster. Thus, in order to check the program's buffer-overrun safety, it is sufficient to show the safety of the single buffer access `u[nx+2]`, instead of doing that for all the reporetd alarms. On the other hand, finding the access `u[-nx-2]` unsafe will help to spot other potential vulnerabilities accordingly. □

Example 3.1.2 (*Inter-procedural alarm dependencies*). The following code excerpted from Appcontour-1.1.0 shows inter-procedural alarm dependencies. Our method finds dependencies among the three alarms at line 3, 4, and 10. In the example, array `invmergerules` and `invmergerulesnn` have the same size 8.

```
1  int lookup_mergearcs(char *rule) {
2    ...
3    for (i = 1; invmergerules[i]; i++)
4      if (strcasecmp(rule, invmergerulesnn[i] == 0))
5        return (i);
6     ...
7   }
8   int rule_mergearcs(struct sketch *s, int rule, int rcount) {
9     if (debug)
10      printf("%s count %d", invmergerules[rule], rcount);
11    ...
12  }
13  int apply_rule(char *rule, struct sketch *sketch) {
14    ...
15    if ((code = lookup_mergearcs(rule)))
16      res = rule_mergearcs(sketch, code, rcount);
17    ...
18  }
```

Note the following two facts in this example:

1. If the alarm of the buffer access `invmergerules[i]` at line 3 is false, so are the others because
   - the buffer accesses at line 3 and 4 use the same index variable `i` and there is no update on the value between the two accesses.
   - the value of `i` at line 3 flows to the variable `rule` at line 10 through function calls and returns ($5 \rightarrow 15 \rightarrow 16 \rightarrow 10$).
2. If the buffer access `invmergerules[rule]` at line 10 overflows, so do the others in a similar reason.

We can find false and true alarm clusters in a similar manner as we did in the example 3.1.1. Instead of inspecting all of the alarms, checking either the alarm at line 10 true or the alarm at line 3 false is sufficient to decide truth/falsehood of the remaing alarms. □

Example 3.1.3 (*Multiple dominant alarms*). The following code excerpted from GNU Chess 5.0.5 shows an example of a cluster with multiple dominant alarms. Three alarms are reported at line 3, 4, and 9. The arrays `cboard` and `ephash` have the same size 64.

```
1  void MakeMove(int side, int *move) {
2    ...
3    fpiece = cboard[f];
4    tpiece = cboard[t];
5    ...
6    if (fpiece == pawn && abs(f-t) == 16) {
7      sq = (f + t) / 2;
8      ...
9      HashKey ^= ephash[sq];
10   }
11 }
```

Since `sq` is the average of `f` and `t`, if both buffer accesses at line 3 and 4 are safe, the buffer access at line 9 is also safe. In this example, we have a false cluster which have multiple dominant alarms (the alarms at line 3 and 4). □

Although all the example programs are concerned with buffer-overflow detection for C programs, all techniques and algorithms which will be described in this paper can be generalized to other languages and safety properties as well because we are based on a general model of programs and static analyses.

3.1.4 Contributions

In this paper, we make the following contributions:

- We propose a sound alarm-clustering method for static analyzers. Our framework is general and applicable to any semantics-based static analyzers. It is orthogonal to both refining approaches and statistical ranking schemes.
- We provide three concrete instance analyses of the proposed framework. We present design and implementation of our clustering method based on interval, octagon, and symbolic domains.
- We prove the effectiveness of our clustering method with a realistic static analyzer for buffer-overflow detection. On 14 open-source benchmarks, our

clustering method identified 45% of alarms to be non-dominating. This result amounts to 45% reduction in the number of investigated alarms if the other 55% turns out to be false.

3.1.5 Outline

Section 3.2 presents our alarm-clustering framework, which is concerned with how to find clusters for a given set of dominant alarms. Section 3.3 provides algorithms to find a good set of dominant alarms. Section 3.4 shows three instances of the proposed framework. Section 3.5 presents experimental results.

3.2 Alarm Clustering Framework

In this section, we describe our general framework for alarm clustering, which provides a method to find clusters for a given set of dominant alarms.

3.2.1 Static Analysis

We first define a class of static analyses that we consider in this paper. The analysis is used to prove safety properties about programs. We use the notions described in §2.1.

3.2.2 Alarm Clustering

Alarms Suppose $\Omega : \Phi \to \wp(\mathbb{S})$ specifies erroneous states at each partitioning indices (e.g. program points). The static analyzer reports an alarm at partitioning index $\varphi \in \Phi$ if the abstract semantics $[\![\hat{P}]\!]$ involves some error states, i.e.,

$$\gamma_S([\![\hat{P}]\!](\varphi)) \cap \Omega(\varphi) \neq \varnothing$$

In the rest of the paper, we assume we have at most a single alarm at a partitioning index and hence use partitioning index and alarm interchangeably; alarm φ means the one at the trace partitioning index φ.

The alarm φ is a false alarm when the static analyzer reports the alarm but the concrete semantics does not involve any error states at φ:

$$[\![P]\!]_{/\delta}(\varphi) \cap \Omega(\varphi) = \varnothing$$

Otherwise, i.e., $[\![P]\!]_{/\delta}(\varphi) \cap \Omega(\varphi) \neq \varnothing$, the alarm is true.

Alarm Dependences Our goal is to find logical dependencies between alarms. The ideal, concrete dependencies between alarms can be defined as follows. Given two alarms φ_1 and φ_2, φ_2 has a dependence on φ_1 if φ_2 is always false whenever φ_1 is false, i.e.,

$$[\![P]\!]_{/\delta}(\varphi_1) \cap \Omega(\varphi_1) = \varnothing \implies [\![P]\!]_{/\delta}(\varphi_2) \cap \Omega(\varphi_2) = \varnothing.$$

Note that the concrete dependence of φ_2 on φ_1 leads to another dependence as contraposition:

$$[\![P]\!]_{/\delta}(\varphi_2) \cap \Omega(\varphi_2) \neq \varnothing \implies [\![P]\!]_{/\delta}(\varphi_1) \cap \Omega(\varphi_1) \neq \varnothing$$

That is, if φ_2 is a true alarm, so is φ_1.

However, because it is in general impossible to find all of such concrete dependencies, our goal is to find abstract dependencies that are sound with respect to the concrete dependencies. That is, we aim to find a subset of the concrete dependencies. Our idea is to use a sound refinement by refutation; if we can kill the alarm φ_2 from the abstract semantics refined under the assumption that alarm φ_1 is false, it means that φ_2 has concrete dependence on φ_1.

We will describe a simple example that conveys the idea.

Example 3.2.1 (*Abstract alarm dependence*). Suppose that an interval domain-based analyzer reports two buffer-overflow alarms in the following code (alarms are underlined, and the values of variables in intervals are annotated in comments).

```
int foo(int* buf, int i) { // buf.size = [11, 21], i = [0, +oo]
φ1 : buf[i] = 10;
φ2 : int j = i / 2;       // j = [0, +oo]
φ3 : return buf[j];
}
```

Under the assumption that alarm φ_1 is false, `i` at φ_1 holds $[0, 20]$ after using a sound refinement by refutation. Note that we consider an underapproximation of the erroneous states at φ_1 to guarantee the soundness of the refinement. After the refinement, `j` at φ_3 holds $[0, 10]$, which does not overflow `buf`. We may conclude φ_2 has concrete dependence on φ_1. That is, if φ_1 is a false alarm, so is φ_2. Also, if φ_2 is a true alarm, so is φ_1. The soundness is guaranteed by our alarm clustering framework. $\square$

In the rest of the section, we define the notion of sound refinement by refutation and abstract alarm dependence. Then, we define alarm clustering based on the abstract alarm dependence.

Refinement by Refutation The key idea for computing the alarm dependence is to refine the original fixpoint by assuming an alarm is false, and then checks which other alarms are filtered out. Using the assumption of alarm φ being false, we can obtain a sliced abstract semantics $\llbracket\tilde{P}\rrbracket_\varphi$. Underlying abstract domain for the refinement may be different from the one used for computing a fixpoint. But we consider the same abstract domain for brevity. The definition of $\llbracket\tilde{P}\rrbracket_\varphi$ is,

$$\llbracket\tilde{P}\rrbracket_\varphi = \mathsf{fix}^{\#}\lambda Z.\llbracket\hat{P}\rrbracket_{\neg\varphi} \sqcap \hat{F}(Z)$$

where $\mathsf{fix}^{\#}$ is a fixpoint operator and $\llbracket\hat{P}\rrbracket_{\neg\varphi}$ is the same as the original fixpoint $\llbracket\hat{P}\rrbracket$ except that an underapproximation of the erroneous states at partitioning index φ ($\hat{\Omega}(\varphi) \sqsubseteq \alpha_S(\Omega(\varphi))$) is sliced out:

$$\llbracket\hat{P}\rrbracket_{\neg\varphi} = \llbracket\hat{P}\rrbracket[\varphi \mapsto \llbracket\hat{P}\rrbracket(\varphi)\ \hat{\ominus}\ \hat{\Omega}(\varphi)]$$

where $F[a \mapsto b]$ is the same as F except that it maps a to b. Abstract slice operator $\hat{\ominus}$ depends on the type of alarm and abstract domain. The $\hat{\ominus}$ operator should be a sound abstract slice operator such that

$$\alpha_S \circ \ominus \sqsubseteq \hat{\ominus} \circ \alpha_{S \times S}$$

where the operator $\ominus$ is the set difference and $\alpha_{S\times S}$ is an abstraction lifted for pairs. We require that the abstract domain $\hat{\mathbb{S}}$ comes with a meet operator ($\sqcap$) and a sound abstract slice operator ($\hat{\ominus}$).

We can extend this refinement to the case of refuting multiple alarms. Suppose that we assume that set $\overrightarrow{\varphi}$ of alarms is false. The refinement $[\![\tilde{P}]\!]_{\overrightarrow{\varphi}}$ of the fixpoint $[\![\hat{P}]\!]$ with respect to these assumptions is,

$$[\![\tilde{P}]\!]_{\overrightarrow{\varphi}} = \mathsf{fix}^{\#} \lambda Z. [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} \sqcap \hat{F}(Z)$$

where $[\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} = \bigsqcap_{\varphi_i \in \overrightarrow{\varphi}} [\![\hat{P}]\!]_{\neg\varphi_i}$.

Abstract Alarm Dependence We now define abstract alarm dependence based on the refinement by refutation. $\varphi_1 \rightsquigarrow \varphi_2$, the dependence between alarm φ_1 and φ_2, denotes that alarm φ_2 has abstract dependence on alarm φ_1.

Definition 1 ($\varphi_1 \rightsquigarrow \varphi_2$). *Given two alarms φ_1 and φ_2, φ_2 has an abstract dependence on φ_1, iff the refinement $[\![\tilde{P}]\!]_{\varphi_1}$ by refuting φ_1 kills φ_2; i.e.*

$$\varphi_1 \rightsquigarrow \varphi_2 \quad \textit{iff} \quad \gamma_S([\![\tilde{P}]\!]_{\varphi_1}(\varphi_2)) \cap \Omega(\varphi_2) = \varnothing.$$

The following lemma shows that the abstract alarm dependence is sound with respect to the concrete dependence:

Lemma 1. *Given two alarms φ_1 and φ_2, if $\varphi_1 \rightsquigarrow \varphi_2$, then φ_2 is false whenever φ_1 is false.*

Proof. Available in Appendix. $\square$

As a contraposition of Lemma 1, we also have a different sense of soundness of abstract alarm dependence.

COROLLARY 1. *Given two alarms φ_1 and φ_2, if $\varphi_1 \rightsquigarrow \varphi_2$, then alarm φ_1 is true whenever alarm φ_2 is true.*

We extend the definition and lemma of the abstract dependence for multiple alarms. The alarm dependence in Example 3.1.3 is the example of such dependencies.

DEFINITION 2 ($\overrightarrow{\varphi} \rightsquigarrow \varphi_0$). *Given set $\overrightarrow{\varphi}$ of alarms and alarm φ_0, we write $\overrightarrow{\varphi} \rightsquigarrow \varphi_0$, and say that φ_0 has abstract dependence on set $\overrightarrow{\varphi}$, iff the refinement $[\![\tilde{P}]\!]_{\overrightarrow{\varphi}}$ by refuting set $\overrightarrow{\varphi}$ of alarms satisfies*

$$\gamma_S([\![\tilde{P}]\!]_{\overrightarrow{\varphi}}(\varphi_0)) \cap \Omega(\varphi_0) = \varnothing.$$

LEMMA 2. *Given set $\overrightarrow{\varphi}$ of alarms and alarm φ_0, if $\overrightarrow{\varphi} \rightsquigarrow \varphi_0$, then alarm φ_0 is false whenever all alarms in $\overrightarrow{\varphi}$ are false.*

PROOF. Available in Appendix. □

In fact, the contraposition of Lemma 2 is not quite useful since it specifies only some alarms among set $\overrightarrow{\varphi}$ of alarms are true when alarm φ_0 is true.

Alarm Cluster Using abstract alarm dependencies, we can build false and true-alarm clusters. Given a set of dominant alarms $\overrightarrow{\varphi}$, the false-alarm cluster is defined as follows:

DEFINITION 3 (FALSE-ALARM CLUSTER). *Let $\mathcal{A}$ be set of all alarms in program P and $\rightsquigarrow$ be the abstract dependence relation. A false-alarm cluster $\mathcal{C}^F_{\overrightarrow{\varphi}} \subseteq \mathcal{A}$ with its dominant alarms $\overrightarrow{\varphi}$ is $\{\varphi' \in \mathcal{A} \mid \overrightarrow{\varphi} \rightsquigarrow \varphi'\}$.*

The soundness of alarm cluster is directly implied by the soundness of abstract alarm dependence.

THEOREM 1. *Every alarm in $\mathcal{C}^F_{\overrightarrow{\varphi}}$ is false whenever all alarms in $\overrightarrow{\varphi}$ are false.*

PROOF. Immediate from Lemma 2. □

Now we define the true-alarm cluster as follows:

DEFINITION 4 (TRUE-ALARM CLUSTER). *Let $\mathcal{A}$ be set of all alarms in program P and $\rightsquigarrow$ be the abstract dependence relation. A true-alarm cluster $\mathcal{C}^T_{\varphi} \subseteq \mathcal{A}$ with its dominant alarms φ is $\{\varphi' \in \mathcal{A} \mid \varphi' \rightsquigarrow \varphi\}$*

Note that true-alarm clusters are only derived from a single alarm dependence such as $\varphi' \rightsquigarrow \varphi$. Multiple dependencies, such as $\overrightarrow{\varphi_0} \rightsquigarrow \varphi$, are not useful to construct true alarm clusters because the dependencies just mean that one of the alarms in $\overrightarrow{\varphi_0}$ is true then the dominant alarm is true. This judgement does not tell us exactly which alarms among set $\overrightarrow{\varphi_0}$ are true. For this reason, we only consider single alarm dependencies.

Given a dominant alarm φ, the soundness of a true-alarm cluster are defined as follows:

THEOREM 2. *Every alarm in $\mathcal{C}^T_{\varphi}$ is true whenever alarm φ is true.*

PROOF. Immediate from Corollary 1. □

From this point, we only focus on false-alarm clusters for two reasons. First, both type of clusters can be found from the same dependence relation, so whether to make true or false alarm is simply the matter of interpretation. Second, true-alarm clusters can exploit fewer dependencies than false-alarm cluster, thus they cluster less alarms. In the rest of the paper, a cluster $\mathcal{C}_{\overrightarrow{\varphi}}$ means a false-alarm cluster $\mathcal{C}^F_{\overrightarrow{\varphi}}$.

3.3 Alarm-Clustering Algorithms

In this section, we show how to find the set of dominant alarms ($\overrightarrow{\varphi}$). The alarm-clustering framework ensures that, given a set of dominant alarms $\overrightarrow{\varphi}$, the refutation method produces sound alarm clusters (Theorem 1 and 2). However, how to find a good set of dominant alarms is absent in the framework. This section presents algorithms to choose the dominant alarms.

Suppose that we are given a set $\mathcal{A}$ of alarms reported by a static analyzer. We can partition $\mathcal{A}$ into two disjoint sets, groupable($\mathcal{G}$) and ungroupable($\mathcal{U}$) alarms:

$$\mathcal{A} = \mathcal{G} \uplus \mathcal{U}.$$

We say an alarm φ' is groupable if φ' can be clustered by some dominant alarms ($\overrightarrow{\varphi}$):

$$\mathcal{G} = \{\varphi' \in \mathcal{A} \mid \exists \overrightarrow{\varphi} \subseteq \mathcal{A}.\ \varphi' \in \mathcal{C}_{\overrightarrow{\varphi}}\}$$

and the ungroupable alarms are those that cannot be clustered by our method no matter how the dominant alarms are chosen:

$$\mathcal{U} = \{\varphi' \in \mathcal{A} \mid \forall \overrightarrow{\varphi} \subseteq \mathcal{A}.\ \varphi' \notin \mathcal{C}_{\overrightarrow{\varphi}}\}.$$

Ungroupable alarms exist because the power of the underlying abstract domain of the clustering analysis is not sufficient to detect alarm dependences for them. Consider the following example.

Example 3.3.1 (*Groupable and ungroupable alarms*).

```
1  // foo.size = [10, 10] and i = [0, +oo]
2  foo[i] = 0;
3  x = foo[i];
4
5  // bar.size = [10, +oo] and j = [0, +oo]
6  bar[j] = 0;
```

Alarms at line 2 and 3 are groupable because they are dominated by the alarm at line 2. The refinement by refuting the first alarm yields abstract state $\{$`foo.size` $\mapsto [10, 10],$ `i` $\mapsto [0, 9]\}$. On the other hand, alarm at line 6 is ungroupable since the alarm is not dominated even by itself. It is because we cannot soundly refute the alarm using the interval domain. Instead, the alarm may be groupable if the alarm clustering process uses richer domain such as the octagon that can express linear inequalities (`i < foo.size`). □

Given dominant alarms $\overrightarrow{\varphi}$, the final alarm reports that users have to examine is as follows:

$$\overrightarrow{\varphi} \cup (\mathcal{G} \setminus \mathcal{C}_{\overrightarrow{\varphi}}) \cup \mathcal{U} \tag{3.1}$$

Instead of inspecting all of the groupable alarms $\mathcal{G}$, our technique allows the users to inspect only the dominant alarms, plus potentially unclustered ones $(\mathcal{G} \setminus \mathcal{C}_{\overrightarrow{\varphi}})$.

We present two algorithms, which have different trade-offs between the cost and the number of final alarm reports. The first algorithm, presented in Section 3.3.1, guarantees to find a set of *minimal* dominant alarms: the set dominates all groupable alarms and does not contain unnecessaries. However, the algorithm's running time is proportional to the number of alarms to cluster. On the other hand, the algorithm in Section 3.3.2 quickly finds a dominant alarm set regardless of the number of alarms. Instead, the set found is not guaranteed to be minimal.

3.3.1 Algorithm 1: Finding Minimal Dominant Alarms

The first algorithm finds minimal dominant alarms so that minimize the number of final alarms (3.1) for users to inspect. The set of minimal dominant alarms is defined as follows:

Definition 5 (Minimal Dominant Alarms). *Given a set of alarms $\mathcal{A}$ and groupable alarms $\mathcal{G} \subseteq \mathcal{A}$, we say $\overrightarrow{\varphi}$ is a minimal set of dominant alarms if*

1. *$\overrightarrow{\varphi}$ clusters all groupable alarms, i.e., $\mathcal{C}_{\overrightarrow{\varphi}} = \mathcal{G}$, and*
2. *$\overrightarrow{\varphi}$ is a minimal such set, i.e., $\forall \overrightarrow{\varphi}' \subseteq \mathcal{A}.\ \mathcal{C}_{\overrightarrow{\varphi}'} = \mathcal{G} \ \wedge \ \overrightarrow{\varphi} \subseteq \overrightarrow{\varphi}' \implies \overrightarrow{\varphi} = \overrightarrow{\varphi}'$*

After finding such a set of minimal dominant alarms $\overrightarrow{\varphi}$, the final alarm reports for users to inspect is $\overrightarrow{\varphi} \cup \mathcal{U}$.

Basic Algorithm We utilize existing algorithms that are initially developed for finding minimal abstractions [41]. They proposed algorithm ScanCoarsen and

ActiveCoarsen to find a program abstraction that are minimal yet sufficient to prove target queries. We adapt their idea to the problem of finding a minimal set of dominant alarms. Below, we explain our adaptation of the algorithms.

Let $\mathbf{F} : \wp(\mathcal{A}) \to \{0, 1\}$ be the clustering analysis defined as follows:

$$\mathbf{F}(\overrightarrow{\varphi}) = (\mathcal{C}_{\overrightarrow{\varphi}} = \mathcal{G})$$

which gives 1 if the false alarm cluster (Definition 3) with the dominant alarms $\overrightarrow{\varphi}$ is equivalent to the set of groupable alarms, and 0 otherwise. The following lemma and corollary show that $\mathbf{F}$ is monotone, which is a requirement of the algorithms in [41]:

Lemma 3. $\overrightarrow{\varphi} \subseteq \overrightarrow{\varphi}' \implies \mathcal{C}_{\overrightarrow{\varphi}} \subseteq \mathcal{C}_{\overrightarrow{\varphi}'}$

Proof. Available in Appendix. $\square$

Corollary 2. $\overrightarrow{\varphi} \subseteq \overrightarrow{\varphi}' \implies \mathbf{F}(\overrightarrow{\varphi}) \leq \mathbf{F}(\overrightarrow{\varphi}')$.

Our goal is to find a minimal $\overrightarrow{\varphi}$ such that $\mathbf{F}(\overrightarrow{\varphi}) = 1$. We first need to partition $\mathcal{A}$ into groupable and ungroupable alarms. The following corollary provides an algorithm to find out ungroupable alarms:

Corollary 3. $\mathcal{U} = \{\varphi \in \mathcal{A} \mid \varphi \notin \mathcal{C}_{\mathcal{A}}\}$

The Corollary 3 means that alarm φ is ungroupable if we cannot cluster it using the entire set of alarms ($\mathcal{A}$) as dominant alarms. Thus, we can find $\mathcal{U}$ by computing $\mathcal{C}_{\mathcal{A}}$. The groupable alarms are computed simply by $\mathcal{G} = \mathcal{A} \backslash \mathcal{U}$. This method is given in Algorithm 1.

Algorithm 2 presents ScanCluster that finds a minimal set of dominant alarms. The invariant of the algorithm is that L contains alarms that are necessary to cluster all the groupable alarms and U is an over-approximation of the minimal set to find. The algorithm starts with ScanCluster($\emptyset$, $\mathcal{A}$). We repeatedly remove an alarm φ from $U \setminus L$ if φ is unnecessary to cluster all groupable alarms (line 5). If the current dominant alarms no longer cluster all the groupable alarms, we put

Algorithm 1 Algorithm for finding groupable and ungroupable alarms.

```
1: procedure CATEGORIZE($[\![\hat{P}]\!]$, $\mathcal{A}$)
2:     $\langle \mathcal{U}, \mathcal{G} \rangle := \langle \emptyset, \emptyset \rangle$        ▷ ungroupable and groupable alarms
3:     for all $c \in \mathcal{A}$ do
4:         if $\gamma_S([\![\tilde{P}]\!]_{\mathcal{A}}(c)) \cap \Omega(c) \neq \varnothing$ then
5:             $\mathcal{U} := \mathcal{U} \cup \{c\}$
6:         end if
7:     end for
8:     $\mathcal{G} := \mathcal{A} - \mathcal{U}$
9:     return $\langle \mathcal{U}, \mathcal{G} \rangle$
10: end procedure
```

Algorithm 2 Clustering via Scanning

```
1: procedure SCANCLUSTER($L$, $U$)
2:     if $L = U$ then return $U$
3:     end if
4:     choose $\varphi \in U \setminus L$
5:     if $\mathbf{F}(U \setminus \{\varphi\}) = 1$ then        ▷ try removing $\varphi$
6:         return SCANCLUSTER($L, U - \{\varphi\}$)        ▷ $\varphi$ is not necessary
7:     else
8:         return SCANCLUSTER($L \cup \{\varphi\}, U$)        ▷ $\varphi$ is necessary
9:     end if
10: end procedure
```

φ' back to the dominant alarm set (line 7). The algorithm requires $|\mathcal{A}|$ calls to $\mathbf{F}$ and the following theorem shows the correctness of the algorithm.

THEOREM 3. *The algorithm* SCANCLUSTER($\emptyset, \mathcal{A}$) *returns a minimal set of dominant alarms.*

PROOF. Similar to the proof of Theorem 1 in [41]. □

Another method is applying randomization into SCANCLUSTER by using ACTIVECOARSEN in [41]. The key idea behind the algorithm is to remove random multiple alarms each iteration, as opposed to ScanCoarsen that removes a single alarm at a time. Thus, we may need less iterations.

But it is effective only if a small subset of alarms matters for clustering all groupable alarms. In other words, minimal dominat alarms should be sparse. In ActiveCoarsen, the expected number of calls to $\mathbf{F}$ is $O(s \log |\mathcal{A}|)$ where s is the size of the largest minimal set of dominant alarms. If minimal dominant alarms are dense, the number of calls becomes close to $O(|\mathcal{A}| \log |\mathcal{A}|)$, which is greater than $|\mathcal{A}|$ calls to $\mathbf{F}$ in ScanCluster. For this reason, on 14 benchmark programs in Section 3.5, ActiveCoarsen is several times slower than ScanCluster.

Further Optimization We further improve ScanCluster by considering only *refutable* alarms candidates of dominant alarms. Let R be the set of refutable alarms:

$$R = \{\varphi \in \mathcal{A} \mid \hat{T} \mathbin{\hat{\ominus}} \hat{\Omega}(\varphi) \sqsubset \hat{T}(\varphi)\}$$

We say an alarm φ is refutable if some erroneous states at φ can be sliced out in the underlying abstract domain. A refutable alarm cannot dominate other alarms. We exclude alarms not refutable from the initial set of alarms ($\mathcal{A}$) in running ScanCluster. That is, we run ScanCluster($\emptyset$, $\mathcal{A} \setminus R$) instead of ScanCluster($\emptyset$, $\mathcal{A}$). Note that refutable alarms are independent from the dichotomy between groupable and ungroupable alarms; both groupable and ungroupable alarms may contain refutable alarms. For instance, alarm φ_1 in Example 3.2.1 is ungroupable and refutable. The following lemma shows that we can safely exclude alarms not refutable in searching for minimal dominant alarms.

Lemma 4. *If an alarm φ is not refutable (i.e., $\hat{T} \mathbin{\hat{\ominus}} \hat{\Omega}(\varphi) = \hat{T}(\varphi)$), φ is not included in any set of minimal dominant alarms.*

Proof. Suppose a dominant alarm set $\overrightarrow{\varphi}$ clusters all groupable alarms, i.e., $\mathcal{C}_{\overrightarrow{\varphi}} = \mathcal{G}$, and $\varphi \in \overrightarrow{\varphi}$. Let $\overrightarrow{\varphi}' = \overrightarrow{\varphi} \setminus \{\varphi\}$. Then, $[\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} = [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}'}$ ($\because \hat{T} \mathbin{\hat{\ominus}} \hat{\Omega}(\varphi) = \hat{T}(\varphi)$). Therefore, $[\![\tilde{P}]\!]_{\overrightarrow{\varphi}} = [\![\tilde{P}]\!]_{\overrightarrow{\varphi}'}$ and $\mathcal{C}_{\overrightarrow{\varphi}} = \mathcal{C}_{\overrightarrow{\varphi}'}$, which menas $\overrightarrow{\varphi}'$ is not minimal. To conclude, φ is not included in any set of minimal dominant alarms. $\square$

In our experiment, we have observed a significant performance boost by considering refutable alarms only. In 14 benchmark programs, 32% of total alarms were not refutable. Thus, ScanCluster algorithm becomes approximately 1.5x (1/0.68) faster than non-optimized.

3.3.2 Algorithm 2: Non-Minimal but Efficient

In this section, we present a more appropriate clustering algorithm in case we have limited time budgets. The algorithm finds a subset of all abstract alarm dependences by a single fixpoint computation. The idea is to refine the analysis result as much as possible by refuting all alarms and track which dominant alarm candidate possibly kills which alarm. Then, we cluster the alarms which must be killed by the same dominant alarm candidate. Algorithm 3 describes our method that clusters alarms based on a (not all) subset of possible dependencies.

We first describe the setting which the algorithm is based on. We assume that a program is represented by a control-flow graph. Φ is the set of nodes (or program points) and every node has several predecessors and successors specified by function pred and succ (line 2). The analyzer computes a fixpoint table $[\![\hat{P}]\!] \in \Phi \to \hat{\mathbb{S}}$ that maps each node in the program to its output abstract memory state. The map is defined by the least fixpoint of the following function:

$$\hat{F} : (\Phi \to \hat{\mathbb{S}}) \to (\Phi \to \hat{\mathbb{S}})$$
$$\hat{F}([\![\hat{P}]\!]) = \lambda\varphi.\hat{f}(\varphi)(\textstyle\bigsqcup_{p \in \mathsf{predof}(\varphi)} [\![\hat{P}]\!](p))$$

where $\hat{f}(\varphi)$ is an abstract transfer function at node φ. For brevity, we also assume that an alarm can be raised at every program point; i.e. for all $\varphi \in \Phi$, $\hat{\Omega}(\varphi) \neq \bot$ where $\hat{\Omega}$ is abstract erroneous information such that ($\hat{\Omega} \sqsubseteq \alpha_S \circ \Omega$) (line 8).

Our algorithm works in the following way:

- We start by assuming that each alarm is a dominant alarm of a cluster including only itself. This can be expressed by slicing out the erroneous states at every alarm point but not propagating refinement yet.

- From an alarm point, say φ_1, we start building its cluster. We propagate its sliced, non-erroneous abstract state to another alarm point say φ_2 and see if the propagation further refines the non-erroneous abstract state at φ_2.

- If the propagated state is smaller than that at φ_2, it means refuting φ_1 will refute alarm φ_2, hence dependence $\varphi_1 \rightsquigarrow \varphi_2$ and thus we add φ_2 to the φ_1-dominating cluster.

- If the propagated state is larger than that at φ_2, then dependence $\varphi_1 \rightsquigarrow \varphi_2$ is not certain hence, instead of adding φ_2 to the φ_1-dominating cluster, we start building the φ_2-dominating cluster.

- If the propagated state is incomparable to that at φ_2, then we pick both alarms as dominant ones and start building the φ_1-and-φ_2-dominating cluster by propagating the slicing effect of simultaneously refuting (i.e., taking the meet of refuting) both alarms.

From line 1 to 9, we give definitions used in the algorithm. Everything other than function R at line 7 is trivially explained by the comment on the same line. Function R keeps the information of dominant alarm candidate. As specified in the comment, if $R(\varphi) = \overrightarrow{\varphi}$ for some program point φ and set $\overrightarrow{\varphi}$ of dominant alarms, it means that the abstract state at φ is refined by some dominant alarm candidate $\overrightarrow{\varphi}$, thus alarm φ can be a member of the $\overrightarrow{\varphi}$-dominating cluster. Line 31 shows that function R initially maps each program point φ to a set that only contains itself, which means that initially, alarm φ is the only member of the φ-dominating cluster.

Without considering gray-boxed parts, procedure FixpointIterate in the algorithm is a traditional fixpoint iteration to compute a pre-fixpoint of a decreasing chain. We pick a work from worklist (line 12), compute a new abstract state (line 14 and 15), and propagate the change to successors if the newly computed state is strictly less than the previous one (line 22). We repeat this until no work remains. We start the fixpoint computation from the one obtained by refuting all alarms (line 30).

Gray-boxed parts from line 19 to line 21 show how the algorithm tracks which dominant alarm candidates yield the refined abstract state $\hat{s}_{new}$ computed from the new abstract state $\hat{s}'$ and the previous one $\hat{s}$ at line 15. If $\hat{s}'$ is smaller than $\hat{s}$ (line 19), $\hat{s}_{new}$ is the same as $\hat{s}'$ and thus $\overrightarrow{\varphi}'$ is its dominant alarm candidates. The algorithm similarly handles the case when $\hat{s}$ is smaller than or equals to $\hat{s}'$ (line 20). If $\hat{s}$ and $\hat{s}'$ are incomparable (line 21), the meet of the two corresponds to the abstract state refined by refuting their dominant alarm candidates at the same time. Therefore, the resulting dominant alarm candidates $\overrightarrow{\varphi}_{new}$ takes the union of $\overrightarrow{\varphi}$ and $\overrightarrow{\varphi}'$.

As the last step of the clustering algorithm, procedure ClusterAlarms validates the dominant alarm candidates in R based on the refined fixpoint T and clusters alarms. For each alarm at φ, we validate that the dominant alarm candidates $R(\varphi)$ really dominates alarm φ by checking that the refined abstract state $T(\varphi)$ kills the alarm (line 27). If the alarm is killed, we put alarm φ to the $R(\varphi)$-dominating cluster (line 28 and 29).

The following theorem guarantees the correctness of the algorithm.

Theorem 4. *Algorithm 3 computes sound alarm dependences.*

Proof. Available in Appendix. □

3.4 Instances

In this section, we show how to use our framework to design alarm clustersring methods. We provide three instances based on the interval, octagon, and symbolic domains. All of the methods are implemented on top a realistic buffer-overflow analyzer for C programs [2]. The key component we have to define to use our framework is the abstract slice operator described in Section 3.2.

We begin with a simple yet general definition of sound abstract slice operators. Assume that $\hat{\mathbb{S}}$ is the underlying abstract domain used in our clustering method, which has a Galois connection $\wp(\mathbb{S}) \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \hat{\mathbb{S}}$ with concrete domain $\mathbb{S}$. An element

y in the domain $\hat{\mathbb{S}}$ is called *precisely complementable* [19] if there is a *precise complement* $\overline{y}$, a complement of y (i.e., $y \sqcap \overline{y} = \bot_{\hat{\mathbb{S}}}$ and $y \sqcup \overline{y} = \top_{\hat{\mathbb{S}}}$) satisfying

$$\gamma_S(y) = \wp(\mathbb{S}) \setminus \gamma_S(\overline{y}).$$

Using the notion of precise complements, we define the following simple but general abstract slice operator in $\hat{\mathbb{S}}$.

Definition 6 (Abstract slice operator). *Let $\hat{\mathbb{S}}$ be an abstract domain defined by the Galois connection $\wp(\mathbb{S}) \xleftrightarrow[\alpha_S]{\gamma_S} \hat{\mathbb{S}}$. For $x, y \in \hat{\mathbb{S}}$, $x \ominus_{\hat{\mathbb{S}}} y$ is defined as follows:*

$$x \ominus_{\hat{\mathbb{S}}} y = \begin{cases} x \sqcap \overline{y} & \text{if } y \text{ is precisely complementable} \\ x & \text{otherwise} \end{cases}$$

where $\overline{y}$ is a precise complement of y.

In a powerset domain, every element is precisely complementable. Thus the operator is the same as the set difference operator. Because we simply give up slicing if y is not precisely complementable, the operator is a simple abstraction of the set difference. The following theorem guarantees that the abstract operator in Definition 6 is sound.

Theorem 5. *For an abstract domain $\hat{\mathbb{S}}$ with the Galois connection $\wp(\mathbb{S}) \xleftrightarrow[\alpha_S]{\gamma_S} \hat{\mathbb{S}}$, the following holds for all $x, y \in \hat{\mathbb{S}}$:*

$$\alpha_S(\gamma_S(x) \ominus \gamma_S(y)) \sqsubseteq x \ominus_{\hat{\mathbb{S}}} y$$

Proof.

$$\begin{aligned}
x \ominus_{\hat{\mathbb{S}}} y &= x \sqcap \overline{y} \;\sqsupseteq\; \alpha_S \circ \gamma_S(x) \sqcap \alpha_S \circ \gamma_S(\overline{y}) && (\alpha_S \circ \gamma_S \sqsubseteq id) \\
&\sqsupseteq \alpha_S(\gamma_S(x) \sqcap \gamma_S(\overline{y})) && (\alpha_S \text{ is monotone and by def. of glb}) \\
&= \alpha_S(\gamma_S(x) \sqcap \overline{\gamma_S(y)}) && (y \text{ is precisely complementable}) \\
&= \alpha_S(\gamma_S(x) \ominus \gamma_S(y)) && (\text{By def. of the set minus operator})
\end{aligned}$$

□

3.4.1 Setting: Baseline Analyzer

Our baseline analyzer is an interval domain-based flow-sensitive abstract interpreter [2] on which our clustering methods are implemented. The analyzer is a realistic buffer-overflow detector performing sound and inter-procedural analysis. The design has been presented in §2.2.1. Throughout this chapter, we will use Φ and $\mathbb{C}$ interchangeably because we choose $\mathbb{C}$ as the set of partitioning indices. In addition, we will use φ and c interchangeably.

Alarms We define erroneous states and alarms of the static analysis. We assume queries, triples in $Q \subseteq \Phi \times \mathit{Var} \times \mathit{Var}$, are given as input to our static analysis. A query $\langle \varphi, x, y \rangle$ represents an assertion that x should be less than y at program point φ. Given a query, the set of erroneous states is characterized by the following function:

$$\begin{array}{rcl} \Omega & : & Q \to \wp(\mathbb{S}) \\ \Omega(\varphi, x, y) & = & \{s \in \mathbb{S} \mid s(x) \geq s(y)\} \end{array}$$

For given query $\langle \varphi, x, y \rangle$, our analyzer raises an alarm $\langle \varphi, x, y \rangle$ if $\gamma_{\mathbb{I}}(\llbracket \hat{P} \rrbracket^{\mathbb{I}}(\varphi)) \cap \Omega(\varphi, x, y) \neq \varnothing$ meaning the query $\langle \varphi, x, y \rangle$ cannot be proved.

3.4.2 Clustering using Interval Domain

We describe abstract slice operator of the interval domain. Suppose we have an alarm $\langle \varphi, x, y \rangle$. Recall that the refutation of the alarm is defined as follows:

$$\llbracket \hat{P} \rrbracket^{\mathbb{I}}_{\neg\varphi} = \llbracket \hat{P} \rrbracket^{\mathbb{I}}[\varphi \mapsto \llbracket \hat{P} \rrbracket^{\mathbb{I}}(\varphi) \ \hat{\ominus}_{\hat{\mathbb{S}}_{\mathbb{I}}} \ \hat{\Omega}(\varphi, x, y)]$$

where $\hat{\Omega}(\varphi, x, y)$ is an underapproximation of the erroneous states such that $\hat{\Omega}(\varphi, x, y) \sqsubseteq \alpha_{\mathbb{S}_{\mathbb{I}}}(\Omega(\varphi, x, y))$. The reason for using an underapproximation is that the interval analysis often fails to capture relational properties of variables. The underapproximation of the erroneous states $\hat{\Omega}(\varphi, x, y)$ is defined as follows:

$$\hat{\Omega}(\varphi, x, y) = \begin{cases} \bot_{\hat{\mathbb{S}}_{\mathbb{I}}}[x \mapsto [y_{max}, +\infty], y \mapsto [-\infty, x_{min}]] & (y_{max} \geq x_{min}, \\ & y_{max} \neq +\infty, x_{min} \neq -\infty) \\ \bot_{\hat{\mathbb{S}}_{\mathbb{I}}} & \text{(otherwise)} \end{cases}$$

where $[x_{min}, x_{max}] = [\![\hat{P}]\!]^{\mathbb{I}}(\varphi)(x)$ and $[y_{min}, y_{max}] = [\![\hat{P}]\!]^{\mathbb{I}}(\varphi)(y)$. And the following is a precise complement of $\hat{\Omega}(\varphi, x, y)$.

$$\overline{\hat{\Omega}(\varphi, x, y)} = \begin{cases} \top_{\hat{\mathbb{S}}_{\mathbb{I}}}[x \mapsto [-\infty, y_{max} - 1], y \mapsto [x_{min} + 1, +\infty]] & (y_{max} \geq x_{min}, \\ & y_{max} \neq +\infty, x_{min} \neq -\infty) \\ \top_{\hat{\mathbb{S}}_{\mathbb{I}}} & \text{(otherwise)} \end{cases}$$

Example 3.4.1.

Consider the following code. The code is simply adapted from Example 3.1.3.

```
φ1 : sz := 64;
φ2 : f := unknown();
φ3 : t := unknown();
φ4 : sq := (f + t) / 2;
```

Suppose the following set of queries Q is given.

$$Q = \{\langle \varphi_2, \texttt{f}, \texttt{sz} \rangle, \langle \varphi_3, \texttt{t}, \texttt{sz} \rangle, \langle \varphi_4, \texttt{sq}, \texttt{sz} \rangle\}$$

The variable `sz` refers to the size of `cboard` and `ephash` in Example 3.1.3. We will show the steps of deriving $\{\varphi_2, \varphi_3\} \rightsquigarrow \varphi_4$.

The analysis result at φ_4 is as follows:

$$[\![\hat{P}]\!]^{\mathbb{I}}(\varphi_4) = \{\texttt{sz} \mapsto [64, 64],\ \texttt{f}, \texttt{t}, \texttt{sq} \mapsto [-\infty, \infty]\}$$

The followings are the underapproximation of the erroneous states:

$$\overline{\hat{\Omega}(\varphi_2, \mathtt{f}, \mathtt{sz})} = \top_{\hat{\mathbb{S}}_{\mathbb{I}}}[\mathtt{f} \mapsto [-\infty, 63], \mathtt{sz} \mapsto [-\infty, +\infty]]$$
$$\overline{\hat{\Omega}(\varphi_3, \mathtt{t}, \mathtt{sz})} = \top_{\hat{\mathbb{S}}_{\mathbb{I}}}[\mathtt{t} \mapsto [-\infty, 63], \mathtt{sz} \mapsto [-\infty, +\infty]]$$

The sliced abstract semantics is:

$$\begin{array}{rcl}
[\![\hat{P}]\!]^{\mathbb{I}}_{\neg\varphi_2}(\varphi_2) & = & [\![\hat{P}]\!]^{\mathbb{I}}(\varphi_2) \ \hat{\ominus}_{\hat{\mathbb{S}}_{\mathbb{I}}} \ \overline{\hat{\Omega}(\varphi_2, \mathtt{f}, \mathtt{sz})} = [\![\hat{P}]\!]^{\mathbb{I}}(\varphi_2) \sqcap \overline{\hat{\Omega}(\varphi_2, \mathtt{f}, \mathtt{sz})} \\
& = & \{\mathtt{sz} \mapsto [64, 64], \mathtt{f} \mapsto [-\infty, 63]\} \\
[\![\hat{P}]\!]^{\mathbb{I}}_{\neg\varphi_3}(\varphi_3) & = & [\![\hat{P}]\!]^{\mathbb{I}}(\varphi_3) \ \hat{\ominus}_{\hat{\mathbb{S}}_{\mathbb{I}}} \ \overline{\hat{\Omega}(\varphi_3, \mathtt{t}, \mathtt{sz})} = [\![\hat{P}]\!]^{\mathbb{I}}(\varphi_3) \sqcap \overline{\hat{\Omega}(\varphi_2, \mathtt{t}, \mathtt{sz})} \\
& = & \{\mathtt{sz} \mapsto [64, 64], \mathtt{f} \mapsto [-\infty, \infty], \mathtt{t} \mapsto [-\infty, 63]\}
\end{array}$$

By propagating the refinement, we obtain

$$[\![\tilde{P}]\!]^{\mathbb{I}}_{\{\varphi_2, \varphi_3\}}(\varphi_4) = \{\mathtt{sz} \mapsto [64, 64],\ \mathtt{f}, \mathtt{t}, \mathtt{sq} \mapsto [-\infty, 63]\}.$$

Finally, we derive $\{\varphi_2, \varphi_3\} \rightsquigarrow \varphi_4$ because $\gamma_{\mathbb{I}}([\![\tilde{P}]\!]^{\mathbb{I}}_{\{\varphi_2, \varphi_3\}}(\varphi_4)) \cap \Omega(\varphi_4, \mathtt{sq}, \mathtt{sz}) = \varnothing$. □

The soundness of the abstract slice operator is guaranteed by the following theorem:

THEOREM 6. $\forall \varphi \in \Phi. \gamma_{\mathbb{I}}([\![\hat{P}]\!]^{\mathbb{I}}(\varphi)) \ominus \Omega(\varphi, x, y) \sqsubseteq \gamma_{\mathbb{I}}([\![\hat{P}]\!]^{\mathbb{I}}(\varphi) \ominus_{\hat{\mathbb{S}}_{\mathbb{I}}} \hat{\Omega}(\varphi, x, y))$

PROOF. Available in Appendix. □

3.4.3 Clustering using Octagon Domain

Now we present another alarm clustering technique using the octagon abstract domain [45] that captures relational properties between variables. Our octagon-based clustering find abstract dependencies beyond the capability of the interval-based clustering. Octagon domain $\hat{\mathbb{S}}_{\mathbb{O}}$ represents a set of octagonal constraints of the form $\pm x \pm y \leq k$ where $x, y \in \mathit{Var}$ and $k \in \mathbb{Z} \cup \{+\infty\}$. For an octagon $o \in \hat{\mathbb{S}}_{\mathbb{O}}$,

$o_{xy} = k$ denotes an octagonal constraint $y - x \leq k$. [1] The design has been presented in §2.2.2.

For clustering with the octagon domain, we first transform the interval fixpoint table $[\![\hat{P}]\!]^{\mathbb{I}}$ into an octagon table $[\![\hat{P}]\!]^{\mathbb{O}}$ that satisfies the following:

$$\left([\![\hat{P}]\!]^{\mathbb{O}}(\varphi)\right)_{xy} = \sup\{s(x) - s(y) \mid s \in \gamma_{\mathbb{I}}([\![\hat{P}]\!]^{\mathbb{I}}(\varphi))\}$$

The refutation of an alarm $\langle \varphi, x, y \rangle$ is similarly defined.

$$[\![\hat{P}]\!]^{\mathbb{O}}_{\neg\varphi} = [\![\hat{P}]\!]^{\mathbb{O}}[\varphi \mapsto [\![\hat{P}]\!]^{\mathbb{O}}(\varphi) \mathbin{\hat{\ominus}} \alpha_{\mathbb{O}}(\Omega(\varphi, x, y))]$$

Because the expressiveness power of octagons is good enough to represent the erroneous states, we do not have to use an underapproximation, as opposed to the interval clustering. The precise complement of the erroneous state $\alpha_{\mathbb{O}}(\Omega(\varphi, x, y))$ is defined as follows:

$$\left(\alpha_{\mathbb{O}}(\Omega(\varphi, x, y)))\right)_{ij} = \begin{cases} 0 & \text{if } i = y \text{ and } j = x \\ +\infty & \text{o.w} \end{cases}$$

The following is the precise complement of the erroneous state:

$$\left(\overline{\alpha_{\mathbb{O}}(\Omega(\varphi, x, y))}\right)_{ij} = \begin{cases} -1 & \text{if } i = x \text{ and } j = y \\ +\infty & \text{o.w} \end{cases}$$

Example 3.4.2.

Consider the following code, which has been slightly modified from Example 3.4.1.

```
φ1 : sz := unknown();
φ2 : f := unknown();
φ3 : t := unknown();
φ4 : sq := f;
```

[1]For brevity, we only consider octagonal constraints of the following form: $x - y \leq k$.

Suppose we are given the same set of queries as in Example 3.4.1.

$$Q = \{\langle \varphi_2, \texttt{f}, \texttt{sz}\rangle, \langle \varphi_3, \texttt{t}, \texttt{sz}\rangle, \langle \varphi_4, \texttt{sq}, \texttt{sz}\rangle\}$$

Because the value of `sz` is unbounded, we cannot find any dependencies with the interval domain-based clustering. But we can find $\varphi_2 \rightsquigarrow \varphi_4$ with the octagon domain.

Initial octagon table $\llbracket \hat{P} \rrbracket^{\mathbb{O}}$ is $\top_{\Phi \to \hat{\mathbb{S}}_{\mathbb{O}}}$ because all the interval values would be unbounded. The erroneous state at φ_2 is as follows:

$$\left(\overline{\alpha_{\mathbb{O}}(\Omega(\varphi_2, \texttt{f}, \texttt{sz}))}\right)_{ij} = \begin{cases} -1 & \text{if } i = \texttt{f} \text{ and } j = \texttt{sz} \\ +\infty & \text{o.w} \end{cases}$$

The sliced abstract semantics is:

$$\begin{aligned} \llbracket \hat{P} \rrbracket^{\mathbb{O}}_{\neg\varphi_2}(\varphi_2) &= \llbracket \hat{P} \rrbracket^{\mathbb{O}}(\varphi_2) \ \hat{\ominus}_{\hat{\mathbb{S}}_{\mathbb{O}}} \ \left(\overline{\alpha_{\mathbb{O}}(\Omega(\varphi_2, \texttt{f}, \texttt{sz}))}\right) = \top_{\hat{\mathbb{S}}_{\mathbb{O}}} \sqcap \left(\overline{\alpha_{\mathbb{O}}(\Omega(\varphi_2, \texttt{f}, \texttt{sz}))}\right) \\ &= \left(\overline{\alpha_{\mathbb{O}}(\Omega(\varphi_2, \texttt{f}, \texttt{sz}))}\right) \end{aligned}$$

By propagating the refinement, we obtain

$$\left(\llbracket \tilde{P} \rrbracket^{\mathbb{O}}_{\varphi_2}(\varphi_4)\right)_{ij} = \begin{cases} -1 & \text{if } i = \texttt{f} \text{ and } j = \texttt{sz} \\ -1 & \text{if } i = \texttt{sq} \text{ and } j = \texttt{sz} \\ +\infty & \text{o.w} \end{cases}$$

Finally, we derive $\varphi_2 \rightsquigarrow \varphi_4$ because $\gamma_{\mathbb{O}}(\llbracket \tilde{P} \rrbracket^{\mathbb{O}}_{\varphi_2}(\varphi_4)) \cap \Omega(\varphi_4, \texttt{sq}, \texttt{sz}) = \varnothing$. $\square$

The soundness of the abstract slice operator is guaranteed by the following theorem.

Theorem 7. $\forall \varphi \in \Phi.\ \alpha_{\mathbb{O}}(\gamma_{\mathbb{O}}(\llbracket \hat{P} \rrbracket^{\mathbb{O}}(\varphi)) \ominus \Omega(\varphi, x, y)) \sqsubseteq \llbracket \hat{P} \rrbracket^{\mathbb{O}}(\varphi) \ominus_{\hat{\mathbb{S}}_{\mathbb{O}}} \alpha_{\mathbb{O}}(\Omega(\varphi, x, y))$

Proof. By the fact that $\alpha_{\mathbb{O}}(\Omega(\varphi, x, y))$ is precisely complementable and Theorem 5, the theorem holds. $\square$

3.4.4 Clustering using Symbolic Execution

In this subsection, we present a symbolic domain–based clustering. With a reasonable cost, we perform intraprocedural symbolic execution to find abstract dependencies beyond the capability of interval and octagon-based clustering.

We use a conventional symbolic domain [32]. The set of concrete memory states are abstracted by a symbolic memory state $\hat{\mathbb{S}}_{\mathbb{SE}} = 2^{Guard \times \hat{Mem}}$, where the memory state $\hat{Mem} = \hat{Addr} \overset{\text{fin}}{\to} \hat{Val}$ is a finite map from symbolic addresses ($\hat{Addr}$) to symbolic values ($\hat{Val}$):

$$\begin{array}{rcl} \hat{Addr} & = & Var + Symbol \\ \hat{Val} & = & \mathbb{Z} + \hat{Addr} + (\hat{Val} \times \texttt{Bop} \times \hat{Val}) \\ Guard & = & Guard \wedge Guard + (\hat{Val} \times \texttt{Rel} \times \hat{Val}) + \{\text{true}, \text{false}\} \end{array}$$

A guard ($Guard$) represents a path condition under which the current program point is reachable from the function entry. `Rel` denotes a set of comparison operators (e.g., $<$). Guards may be connected by logical operators (conjuction $\wedge$). Symbols ($Symbol$) are used to indicate symbolic values. A symbolic value can be a number ($\mathbb{Z}$), or an address ($\hat{Addr}$), or a binary value ($\hat{Val} \times \texttt{Bop} \times \hat{Val}$). `Bop` denotes a set of binary operator symbols.

The partial order between two symbolic memory states $\mathcal{S}_1, \mathcal{S}_2$ are defined as follows:

$$\mathcal{S}_1 \sqsubseteq \mathcal{S}_2 \iff \forall \langle g, m\rangle \in \mathcal{S}_1.\ \exists \langle g', m'\rangle \in \mathcal{S}_2.\ (g \wedge \bigwedge_{z \in dom(m)} z = m(z)) \implies (g' \wedge \bigwedge_{z' \in dom(m')} z' = m'(z'))$$

Therefore, $\{\langle \text{true}, id\rangle\}$ is $\top_{\hat{\mathbb{S}}_{\mathbb{SE}}}$ where $id = \{l \mapsto l \mid l \in \hat{Addr}\}$.

The abstract semantics is a fixpoint table $[\![\hat{P}]\!]^{\mathbb{SE}} \in \Phi \to \hat{\mathbb{S}}_{\mathbb{SE}}$ that maps each program point to a symbolic memory state. The map is defined by the greatest fixpoint of function $F_{\mathbb{SE}}$ (i.e., $[\![\hat{P}]\!]^{\mathbb{SE}} = \bigsqcap_{i \in \mathbb{N}} F_{\mathbb{SE}}{}^{i}(\top_{\Phi \to \hat{\mathbb{S}}_{\mathbb{SE}}})$) :

$$\begin{gathered} F_{\mathbb{SE}} : (\Phi \to \hat{\mathbb{S}}_{\mathbb{SE}}) \to (\Phi \to \hat{\mathbb{S}}_{\mathbb{SE}}) \\ F_{\mathbb{SE}}([\![\hat{P}]\!]) = \lambda \varphi. \hat{f}_{\mathbb{SE}}\ \varphi\ (\textstyle\bigsqcup_{p \in \mathsf{predof}(\varphi)} [\![\hat{P}]\!](p)) \end{gathered}$$

where $\hat{f}_{\mathbb{SE}}$ is defined as follows:

$$\hat{f}_{\mathbb{SE}}\ \varphi\ \mathcal{S} = \begin{cases} \{\langle g, \hat{m}[x \mapsto [\![e]\!](\hat{m})]\rangle \mid \langle g, \hat{m}\rangle \in \mathcal{S}\} & (\texttt{cmd}(\varphi) = \texttt{x} := e) \\ \{\langle g \wedge (\texttt{x} \le \texttt{n}), \hat{m}\rangle \mid \langle g, \hat{m}\rangle \in \mathcal{S}\} & (\texttt{cmd}(\varphi) = \{\!\{\texttt{x} \le \texttt{n}\}\!\}) \\ \{\langle g, \hat{m}[x \mapsto x]\rangle \mid \langle g, \hat{m}\rangle \in \mathcal{S}\} & (\texttt{cmd}(\varphi) = \texttt{x} := \texttt{unknown()}) \end{cases}$$

and the evaluation $[\![e]\!]$ of an expression e in a memory $\hat{m}$ is defined as usual : $[\![\texttt{n}]\!](\hat{m}) = \texttt{n}$, $[\![\texttt{x}]\!](\hat{m}) = \hat{m}(\texttt{x})$, and $[\![e_1 \texttt{ + } e_2]\!](\hat{m}) = [\![e_1]\!](\hat{m}) \texttt{ + } [\![e_2]\!](\hat{m})$. We apply a simple widening operator to ensure the termination of the analysis; changing a symbolic memory state to $\top_{\hat{\mathbb{S}}_{\mathbb{SE}}}$ after some k iterations.

For clustering using symbolic execution, the interval analysis result is embedded in a program control flow graph in the form of conditional commands. In other words, we add nodes associated with `assume` commands into the control flow graph referring to the prior interval analysis result. For example, for a program point φ and a variable `x`, suppose $[\![\hat{P}]\!]^{\mathbb{I}}(\varphi)(\texttt{x}) = [-\infty, 3]$. Then we insert a node φ' such that $\texttt{cmd}(\varphi') = \{\!\{\texttt{x} \le 3\}\!\}$ between φ and all nodes in $\textsf{predof}(\varphi)$.

The refutation of an alarm $\langle \varphi, x, y\rangle$ on the fixpoint symbolic state is defined as follows:

$$[\![\hat{P}]\!]^{\mathbb{SE}}_{\neg\varphi} = [\![\hat{P}]\!]^{\mathbb{SE}}[\varphi \mapsto \{\langle g \wedge x < y, \hat{m}\rangle \mid \langle g, \hat{m}\rangle \in [\![\hat{P}]\!]^{\mathbb{SE}}(\varphi)\}]$$

After the refinement resulting in $[\![\tilde{P}]\!]^{\mathbb{SE}}_{\varphi}$, we check the validity of the following condition to determine if another alarm, namely $\langle \varphi', x', y'\rangle$, has been killed by the refutation:

$$\forall \langle g, \hat{m}\rangle \in [\![\tilde{P}]\!]^{\mathbb{SE}}_{\varphi}(\varphi').\ g \wedge (\bigwedge_{z \in dom(\hat{m})} z = \hat{m}(z)) \implies x' < y'$$

Example 3.4.3. Consider the following code (slightly modified from Example 3.4.1).

```
φ1 : sz := unknown();
φ2 : f := unknown();
φ3 : t := unknown();
φ4 : sq := (f + t) / 2;
```

Suppose we are given the same set of queries as in Example 3.4.1.

$$Q = \{\langle \varphi_2, \texttt{f}, \texttt{sz}\rangle, \langle \varphi_3, \texttt{t}, \texttt{sz}\rangle, \langle \varphi_4, \texttt{sq}, \texttt{sz}\rangle\}$$

Because the value of sz is unbounded, we cannot find any dependencies with the interval domain-based clustering. In addition, because the command at φ_4 is beyond the expressiveness power of the octagon domain, we cannot find any dependencies with the octagon domain. But we can find $\{\varphi_2, \varphi_3\} \rightsquigarrow \varphi_4$ with the symbolic domain.

The symbolic memory state at φ_4 is:

$$[\![\hat{P}]\!]^{\mathbb{SE}}(\varphi_4) = \{\langle \text{true}, id[\texttt{sq} \mapsto (\texttt{f} + \texttt{t})/2]\rangle\}$$

The refutation results of alarms φ_2 and φ_3 are as follows:

$$[\![\hat{P}]\!]^{\mathbb{SE}}_{\neg\varphi_2}(\varphi_2) = \{\langle (\texttt{f} < \texttt{sz}), id\rangle\}$$
$$[\![\hat{P}]\!]^{\mathbb{SE}}_{\neg\varphi_3}(\varphi_3) = \{\langle (\texttt{t} < \texttt{sz}), id\rangle\}$$

By propagating the refinement, we obtain

$$[\![\tilde{P}]\!]^{\mathbb{SE}}_{\{\varphi_2,\varphi_3\}}(\varphi_4) = \{\langle (\texttt{f} < \texttt{sz}) \wedge (\texttt{t} < \texttt{sz}), id[\texttt{sq} \mapsto (\texttt{f} + \texttt{t})/2]\rangle\}$$

Finally, we find $\{\varphi_2, \varphi_3\} \rightsquigarrow \varphi_4$ because the following holds:

$$(\texttt{f} < \texttt{sz}) \wedge (\texttt{t} < \texttt{sz}) \wedge (\texttt{sq} = (\texttt{f} + \texttt{t})/2) \implies \texttt{sq} < \texttt{sz}$$

□

3.5 Experiments

We apply our clustering methods on 14 packages from three different categories (Bugbench [14], GNU softwares, and SourceForge open source projects). Table 3.1 shows the benchmark programs.

Table 3.1: The overall effectiveness.

Program	**# Alarms**			**% Reduc.**		**Time(s)**		
	B	**I**	**S+I**	**I**	**+S**	**B**	**I**	**S**
nlkain-1.3	124	66	66	47%	0%	0.3	2.4	0.6
polymorph-0.4.0	21	15	14	29%	5%	0.1	0.02	0.01
ncompress-4.2.4	82	70	52	15%	22%	1.7	2.6	0.9
sbm-0.0.4	269	231	189	14%	16%	4.3	131.6	3.1
stripcc-0.2.0	190	132	110	31%	12%	3.1	5.3	0.6
barcode-0.9.6	416	355	287	15%	16%	3.3	16	3.6
129.compress	66	49	35	26%	21%	91.6	1167.2	0.2
archimedes-0.7.0	119	24	24	80%	0%	16.6	48.8	2.2
man-1.5h1	287	234	191	18%	15%	31.4	99.3	1.6
gzip-1.2.4	390	325	294	17%	8%	15.6	110.7	6.1
combine-0.3.3	836	485	318	42%	20%	21.8	586.1	123.9
gnuchess-5.05	1040	427	329	59%	9%	67.4	3842.1	41.8
bc-1.06	730	482	337	34%	20%	50.6	1943.3	24.3
grep-2.5.1	948	819	811	14%	1%	35.6	321.6	0.1
TOTAL	5518	3714	3057	33%	12%	343.4	8277.02	209.01

B : Baseline analysis, **I**: Interval domain-based clustering,
S : Symbolic execution-based clustering.

Effectiveness To evaluate how much our clustering can reduce the alarm-investigation effort, we measure the number of distinct dominant alarms after clustering and compare it to the number of original alarms reported by the basline analysis. We apply interval domain-based clustering and symbolic execution-based clustering. We do not employ octagon-based clustering because in practice, symbolic execution-based approach finds alarm dependencies that are detectable by octagon-based clustering with a cheaper cost. For instance, in our previous work [39], the octagon-based clustering reduced 8% of alarms, but our new symbolic execution-based clustering reduces 12% with a smaller cost. We use ScanCluster algorithm for interval domain-based clustering and the heuristic algorithm for symbolic execution-based clustering because each of symbolic executions requires significant overhead. In Table 3.1, the column labeled "# Alarms" shows the numbers of alarms reported by the baseline analyzer (**B**), reduced by clustering using interval domain (**I**), reduced further by clustering using symbolic execution (**S+I**), respectively. The next columns labeled "% Reduc." show the reduction ratios of each additional alarm clustering analysis (**I**, **+S**). As shown in Table 3.1, our method identifies 45% of the alarms non-dominating.

We investigate the most effective and the least effective cases of the interval-based clustering. Our interval domain-based algorithm turned out to be the most effective for archimedes-0.7.0 and gnuchess-5.05 (reduced by 80% and 59%) because of the following reasons. First, the sizes of almost all buffers in the programs are fixed. In this case, we can slice out erroneous state accurately, which is essential for the refinement by refutation using interval domain. Second, there were many different buffers of the same size which are accessed using the same index variable. On the other hand, our interval domain-based clustering is least effective for sbm-0.0.4 and grep-2.5.1 (reduced by 14%). It is because almost all buffers in the program are dynamically allocated, thus the sizes of them were hard to accurately track. Indeed, we found that the interval values of the buffer sizes were, in most cases, $[0, \infty]$ which means the buffer can have arbitrary size. In this case, we cannot slice out the erroneous states at all.

We also investigate effective cases of the symbolic execution-based clustering. Programs ncompress-4.2.4, 129.compress, combine-0.3.3, and bc-1.06 contain many consecutive buffer accesses having relationship of form $\sum_i a_i x_i \leq c$ where each x_i is a variable and c is a constant. This type of relationship can be precisely expressed and handled by SMT solvers.

Clustering Overhead We measure the analysis time to assess the overhead of clustering analysis. All our experiments are performed on a Linux machine with a 2.8 GHz Intel Xeon processor and 24 GB of memory. In Table 3.1, the columns labeled "Time" present times for the baseline analysis (**B**) and the additional alarm clustering using interval domain (**I**) and symbolic execution (**S**).

The overhead of interval domain-based alarm clustering on average surpasses the baseline analysis time because the ScanCluster algorithm checks whether each of alarms is dominating. In spite of the significant overhead, we consider the interval-based clustering still practical because manual investigation of each alarm often takes much more than about 5 seconds, which is the amortized time for identifying a single alarm non-dominating.

On the other hand, the overhead of symbolic execution-based clustering is smaller than the baseline analysis time by employing the heuristic algorithm and avoiding inter-procedural analysis.

Comparison Between the Two Clustering Algorithms Furthermore, we investigate cost and precision of a minimal clustering and the heuristic algorithms in the interval-based clustering. As the minimal clustering algorithm, we adopt the ScanCluster algorithm.We expect the latter algorithm to be cheaper than the former in programs with more sparse dominating alarms. Table 3.2 demonstrates the comparison. The columns labeled "**H**" show the number of dominant alarms, the reduction ratios, and clustering time respectively when the heuristic algorithm is applied. The columns labeled "**M**" presents the results when the minimal clustering algorithm is applied. The heuristic algorithm finds 12% less alarms non-dominating, but about 212x faster than the minimal clustering algorithm.

Table 3.2: Comparison between the minimal and heuristic algorithms.

Program	**LOC**	**# Alarms**			**% Reduc.**		**Time(s)**		
		B	**H**	**M**	**H**	**M**	**B**	**H**	**M**
nlkain-1.3	831	124	104	66	16%	47%	0.3	0.06	2.4
polymorph-0.4.0	1357	21	16	15	24%	29%	0.1	0.01	0.02
ncompress-4.2.4	2195	82	71	70	14%	15%	1.7	0.2	2.6
sbm-0.0.4	2467	269	261	231	3%	14%	4.3	1.2	131.6
stripcc-0.2.0	2555	190	156	132	18%	31%	3.1	0.4	5.3
barcode-0.9.6	4460	416	361	355	13%	15%	3.3	0.5	16
129.compress	5585	66	58	49	12%	26%	91.6	0.4	1167.2
archimedes-0.7.0	7569	119	52	24	56%	80%	16.6	1.2	48.8
man-1.5h1	7232	287	244	234	15%	18%	31.4	4.8	99.3
gzip-1.2.4	11213	390	356	325	9%	17%	15.6	2.1	110.7
combine-0.3.3	11472	836	576	485	31%	42%	21.8	3.2	586.1
gnuchess-5.05	11629	1040	693	427	33%	59%	67.4	12.3	3842.1
bc-1.06	12830	730	640	482	12%	34%	50.6	8.9	1943.3
grep-2.5.1	31154	948	839	819	11%	14%	35.6	3.5	321.6
TOTAL	112549	5518	4438	3726	20%	32%	343.4	38.77	8277.02

B : Baseline analysis, **H**: The heuristic clustering algorithm using interval domain, **M** : The minimal clustering algorithm using interval domain

Algorithm 3 Clustering algorithm

```
1: $w \in \mathit{Work} = \Phi \quad W \in \mathit{Worklist} = 2^{\mathit{Work}}$
2: $\mathsf{pred} \in \mathit{Predecessors} = \Phi \to 2^{\Phi}$
3: $\mathsf{succ} \in \mathit{Successors} = \Phi \to 2^{\Phi}$
4: $\hat{f} \in \Phi \to \hat{\mathbb{S}} \to \hat{\mathbb{S}}$    ▷ abstract transfer function for each program point
5: $T \in \mathit{Table} = \Phi \to \hat{\mathbb{S}}$    ▷ abstract state indexed by program point
6: $\overrightarrow{\varphi} \in \mathit{DomCand} = 2^{\Phi}$    ▷ dominant alarm candidate. set of alarms.
7: $R \in \mathit{RefinedBy} = \Phi \to \mathit{DomCand}$    ▷ $\{\varphi \mapsto \overrightarrow{\varphi}\} \in R$ : $T(\varphi)$ is refined by $\overrightarrow{\varphi}$
8: $\hat{\Omega} \in \mathit{ErrorInfo} = \Phi \to \hat{\mathbb{S}}$    ▷ abstract erroneous state information
9: $\mathcal{C} \in \mathit{Clusters} = \mathit{DomCand} \to 2^{\Phi}$    ▷ alarm clusters indexed by dominant alarms
10: procedure FixpointIterate($W, T, R$)
11:     repeat
12:         $\varphi$ := choose($W$)    ▷ pick a work from worklist
13:         $\hat{s} := T(\varphi)$    ▷ previous abstract state
14:         $\hat{s}' := \hat{f}(\varphi)(\bigsqcup_{\varphi_i \in \mathsf{pred}(\varphi)} T(\varphi_i))$    ▷ new abstract state
15:         $\hat{s}_{new} := \hat{s}' \sqcap \hat{s}$
16:
17:         $\overrightarrow{\varphi} := R(\varphi)$    ▷ previous set of dominant alarm candidates
18:         $\overrightarrow{\varphi}' := \bigcup_{\varphi_i \in \mathsf{pred}(\varphi)} R(\varphi_i)$    ▷ new set of dominant alarm candidates
19:         if $\hat{s} \sqsupset \hat{s}'$ then $\overrightarrow{\varphi}_{new} = \overrightarrow{\varphi}'$
20:         else if $\hat{s} \sqsubseteq \hat{s}'$ then $\overrightarrow{\varphi}_{new} = \overrightarrow{\varphi}$
21:         else $\overrightarrow{\varphi}_{new} := \overrightarrow{\varphi} \cup \overrightarrow{\varphi}'$
22:         if $\hat{s}_{new} \sqsubset \hat{s}$ then    ▷ propagate the change to successors
23:             $W := W \cup \mathsf{succ}(\varphi)$; $T(\varphi) := \hat{s}_{new}$; $R(\varphi) := \overrightarrow{\varphi}_{new}$
24:     until W = $\varnothing$
25: procedure ClusterAlarms($T, R$)
26:     for all $\varphi \in \Phi$ do
27:         if $T(\varphi) \sqcap \hat{\Omega}(\varphi) = \bot$ then
28:             $\mathcal{C} := \mathcal{C}\{R(\varphi) \mapsto \mathcal{C}(R(\varphi)) \cup \{\varphi\}\}$
29: procedure main()
30:     $T := [\![\hat{P}]\!]_{\neg\Phi}$    ▷ $[\![\hat{P}]\!]$ is the original fixpoint
31:     $R := \{\varphi \mapsto \{\varphi\} \mid \varphi \in \Phi\}$
32:     FixpointIterate(Φ,T,R)
33:     ClusterAlarms(T,R)
```

Chapter 4

Method 2. A Progress Bar for Static Analyzers

4.1 Introduction

We aim to develop a progress bar for static analyzers. Realistic semantic-based static analyzers usually take a long time to analyze real-world software. For instance, Sparrow [2], our static analyzer for full C, takes more than 4 hours to analyze one million lines of C code [49]. ASTRÉE [3] has also been reported to take over 20 hours to analyze programs of size over 500KLOC [16]. Nonetheless, such static analyzers are silent during their operation and users cannot but wait several hours without any progress information.

Estimating static analysis progress at real-time is challenging in general. Static analyzers take most of their time in fixpoint computation, but estimating the progress of fixpoint algorithms has been unknown. One challenge is that the analysis time is generally not proportional to the size of the program to analyze. For instance, Sparrow [49] takes 4 hours in analyzing one million lines but require 10 hours to analyze programs of sizes around 400KLOC. Similar observations have been made for ASTRÉE as well: ASTRÉE takes 1.5 hours for 70KLOC but takes 40 minutes for 120KLOC [16].

In this paper, we present an idea for estimating static analysis progress. Our basic approach is to measure the progress by calculating lattice heights of intermediate analysis results and comparing them with the height of the final analysis result. To this end, we employ a semantic-based pre-analysis and a statistical regression technique. First, we use the pre-analysis to approximate the height of the fixpoint. This estimated height is then fine-tuned with the statistical method. Second, because this height progress usually does not indicate the actual progress (speed), we normalize the progress using the pre-analysis.

We show that our technique effectively estimates static analysis progress in a realistic setting. We have implemented our idea on top of Sparrow [2]. In our experiments with various open-source benchmarks, the proposed technique is found to be useful to estimate the progress of interval, octagon, and pointer analyses. The pre-analysis overheads are 3.8%, 7.3%, and 36.6% on average in interval, pointer, and octagon analysis, respectively.

Contributions This paper makes the following contributions:

- We present a technique for estimating static analysis progress. To our knowledge, our work is the first attempt to estimate static analysis progress.
- We show its applicability for numerical analyses (with intervals and octagons) and a pointer analysis on a suite of real C benchmarks.

Outline Section 4.2 describes the overall approach to our progress estimation and the remaining sections fill the details. Section 4.3 defines a class of non-relational static analyses and Section 4.4 gives the details on how we develop a progress bar for these analyses. Section 4.5 experimentally evaluates the proposed technique. Section 4.6 discusses the application to relational analyses.

4.2 Overall Approach to Progress Estimation

In this section, we describe the high-level idea of our progress estimation technique. In §4.4, we give details that we used in our experiments.

4.2.1 Static Analysis

We consider a static analysis designed by abstract interpretation described in §2.1. Throughout this chapter, we will let $\mathbb{D}$ refer to $\Phi \to \hat{\mathbb{S}}$.

4.2.2 Progress Estimation

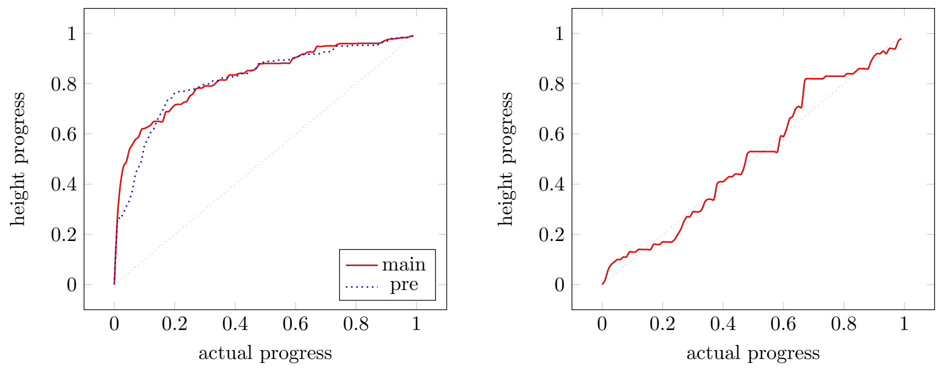
We aim to develop a progress bar that proceeds at a linear rate. That is, the estimated progress directly indicates the amount of work that has been completed so far. Suppose that the sequence in (2.1) requires n iterations to stabilize, and assume that computing the abstract semantics $\hat{F}(X)$ at each iteration takes a constant time regardless of the input X. Then, the *actual progress* of the analysis at ith iteration is defined by $\frac{i}{n}$. We aim at estimating this progress.

Basically, our method estimates the progress by calculating the lattice heights of intermediate analysis results. Suppose that we have a function $\mathsf{H} : \mathbb{D} \to \mathbb{N}$ that takes an abstract domain element $X \in \mathbb{D}$ and computes its height. The heights of domain elements need not be precisely defined, but we assume that H satisfies two conditions: 1) the height is initially zero. 2) H is monotone. The second condition is for building a progress bar that monotonically increases as the analysis makes progress.

The first job in our progress estimation is to approximate the height of the final analysis result. Let H_{final} be the height of the final analysis result, i.e., $H_{\mathit{final}} = \mathsf{H}(\bigsqcup_{i\in\mathbb{N}} \hat{F}^i(\bot))$. In §4.4.3, we describe a method for precisely estimating H_{final} with the aid of statistical regression. This height estimation method is orthogonal to the rest part of our progress estimation technique. In this overview, let $H^\sharp_{\mathit{final}}$ be the estimated final height and assume, for simplicity, that $H^\sharp_{\mathit{final}} = H_{\mathit{final}}$.

A Naive Approach Given H and $H^\sharp_{\mathit{final}}$, a simple progress bar could be developed as follows. At each iteration i, we first compute the height of the current analysis result:

$$H_i = \mathsf{H}(\hat{F}^i(\bot)).$$



(a) original height-progress (b) normalized height-progress

Figure 4.1: The height progress of a main analysis can be normalized using a pre-analysis. In this program (`sendmail-8.14.6`), the pre-analysis takes only 6.6% of the main analysis time.

Then, we show to the users the following *height progress* of the analysis :

$$P_i = \frac{H_i}{H^{\sharp}_{\mathit{final}}}$$

Note that we can use P_i as a progress estimation: P_i is initially 0, monotonically increases as the analysis makes progress, and has 1 when the analysis is completed.

Problem of the Naive Approach We noticed that this simple method for progress estimation is, however, unsatisfactory in practice. The main problem is that the height progress does not necessarily indicate the amount of computation that has been completed. For instance, the solid line in Figure 4.1(a) depicts how the height progress increases during our interval analysis of program `sendmail-8.14.6` (The dotted diagonal line represents the ideal progress bar). As the figure shows, the height progress rapidly increases during the early stage of the analysis and after that slowly converges. We found that this progress bar is not much useful to infer the actual progress nor to predict the remaining time of the analysis.

Our Approach We overcome this problem by normalizing the height progress using the relationship between the actual progress and the height progress. Suppose at the moment that we are given a function $\mathsf{normalize} : [0,1] \to [0,1]$ that maps the height progress into the corresponding actual progress. Indeed, $\mathsf{normalize}$ represents the inverse of the graph (the solid line) shown in Figure 4.1(a). Given such $\mathsf{normalize}$, the normalized height progress is defined as follows:

$$\bar{P}_i = \mathsf{normalize}(P_i) = \mathsf{normalize}\big(\frac{H_i}{H^{\sharp}_{\mathit{final}}}\big) \tag{4.1}$$

Note that, unlike the original height progress P_i, the normalized progress $\bar{P}_i$ would represent the actual progress, increasing at a linear rate. However, note also that we cannot compute $\mathsf{normalize}$ unless we run the main analysis.

The key insight of our method is that we can predict the $\mathsf{normalize}$ function by using a less precise, but cheaper pre-analysis than the main analysis. Our hypothesis is that if the pre-analysis is semantically related with the main analysis, it is likely that the pre-analysis' height-progress behavior is similar to that of the main analysis. In this article, we show that this hypothesis is experimentally true and allows to estimate sufficiently precise normalization functions.

We first design a pre-analysis as a further abstraction of the main analysis. Let $\mathbb{D}^{\sharp}$ and $F^{\sharp} : \mathbb{D}^{\sharp} \to \mathbb{D}^{\sharp}$ be such abstract domain and semantic function of the pre-analysis, respectively. In §4.4.2, we give the exact definition of the pre-analysis design we used. Next, we run this pre-analysis, computing the following sequence until stabilized:

$$\bigsqcup_{i \in \mathbb{N}} {F^{\sharp}}^{i}(\bot^{\sharp}) = {F^{\sharp}}^{0}(\bot^{\sharp}) \sqcup {F^{\sharp}}^{1}(\bot^{\sharp}) \sqcup {F^{\sharp}}^{2}(\bot^{\sharp}) \sqcup \cdots$$

Suppose that the pre-analysis stabilizes in m steps (m is often much smaller than n, the number of iterations for the main analysis to stabilize). Then, we collect the following data during the course of the pre-analysis:

$$(\frac{H_0^\sharp}{H_m^\sharp}, \frac{0}{m}), \quad (\frac{H_1^\sharp}{H_m^\sharp}, \frac{1}{m}), \quad \cdots, \quad (\frac{H_i^\sharp}{H_m^\sharp}, \frac{i}{m}), \quad \cdots, \quad (\frac{H_m^\sharp}{H_m^\sharp}, \frac{m}{m})$$

where $H_i^\sharp = \mathsf{H}(\gamma(F^{\sharp i}(\bot^\sharp)))$. The second component $\frac{i}{m}$ of each pair represents the actual progress of the pre-analysis at the ith iteration, and the first represents the corresponding height progress. Generalizing the data (using a linear interpolation method), we obtain a normalization function $\mathsf{normalize}^\sharp : [0,1] \to [0,1]$ for the pre-analysis.

The normalization function $\mathsf{normalize}^\sharp$ of such a pre-analysis can be a good estimation of the normalization function $\mathsf{normalize}$ of the main analysis. For instance, the dotted curve in Figure 4.1(a) shows the height progress of our pre-analysis (defined in §4.4.2), which has a clear resemblance with the height progress (the solid line) of the main analysis. Thanks to this similarity, it is acceptable in practice to use the normalization function $\mathsf{normalize}^\sharp$ for the pre-analysis instead of $\mathsf{normalize}$ in our progress estimation. Thus, we revise (4.1) as follows:

$$\bar{P}_i^\sharp = \mathsf{normalize}^\sharp(\frac{H_i}{H_{\mathit{final}}}) \tag{4.2}$$

That is, at each iteration i of the main analysis, we show the estimated normalized progress $\bar{P}_i^\sharp$ to the users. Figure 4.1(b) depicts $\bar{P}_i^\sharp$ for `sendmail-8.14.6` (on the assumption that $H_{\mathit{final}}^\sharp = H_{\mathit{final}}$). Note that, unlike the original progress bar (the solid line in Figure 4.1(a)), the normalized progress bar progresses at an almost linear rate.

4.3 Setting

In this section, we define a class of static analyses on top of which we develop our progress estimation technique. For presentation brevity, we consider non-relational analyses. However, our overall approach to progress estimation is also applicable to relational analyses. In §4.6, we discuss the application to a relational analysis with the octagon domain.

Programs We consider programs described in § 2.2.

Non-Relational Static Analyses We consider a class of static analyses whose abstract domain maps program points to abstract states:

$$\mathbb{D} = \mathbb{C} \to \mathbb{S}$$

where the abstract state is a map from abstract locations to abstract values:

$$\mathbb{S} = \mathbb{L} \to \mathbb{V}$$

We assume that the set of abstract locations is finite and $\mathbb{V}$ is a complete lattice. The abstract semantics of the program is characterized by the least fixpoint of abstract semantic function $F \in (\mathbb{C} \to \mathbb{D}) \to (\mathbb{C} \to \mathbb{D})$ defined as,

$$F(X) = \lambda c \in \mathbb{C}. f_c(\bigsqcup_{c' \hookrightarrow c} X(c')) \tag{4.3}$$

where $f_c \in \mathbb{D} \to \mathbb{D}$ is the transfer function for control point c.

Example 4.3.1 (*Interval Analysis*). The interval analysis described in §2.2.1 is a non-relational analysis where $\mathbb{L} = \mathit{Var}$, $\mathbb{V} = \mathbb{I}$.

Example 4.3.2 (*Pointer Analysis*). The pointer analysis described in §2.2.3 is a non-relational analysis where $\mathbb{L} = \mathit{Var}$, $\mathbb{V} = \wp(\mathit{Var})$.

4.4 Details on Our Progress Estimation

As described in §4.2, our progress estimation is done in two steps: (1) we first run a pre-analysis to obtain an estimated normalization function $\mathsf{normalize}^\sharp$ and an estimated final height $H^{\sharp}_{\mathit{final}}$; (2) using them, at each iteration of the main analysis, we measure the height progress, convert it to the estimated actual progress, and show it to users. However, §4.2 has left out a number of details. In this section, we give the details that we tried:

- In Section 4.4.1, we define our height function H.
- In Section 4.4.2, we describe our pre-analysis design.
- In Section 4.4.3, we present techniques for precise estimation of the final height.

4.4.1 The Height Function

We first define height function $\mathsf{H} : (\mathbb{C} \to \mathbb{D}) \to \mathbb{N}$ that takes an abstract domain element and computes its height. Since our analysis is non-relational, we assume that the height of an abstract domain element is computed point-wise as follows:

$$\mathsf{H}(X) = \sum_{c \in \mathbb{C}} \sum_{l \in \mathbb{L}} \mathsf{h}(X(c)(l)) \tag{4.4}$$

where $\mathsf{h} : \mathbb{V} \to \mathbb{N}$ is the height function for the abstract value domain ($\mathbb{V}$).

Example 4.4.1. For the interval domain, we use the following height function:

$$\begin{aligned} \mathsf{h}(\bot) &= 0 \\ \mathsf{h}([a,b]) &= \begin{cases} 1 & a = b \ \wedge \ a, b \in \mathbb{Z} \\ 2 & a < b \ \wedge \ a, b \in \mathbb{Z} \\ 3 & a \in \mathbb{Z} \ \wedge \ b = +\infty \\ 3 & a = -\infty \ \wedge \ b \in \mathbb{Z} \\ 4 & a = -\infty \ \wedge \ b = +\infty \end{cases} \end{aligned}$$

We defined this height function based on the actual workings of our interval analysis. Constant intervals (the first case) have height 1 since they are usually immediately generated from program texts. The finite intervals (the second case) are often introduced by joining two constant intervals. Intervals with one infinite bound (the third and fourth cases) are due to the widening operator. Note that our widening operator (Example 2.2.1) immediately assigns $\pm\infty$ to unstable bounds. $[-\infty, +\infty]$ is generated with the widening is applied to both bounds.

Example 4.4.2. For the pointer domain, we use the following height function:

$$\mathsf{h}(S) = \begin{cases} 4 & |S| \geq 4 \\ |S| & \text{otherwise} \end{cases}$$

This definition is based on our observation that, in flow-sensitive pointer analysis of C programs, most of the points-to sets have sizes less than 4.

4.4.2 Pre-analysis via Partial Flow-Sensitivity

A key component of our method is the pre-analysis that is used to estimate both the height-progress behavior and the maximum height of the main analysis. One natural method for further abstracting static analyses in §4.3 is to approximate the level of flow-sensitivity. In this subsection, we design a pre-analysis that was found to be useful in progress estimation.

We consider a class of pre-analyses that is partially flow-sensitive version of the main analysis. While the main analysis is fully flow-sensitive (i.e., the orders of program statements are fully respected), our pre-analysis only respects the orders of some selected program points and regards other program points flow-insensitively.

In particular, we are interested in a pre-analysis that only distinguishes program points around headers of flow cycles. In static analysis, the most interesting things usually happen in flow cycles. For instance, because of widening and join, significant changes in abstract states occur at flow cycle headers. Thus, it is reasonable to pay particular attention to height increases occurred at widening points ($\mathbb{W}$). To control the level of flow-sensitivity, we also distinguish some preceding points of widening points.

Formally, the set of distinguished program points is defined as follows. Suppose that a parameter *depth* is given, which indicates how many preceding points of flow cycle headers are separated in our pre-analysis. Then, we decide to distinguish the following set $\mathbb{C}_s \subseteq \mathbb{C}$ of program points:

$$\mathbb{C}_s = \{c \in \mathbb{C} \mid w \in \mathbb{W} \ \wedge \ c \hookrightarrow^{depth} w\}$$

where $c \hookrightarrow^i c'$ means that c' is reachable from c within i steps of $\hookrightarrow$.

We define the pre-analysis that is flow-sensitive only for $\mathbb{C}_s$ as a special instance of the trace partitioning [56]. The set of partitioning indicies Φ is defined by $\Phi = \mathbb{C}_s \cup \{\bullet\}$, where $\bullet$ represents all the other program points not included in $\mathbb{C}_s$. That is, we use the following partitioning function $\delta : \mathbb{C} \to \Phi$:

$$\delta(c) = \begin{cases} c & c \in \mathbb{C}_s \\ \bullet & c \notin \mathbb{C}_s \end{cases}$$

With δ, we define the abstract domain ($\mathbb{D}^\sharp$) and semantic function ($F^\sharp$) of the pre-analysis as follows:

$$\mathbb{C} \to \mathbb{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \Phi \to \mathbb{D}$$

where

$$\gamma(X) = \lambda c.\ X(\delta(c)).$$

The semantic function $F^\sharp : (\Phi \to \mathbb{D}) \to (\Phi \to \mathbb{D})$ is defined as,

$$F^\sharp(X) = \lambda i \in \Phi.\ (\bigsqcup_{c \in \delta^{-1}(i)} f_c(\bigsqcup_{c' \hookrightarrow c} X(\delta(c'))) \tag{4.5}$$

where $\delta^{-1}(i) = \{c \in \mathbb{C} \mid \delta(c) = i\}$.

Note that, in our pre-analysis, we can control the granularity of flow-sensitivity by adjusting the parameter $depth \in [0, \infty]$. A larger *depth* value yields a more precise pre-analysis. In our experiments (§4.5), we use 1 for the default value of *depth* and show that how the progress estimation quality improves with higher *depth* values. It is easy to check that our pre-analysis is sound with respect to the main analysis regardless of parameter *depth*.

4.4.3 Precise Estimation of the Final Height

The last component in our approach is to estimate H_{final}, the height value of the final analysis result. Note that H_{final} cannot be computed unless we actually run the main analysis. Instead, we compute $H^\sharp_{\mathit{final}}$, an estimation of H_{final}. We replace the H_{final} in (4.2) by $H^\sharp_{\mathit{final}}$ as follows:

$$\bar{P}^\sharp_i = \mathsf{normalize}^\sharp(\frac{H_i}{H^\sharp_{\mathit{final}}}) \tag{4.6}$$

Our goal is to compute $H^\sharp_{\mathit{final}}$ such that $|H^\sharp_{\mathit{final}} - H_{\mathit{final}}|$ is as smaller as possible, for which we use the pre-analysis and a statistical method. First, we compute H_{pre}, the final height of the pre-analysis result, i.e.,

$$H_{\mathit{pre}} = \mathsf{H}(\gamma(\mathsf{lfp} F^\sharp))$$

Next, we statistically refine H_{pre} into $H^\sharp_{\mathit{final}}$ such that $|H^\sharp_{\mathit{final}} - H_{\mathit{final}}|$ is likely smaller than $|H_{\mathit{pre}} - H_{\mathit{final}}|$. The job of the statistical method is to predict $\alpha = \frac{H_{\mathit{final}}}{H_{\mathit{pre}}}$ $(0 \leq \alpha \leq 1)$ for a given program. With α, $H^\sharp_{\mathit{final}}$ is defined as follows:

$$H^\sharp_{\mathit{final}} = \alpha \cdot H_{\mathit{pre}}$$

We assume that α is defined as a linear combination of a set of program features in Table 4.1. We used eight syntactic features and six semantic features. The features are selected among over 30 features by feature selection for the purpose of removing redundant or irrelevant ones for better accuracy. We used L1 based recursive feature elimination to find optimal subset of features using 254 benchmark programs.

The feature values are normalized to real numbers between 0 and 1. The Post-fixpoint features are about the post-fixpoint property. Since the pre-analysis result is a post fixpoint of the semantic function F, i.e., $\gamma(\mathsf{lfp} F^{\sharp}) \in \{x \in \mathbb{D} \mid x \sqsupseteq F(x)\}$, we can refine the result by iteratively applying F to the pre-analysis result. Instead of doing refinement, we designed simple indicators that show possibility of the refinement to avoid extra cost. For every traning example, a feature vector is created with a negligible overhead.

We used the ridge linear regression as the learning algorithm. The ridge linear regression algorithm is known as a quick and effective technique for numerical prediction.

Table 4.1: The feature vector used by linear regression to construct prediction models

Category	Feature
Inter-procedural (syntactic)	# function calls in the program
	# functions in recursive call cycles
	# undefined library function calls
Loop-related (syntactic)	the maximum loop size
	the average loop sizes
	the standard deviation of loop sizes
	the standard deviation of depths of loops
	# loopheads
Numerical analysis (semantic)	# bounded intervals in the pre-analysis result
	# unbounded intervals in the pre-analysis result
Pointer analysis (semantic)	# points-to sets of cardinality over 4 in the pre-analysis result
	# points-to sets of cardinality under 4 in the pre-analysis result
Post-fixpoint (semantic)	# program points where applying the transfer function once improves the precision
	height decrease when transfer function is applied once

In a way orthogonal to the statistical method, we further reduce $|H^{\sharp}_{\mathit{final}} - H_{\mathit{final}}|$ by tuning the height function. We reduce $|H^{\sharp}_{\mathit{final}} - H_{\mathit{final}}|$ by considering only subsets of program points and abstract locations. However, it is not the best way to choose the smallest subsets of them when computing heights. For example, we may simply set both of them to be an empty set. Then, $|H^{\sharp}_{\mathit{final}} - H_{\mathit{final}}|$ will be zero, but both H_{final} and $H^{\sharp}_{\mathit{final}}$ will be also zero. Undoubtedly, that results in a useless progress bar as estimated progress is always zero in that case.

Our goal is to choose program points and abstract locations as small as possible, while maintaining the progress estimation quality. To this end, we used the following two heuristics:

- We focus only on abstract locations that contribute to increases of heights during the main analysis. Let $\mathsf{D}(c)$ an over-approximation of the set of such abstract locations at program point c:

  $$\mathsf{D}(c) \supseteq \{l \in \hat{\mathbb{D}} \mid \exists i \in \{1 \ldots n\}.\mathsf{h}(X_i(c)(l)) - \mathsf{h}(X_{i-1}(c)(l)) > 0\}$$

  Note that since we cannot obtain the set a priori, we use an over-approximation.

- We consider only on flow cycle headers in the height calculation. This is because cycle headers are places where significant operations (join and widening) happen.

Thus, we revise the height function $\mathsf{H} : \mathbb{D} \to \mathbb{N}$ in (4.4) as follows:

$$\mathsf{H}(X) = \sum_{c \in \mathbb{W}} \sum_{l \in \mathsf{D}(c)} \mathsf{h}(X(c)(l)) \tag{4.7}$$

Because $\mathbb{W} \subseteq \mathbb{C}$ and $\forall c.\ \mathsf{D}(c) \subseteq \hat{\mathbb{D}}$, the height approximation error for the new H is smaller than that of the original H in (4.4).

We performed 3-fold cross validation using 254 benchmarks including GNU softwares and linux packages. For interval analysis, we obtained 0.06 as a mean absolute error of α, and 0.05 for pointer analysis.

4.5 Experiments

In this section, we evaluate our progress estimation technique described so far. We show that our technique effectively estimates the progress of an interval domain–based static analyzer, and a pointer analyzer for C programs.

4.5.1 Setting

We evaluate our progress estimation technique with Sparrow [2], a realistic C static analyzer that detects memory errors such as buffer-overruns and null dereferences. Sparrow basically performs a flow-sensitive and context-insensitive analysis with the interval abstract domain. The abstract state is a map from abstract locations (including program variables, allocation-sites, and structure fields) to abstract values (including intervals, points-to sets, array and structure blocks). Details on Sparrow's abstract semantics is available at [49]. Sparrow performs a sparse analysis [49] and the analysis has two phases: data dependency generation and fixpoint computation. Our technique aims to estimate the progress of the fixpoint computation step and, in this paper, we mean by analysis time the fixpoint computation time.

We have implemented our technique as described in Section 4.2 and 4.4. We used the height function defined in Example 4.4.1 and 4.4.2. To estimate numerical, and pointer analysis progresses, we split the Sparrow into two analyzers so that each of them may analyze only numeric or pointer-related property respectively. The pre-analysis is based on the partial flow-sensitivity defined in Section 4.4.2, where we set the parameter *depth* as 1 by default. That is, the pre-analysis is flow-sensitive only for flow cycle headers and their immediate preceding points.

All our experiments were performed on a machine with a 3.07 GHz Intel Core i7 processor and 24 GB of memory. For statistical estimation of the final height, we used the `scikit-learn` machine learning library [54].

4.5.2 Results

We tested our progress estimation techniques on 8 GNU software packages for each of analyses. Table 4.2 and 4.3 show our results.

The **Linearity** column in Table 4.2, and 4.3 quantifies the "linearity", which we define as follows:

$$1 - \frac{\sum_{1\leq i\leq n}(\frac{i}{n} - \bar{P}^{\sharp}_{i})^2}{\sum_{1\leq i\leq n}(\frac{i}{n} - \frac{n+1}{2n})^2}$$

Table 4.2: Progress estimation results (interval analysis). **LOC** shows the lines of code before pre-processing. **Main** reports the main analysis time. **Pre** reports the time spent by our pre-analysis. **Linearity** indicates the quality of progress estimation (best : 1). **Height-Approx.** denotes the precision of our height approximation (best : 1). **Err** denotes mean of absolute difference between **Height-Approx.** and 1 (best : 0).

Program	**LOC**	**Time(s)**		**Linearity**	**Overhead**	**Height-Approx.**
		Main	**Pre**			
bison-1.875	38841	3.66	0.91	0.73	24.86%	1.03
screen-4.0.2	44745	40.04	2.37	0.86	5.92%	0.96
lighttpd-1.4.25	56518	27.30	1.21	0.89	4.43%	0.92
a2ps-4.14	64590	32.05	11.26	0.51	35.13%	1.06
gnu-cobol-1.1	67404	413.54	99.33	0.54	24.02%	0.91
gnugo	87575	1541.35	7.35	0.89	0.48%	1.12
bash-2.05	102406	16.55	2.26	0.80	13.66%	0.93
sendmail-8.14.6	136146	1348.97	5.81	0.69	0.43%	0.93
TOTAL	686380	3423.46	130.5	**0.74**	**3.81%**	**Err : 0.07**

Table 4.3: Progress estimation results (pointer analysis).

Program	**LOC**	**Time(s)**		**Linearity**	**Overhead**	**Height-Approx.**
		Main	**Pre**			
screen-4.0.2	44745	15.89	1.56	0.90	9.82%	0.98
lighttpd	56518	11.54	0.87	0.76	7.54%	1.03
a2ps-4.14	64590	10.06	3.48	0.65	34.59%	1.04
gnu-cobol-1.1	67404	32.27	12.22	0.91	37.87%	1.03
gnugo	87575	217.77	3.88	0.64	1.78%	0.97
bash-2.05	102406	3.68	0.78	0.56	21.20%	1.04
proftpd-1.3.2	126996	74.64	11.14	0.82	14.92%	1.03
sendmail-8.14.6	136146	145.62	3.15	0.58	2.16%	0.98
TOTAL	686380	511.47	37.08	**0.73**	**7.25%**	**Err : 0.03**

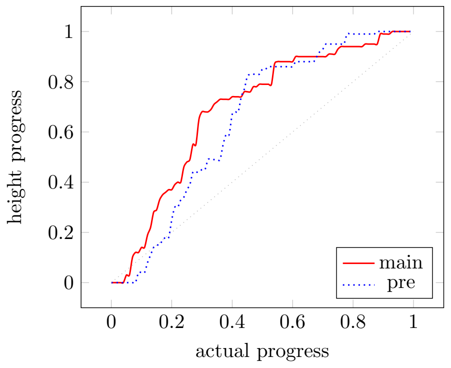
where n is the number of iterations required for the analysis to stabilize and $\bar{P}_i^\sharp$ is the estimated progress at ith iteration of the analysis. This metric is just a simple application of the coefficient of determination in statistics, i.e., R^2, which presents how well $\bar{P}^\sharp$ fits the actual progress rate $\frac{i}{n}$. The closer to 1 linearity is, the more similar to the ideal progress bar $\bar{P}^\sharp_i$ is. Figure 8.1 and 8.2 in appendix present the resulting progress bars for each of benchmark programs providing graphical descriptions of the linearity. In particular, the progress bar proceeds almost linearly for programs of the linearity close to 0.9 (`lighttpd-1.4.25`, `gnugo-3.8` in interval analysis, `gnu-cobol-1.1`, `bash-2.05` in pointer analysis). For some programs of relatively low linearity (`gnu-cobol-1.1`, `bash-2.05` in interval analysis, `gnugo-3.8`, `proftpd-1.3.2` in pointer analysis), the progress estimation is comparatively rough but still useful.

The **Height-Approx.** column stands for the accuracy of final height approximation $\frac{H_{\mathit{final}}}{H^\sharp_{\mathit{final}}}$ where $H^\sharp_{\mathit{final}}$ is estimated final height via the statistical technique described in section 4.4.3. **Err** denotes an average of absolute errors |**Height-Approx.** -1|. To prove our statistical method avoids overfitting problem, we performed 3-fold cross validation using 254 benchmarks including GNU softwares and linux packages. For interval analysis, we obtained 0.063 **Err** with 0.007 standard deviation. For pointer analysis, 0.053 **Err** with 0.001 standard deviation. These results show our method avoids overfitting, evenly yielding precise estimations at the same time.

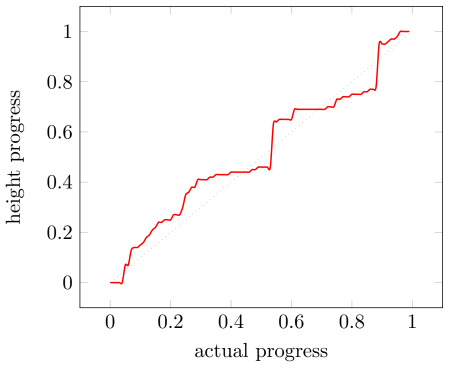
The **Overhead** column shows the total overhead of our method, which includes the pre-analysis running time (Section 4.4.2). The average performance overheads of our method are 3.8% in interval analysis, and 7.3% in pointer analysis respectively.

4.5.3 Discussion

Linearity vs. Overhead In our progress estimation method, we can make tradeoffs between the linearity and overhead. Table 4.2, 4.3 show our progress estimations when we use the default parameter value ($\mathit{depth} = 1$) in the pre-analysis. By using a higher *depth* value, we can improve the precision of the pre-analysis and hence the quality of the resulting progress estimation at the cost of extra over-



(a) original height-progress



(b) normalized height-progress

Figure 4.2: Our method is also applicable to octagon domain–based static analyses.

head. For two programs, the following table shows the changes in linearity and overhead when we change *depth* from 1 to 3:

Program	**Linearity change**	**Overhead change**
bash-2.05 (pointer)	$0.56 \rightarrow 0.70$	$21.2\% \rightarrow 37.5\%$
sendmail-8.14.6 (interval)	$0.69 \rightarrow 0.95$	$0.4\% \rightarrow 18.4\%$

Height Approximation Error In our experiments, we noticed that our progress estimation method is sensitive to the height approximation error ($H^{\sharp}_{final} - H_{final}$). Although we precisely estimate heights of the fixpoints, there are cases where even small error sometimes leads to unsatisfactory results. For instance, the reason why the progress for `gnu-cobol-1.1` is under-estimated is the height approximation error (0.09).

We believe enhancing the precision will be achieved by increasing training examples and relevant features.

4.6 Application to Relational Analyses

The overall approach of our progress estimation technique may adapt easily to relational analyses as well. In this section, we check the possibility of applying our technique to the octagon domain–based static analysis [45].

We have implemented a prototype progress estimator for the octagon analysis (described in §2.2.2) as follows. For pre-analysis, we used the same partial flow-sensitive abstraction described in Section 4.4.2 with *depth* = 1. Regarding the height function H, we also used that of the interval analysis. Note that, since an octagon domain element is a collection of intervals denoting ranges of program variables such as x and y, their sum $x+y$, and their difference $x-y$, we can use the same height function in Example 4.4.1. In this prototype implementation, we assumed that we are given heights of the final analysis results.

Figure 4.2 shows that our technique effectively normalizes the height progress of the octagon analysis. The solid lines in Figure 4.2(a) depicts the height progress of the main octagon analysis of program `wget-1.9` and the dotted line shows that of the pre-analysis. By normalizing the main analysis' progress behavior, we obtain the progress bar depicted in Figure 4.2(b), which is almost linear.

Figure 8.3 depicts the resulting progress bar for other benchmark programs, and the following table reports detailed experimental results.

		Time(s)			
Program	**LOC**	**Main**	**Pre**	**Linearity**	**Overhead**
httptunnel-3.3	6174	49.5	8.2	0.91	16.6%
combine-0.3.3	11472	478.2	16	0.89	3.4%
bc-1.06	14288	63.9	43.8	0.96	68.6%
tar-1.17	18336	977.0	73.1	0.82	7.5%
parser	18923	190.1	104.8	0.97	55.1%
wget-1.9	35018	3895.36	1823.15	0.92	46.8%
TOTAL	69193	5654.0	2069.49	**0.91**	**36.6%**

Even though we completely reused the pre-analysis design and height function for the interval analysis, the resulting progress bars are almost linear. This preliminary results suggest that our method could be applicable to relational analyses.

Chapter 5

Method 3. Static Analysis with Set-closure in Secrecy

5.1 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. 5.1 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 [22]. Since then, although there have been many efforts to improve the efficiency [6, 7, 13, 59], 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 (described in §6.3).

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 5.1.

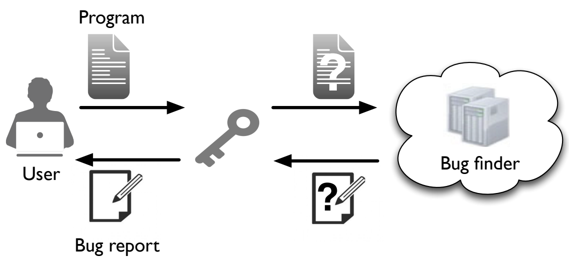


Figure 5.1: Secure static analysis is performed in 3 steps: 1) target program encryption 2) analysis in secrecy, and 3) analysis result decryption

	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 5.1: 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.

5.2 Background

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

5.2.1 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.*

[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.

To facilitate understanding of HE schemes, we introduce a simple symmetric version of the HE scheme [60] based on approximate common divisor problems [29]:

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

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

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 [7, 8, 15, 23] 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 [12, 60, 22, 58] 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.

5.2.2 The BGV-type cryptosystem

Our underlying HE scheme is a variant of the Brakerski-Gentry-Vaikuntanathan (BGV)-type cryptosystem [7, 23]. In this section, we only provide a brief review of the cryptosystem [7]. For more details, please refer to [7, 23]. 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 $\bar{\mathbf{c}}_{\mathsf{mult}} = \mathsf{BGV.Enc}(\mu_1 \cdot \mu_2)$.

5.2.3 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 [7] for the details), analyzers cannot learn no more information than the program size.

5.3 A Basic Construction of a Pointer Analysis in Secrecy

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

5.3.1 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 [28]. Given some program P, we construct resolution rules as specified in Table 5.2. 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.

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

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

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

Table 5.2: Resolution rules for pointer analysis.

5.3.2 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 $\mathtt{x_i}$. 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

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

for variables $\mathtt{x_i}$ and $\mathtt{x_j}$, 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 5.2. 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:

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

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: $x \vee y = (x \wedge y) \oplus x \oplus y$. 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.

5.4 Improvement of the Pointer Analysis in Secrecy

In this section, we present three techniques to reduce the cost of the basic approach described in § 5.3.2. We begin with problems of the basic approach followed by our solutions.

5.4.1 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 § 5.3.2 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 § 5.3.2 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.

5.4.2 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 § 5.3.2. 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	$O(n \log m)$	$(2n+2)m$

5.4.3 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
$\mathsf{x} = \mathsf{y}$	$\mathsf{ptl}(\mathsf{x}) = \mathsf{ptl}(\mathsf{y})$
$\mathsf{x} = *\mathsf{y}$	$\mathsf{ptl}(\mathsf{y}) = \mathsf{ptl}(\mathsf{x}) + 1$
$*\mathsf{p} = \mathsf{y}$	$\mathsf{ptl}(\mathsf{p}) = \mathsf{ptl}(\mathsf{x}) + 1 \wedge \mathsf{ptl}(\mathsf{y}) = \mathsf{ptl}(\mathsf{x})$

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

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

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

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

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

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, $\mathbb{Z}_q = [-q/2, q/2) \cap \mathbb{Z}$ 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 $R_p = \mathbb{Z}[X]/\left(p, \Phi(X)\right)$ 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:

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

First, we view a component μ_i in a vector $\boldsymbol{\mu} = (\mu_1, \cdots, \mu_m)$ as a contant polynomial $\mu_i \in \mathbb{Z}_p[X]/\left(F_i(X)\right)$ 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 $\mathsf{BGV.Enc}\left(\mu_1, \cdots, \mu_m\right)$. For more details to the ciphertext packing, we suggest that readers see the paper [57].

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 5.3: Circuit expression of the level-by-level analysis

Homomorphic Matrix Operations

Applying the resolution rules in the level-by-level analysis in § 5.4.3 can be re-written in a matrix form as shown in Table 5.3. In Table 5.3, $\Delta_\ell = [\delta_{i,j}^{(\ell)}], H_\ell = [\eta_{i,j}^{(\ell)}], U = [u_{i,j}]$, 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 [26] to perform multiplication between encrypted matrices. We use the Replicate homomorphic operation supported by the BGV scheme [26]. 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 [26]. Algorithms for the homomorphic operations on encrypted matrices are described in Fig. 8.5 in Appendix C.

5.4.5 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. 8.4 in Appendix C.

5.5 Experimental Result

In this section, we demonstrate the performance of the pointer analysis in secrecy. In our experiment, we use HElib library [26], 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 5.4 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.

Table 5.4: 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

Why "Basic" Algorithm?

Many optimization techniques to scale the pointer analysis to larger programs [20, 21, 27, 28, 53] cannot be applied into our setting without exposing much information of the program. The details are given in §6.3.

5.6 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.

Chapter 6

Related Works

6.1 Sound Non-statistical Alarm Clustering

Non-statistical alarm clustering To our best knowledge, Le et al.'s work [37] is the first one that proposes non-statistical clustering method. They reduce the number of faults (alarms) by detecting correlations (dependencies) between them. By propagating the effects of the error state along the program path, they detect the correlation of pairs of alarms. They automatically construct a correlation graph which shows how faults are correlated. Based on the graph, we can reduce the number of faults to consider.

However, Le et al.'s method is not sound, while our method is sound. According to their experiment results, the dependencies they use to construct the correlation graph can be spurious (false positive), which means that it is not always safe to rule out faults even though they are correlated to the others.

Statistical approaches Statistical ranking schemes [31, 35, 36] may help to find real errors quickly, but ranking schemes do not reduce alarm-investigation burdens as in our work. Since our technique is orthogonal to statistical ranking schemes, we might combine our technique with them for a more sophisticated alarm reporting interface.

Error recovery technique Our work is more general than error recovery technique that is used for reducing false alarms in many commercial static analysis tools [3, 43, 44]. For each alarm found, the commercial analyzers recover from those alarms; i.e. they assume that an alarm is false when they passed the alarm point. Because error recovery is done within the baseline analysis, possible refinements are bounded by the expressiveness of the abstract domain of the baseline. As we show in § 3.4.4, we can use more expressive domain for clustering purpose than the one used in the baseline, which can be more cost-effective than using expensive abstract domain in the baseline. Additionally, our method can derive true clusters for which cannot be done by the error recovery technique.

Abstraction refinement Our work resembles to Rival's work [55] in the sense that both work refines the abstraction by exploiting the information about error state. In his work, Rival refines the abstraction by slicing out non-error states and sees if the initial state after refinement still insists that the erroneous states are reachable. If the initial state becomes bottom after refinement, the alarm turns out to be false. On the other hand, in our work, we refine the abstraction by slicing out erroneous states at one point and see if erroneous states at other points become non-reachable, which means that we found the dependence between alarms.

Our clustering method can be integrated with other refinement approaches [24, 25, 33, 55]. The goal of them is to remove false alarms by abstraction refinement, while our work is to reduce the number of alarms to investigate. Our work can reduce the number of targets to do the refinement.

6.2 A Progress Bar for Static Analyzers

Though progress estimation techniques have been extensively studied in other fields [48, 34, 10, 42, 46, 47], there have been no research for static analyzers. For instance, a variety of progress estimation techniques have been proposed for long-running software systems such as databases [34, 10, 42] and parallel data processing systems [47, 46]. Static analyzers are also a long-running software system but there are no progress estimation techniques for them. Furthermore, our method is different from existing techniques. Existing progress estimators [46, 42, 47, 10] and algorithm runtime prediction [30] are based solely on statistics or machine learning. By contrast, we propose a technique that combines a semantics-based pre-analysis with machine learning.

6.3 Static Analysis with Set-closure in Secrecy

Application-specific optimizations in using HE For practical deployments of HE, ppplication-specific techniques are often demonstrated in other fields. Kim et al. [11] 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 [9]. Also, private database query protocol using somewhat homomorphic encryption has been proposed [4].

Optimization techniques in pointer analysis There are many optimization techniques to scale the pointer analysis to larger programs [20, 21, 27, 28, 53], but they 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 [20, 27, 28, 53] aims to prevent redundant computation of transitive closure 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 [21, 53], 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.

Chapter 7

Conclusions

We have presented our solutions to the major usability issues in using sound static analyzers: many false alarms, missing progress indicator, and copy-right concerns over software sources.

- We have presented a new, sound non-statistical alarm-clustering method. We proposed an abstract interpretation–based framework of alarm-clustering, which is generally applicable to any semantics-based static analyses. We formally proved the soundness of the framework, presented practical algorithms to find the set of dominant alarms, provided three instance clustering algorithms (based on interval, octagon, and symbolic domains), and showed that the combination of the interval and symbolic clustering method considerably reduces the number of final alarm reports of a realistic C static analyzer.

- We have proposed a technique for estimating static analysis progress. Our technique is based on the observation that semantically related analyses would have similar progress behaviors, so that the progress of the main analysis can be estimated by a pre-analysis. We implemented our technique on top of a realistic C static analyzer and show our technique effectively estimates its progress.

- 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 for the m number of pointer variables. We also show the viability of our work by implementing the pointer analysis in secrecy. A major future direction is adapting other kinds of static analysis operations (*e.g.,* arbitrary $\sqcup$, $\sqsubseteq$, and semantic operations) into HE schemes.

Bibliography

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

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

[3] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 196–207, New York, NY, USA, 2003. ACM.

[4] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J. Wu. Private database queries using somewhat homomorphic encryption. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, pages 102–118, Berlin, Heidelberg, 2013. Springer-Verlag.

[5] Francois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141, 1993.

[6] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer Berlin Heidelberg, 2012.

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.

[8] Zvika Brakerski and Vinod 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, Washington, DC, USA, 2011. IEEE Computer Society.

[9] Ayantika Chatterjee, Manish Kaushal, and Indranil Sengupta. Accelerating sorting of fully homomorphic encrypted data. In Goutam Paul and Serge Vaudenay, editors, *Progress in Cryptology – INDOCRYPT 2013*, volume 8250 of *Lecture Notes in Computer Science*, pages 262–273. Springer International Publishing, 2013.

[10] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 803–814, New York, NY, USA, 2004. ACM.

[11] Jung Hee Cheon, Miran Kim, and Kristin Lauter. Homomorphic Computation of Edit Distance. *IACR Cryptology ePrint Archive*, 2015:132, 2015. To appear in WAHC 2015.

[12] JungHee Cheon, Jean-Sébastien Coron, Jinsu Kim, MoonSung Lee, Tancrède Lepoint, Mehdi Tibouchi, and Aaram Yun. Batch fully homomorphic encryption over the integers. In Thomas Johansson and PhongQ. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 315–335. Springer Berlin Heidelberg, 2013.

[13] JungHee Cheon and Damien Stehlé. Fully homomophic encryption over the integers revisited. In Elisabeth Oswald and Marc Fischlin, editors, *Advances*

in Cryptology – EUROCRYPT 2015, volume 9056 of *Lecture Notes in Computer Science*, pages 513–536. Springer Berlin Heidelberg, 2015.

[14] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. Begbunch: Benchmarking for c bug detection tools. In *Proceedings of the 2Nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, DEFECTS '09, pages 16–20, New York, NY, USA, 2009. ACM.

[15] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Proceedings of the 31st Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'12, pages 446–464, Berlin, Heidelberg, 2012. Springer-Verlag.

[16] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, December 2009.

[17] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[18] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[19] Vijay D'Silva, Leopold Haller, Daniel Kroening, and Michael Tautschnig. Numeric bounds analysis with conflict-driven learning. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction*

and Analysis of Systems, TACAS'12, pages 48–63, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM.

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

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

[23] Craig Gentry, Shai Halevi, and NigelP. Smart. Homomorphic evaluation of the aes circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer Berlin Heidelberg, 2012.

[24] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically refining abstract interpretations. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 443–458, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] BhargavS. Gulavani and SriramK. Rajamani. Counterexample driven refinement for abstract interpretation. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 474–488. Springer Berlin Heidelberg, 2006.

[26] Shai Halevi and Victor Shoup. Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106, 2014. `http://eprint.iacr.org/`.

[27] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[28] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.

[29] Nick Howgrave-Graham. Approximate integer common divisors. In JosephH. Silverman, editor, *Cryptography and Lattices*, volume 2146 of *Lecture Notes in Computer Science*, pages 51–66. Springer Berlin Heidelberg, 2001.

[30] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: The state of the art. *CoRR*, abs/1211.0906, 2012.

[31] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. In *Proceedings of the 12th International Conference on Static Analysis*, SAS'05, pages 203–217, Berlin, Heidelberg, 2005. Springer-Verlag.

[32] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. Mecc: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 301–310, New York, NY, USA, 2011. ACM.

[33] Youil Kim, Jooyong Lee, Hwansoo Han, and Kwang-Moo Choe. Filtering false alarms of buffer overflow analysis using smt solvers. *Inf. Softw. Technol.*, 52(2):210–219, February 2010.

[34] Arnd Christian König, Bolin Ding, Surajit Chaudhuri, and Vivek Narasayya. A statistical approach towards robust progress estimation. *Proc. VLDB Endow.*, 5(4):382–393, December 2011.

[35] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 83–93, New York, NY, USA, 2004. ACM.

[36] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 295–315, Berlin, Heidelberg, 2003. Springer-Verlag.

[37] Wei Le and Mary Lou Soffa. Path-based fault correlations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 307–316, New York, NY, USA, 2010. ACM.

[38] Woosuk Lee, Hyunsook Hong, Kwangkeun Yi, and JungHee Cheon. Static analysis with set-closure in secrecy. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis*, volume 9291 of *Lecture Notes in Computer Science*, pages 18–35. Springer Berlin Heidelberg, 2015.

[39] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 299–314, Berlin, Heidelberg, 2012. Springer-Verlag.

[40] Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. A progress bar for static analyzers. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis*, volume 8723 of *Lecture Notes in Computer Science*, pages 184–200. Springer International Publishing, 2014.

[41] Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 31–42, New York, NY, USA, 2011. ACM.

[42] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 791–802, New York, NY, USA, 2004. ACM.

[43] MathWorks. Polyspace embedded software verification. `http://www.mathworks.com/products/polyspace/index.html`.

[44] Microsoft. Code contracts, 2015. `http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`.

[45] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[46] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. In *Proc. of ICDE*, pages 681–684. IEEE, 2010.

[47] Kristi Morton, Magdalena Balazinska, and Dan Grossman. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 507–518, New York, NY, USA, 2010. ACM.

[48] Brad A. Myers. The importance of percent-done progress indicators for computer-human interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '85, pages 11–17, New York, NY, USA, 1985. ACM.

[49] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for c-like languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 229–238, New York, NY, USA, 2012. ACM.

[50] Hakjoo Oh and Kwangkeun Yi. An algorithmic mitigation of large spurious interprocedural cycles in static analysis. *Softw. Pract. Exper.*, 40(8):585–603, July 2010.

[51] Hakjoo Oh and Kwangkeun Yi. Access-based localization with bypassing. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems*, APLAS'11, pages 50–65, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[53] D.J. Pearce, P.H.J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *SCAM*, 2003.

[54] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[55] Xavier Rival. Understanding the origin of alarms in astrÉe. In *Proceedings of the 12th International Conference on Static Analysis*, SAS'05, pages 303–319, Berlin, Heidelberg, 2005. Springer-Verlag.

[56] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans on Programming Languages and System*, 29(5):26–51, 2007.

[57] Nigel P. Smart and Frederik Vercauteren. Fully Homomorphic SIMD Operations. *IACR Cryptology ePrint Archive*, 2011:133.

[58] N.P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In PhongQ. Nguyen and David Pointcheval, editors, *Public Key Cryptography – PKC 2010*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer Berlin Heidelberg, 2010.

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

[60] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer Berlin Heidelberg, 2010.

[61] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static c analyzer. *Inf. Process. Lett.*, 102(2-3):118–123, April 2007.

Chapter 8

Appendix

A Proofs of Theorems

Lemma 1. Given two alarms φ_1 and φ_2, if $\varphi_1 \rightsquigarrow \varphi_2$, then φ_2 is false whenever φ_1 is false.
(Stated in § 3.2.)

Proof.

$$\begin{array}{ll} \alpha_S(\llbracket P \rrbracket_{/\delta}(\varphi_1) \ominus \Omega(\varphi_1)) \sqsubseteq \llbracket \hat{P} \rrbracket(\varphi_1) \ \hat{\ominus} \ \alpha_S(\Omega(\varphi_1)) & (\alpha_S \circ \ominus \sqsubseteq \hat{\ominus} \circ \alpha_{S \times S}) \\ \alpha_S(\llbracket P \rrbracket_{/\delta}(\varphi_1)) \sqsubseteq \llbracket \hat{P} \rrbracket_{\neg\varphi_1}(\varphi_1) & (\llbracket P \rrbracket_{/\delta}(\varphi_1) \cap \Omega(\varphi_1) = \emptyset) \end{array}$$

Because $\forall \varphi \neq \varphi_1.\ \llbracket \hat{P} \rrbracket(\varphi) = \llbracket \hat{P} \rrbracket_{\neg\varphi_1}(\varphi)$, $\alpha(\llbracket P \rrbracket) \sqsubseteq \llbracket \hat{P} \rrbracket_{\neg\varphi_1}$.

$$\begin{array}{ll} \alpha(\llbracket P \rrbracket) = \alpha(\bigsqcup_{i \in \mathbb{N}} {F_P}^i \bot) \sqsubseteq \llbracket \hat{P} \rrbracket_{\neg\varphi_1} & \\ \bigsqcup_{i \in \mathbb{N}} {F_P}^i \bot \sqsubseteq \gamma(\llbracket \hat{P} \rrbracket_{\neg\varphi_1}) & (id \sqsubseteq \gamma \circ \alpha) \\ \forall i \in \mathbb{N}.\ {F_P}^i \bot \sqsubseteq \gamma(\llbracket \hat{P} \rrbracket_{\neg\varphi_1}) & (\text{By definition of lub}) \end{array}$$

Because $\gamma \circ \alpha \sqsubseteq id$,

$$\forall i \in \mathbb{N}.\ \alpha({F_P}^i \bot) \sqsubseteq \llbracket \hat{P} \rrbracket_{\neg\varphi_1} \tag{8.1}$$

Let $\hat{H} = \lambda \hat{X}. \llbracket \hat{P} \rrbracket_{\neg\varphi_1} \sqcap \hat{F}(\hat{X})$. We will show $\alpha(\llbracket P \rrbracket) \sqsubseteq \mathsf{fix}^{\#}\ \hat{H} = \llbracket \tilde{P} \rrbracket_{\varphi_1}$.
For the purpose, we first show $\forall i \in \mathbb{N}.\ \alpha({F_P}^i \bot) \sqsubseteq \hat{H}^i(\hat{\bot})$.

- Basis :

$$\begin{aligned}
&\alpha(F_P(\bot)) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} && \text{(By 8.1)}\\
&\alpha(F_P(\bot)) \sqsubseteq \hat{F}(\bot) && (\alpha \circ F_P \sqsubseteq \hat{F} \circ \alpha)\\
&\therefore \alpha(F_P(\bot)) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} \sqcap \hat{F}(\hat{\bot}) = \hat{H}(\hat{\bot})
\end{aligned}$$

- Induction step :

  IH : $\alpha({F_P}^k\bot) \sqsubseteq \hat{H}^k(\hat{\bot})$

$$\begin{aligned}
&\alpha({F_P}^{k+1}\bot) = \alpha(F_P \circ {F_P}^k\bot)\\
&\sqsubseteq \alpha(F_P \circ \gamma \circ \alpha \circ {F_P}^k\bot) && (\alpha \circ F_P \text{ is monotone,}\\
& && \text{and } id \sqsubseteq \gamma \circ \alpha)\\
&\sqsubseteq \alpha \circ F_P \circ \gamma(\hat{H}^k(\hat{\bot})) && \text{(By IH)}\\
&\sqsubseteq \hat{F}(\hat{H}^k(\hat{\bot})) && (\alpha \circ F_P \sqsubseteq \hat{F} \circ \alpha)
\end{aligned}$$

$$\begin{aligned}
&\alpha({F_P}^{k+1}\bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} && \text{(By 8.1)}\\
&\therefore \alpha({F_P}^{k+1}\bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} \sqcap \hat{F}(\hat{H}^k(\hat{\bot})) = \hat{H}^{k+1}(\hat{\bot})
\end{aligned}$$

Therefore

$$\begin{aligned}
&\textstyle\bigsqcup_{i\in\mathbb{N}} \alpha({F_P}^i\bot) \sqsubseteq \bigsqcup_{i\in\mathbb{N}} \hat{H}^i(\hat{\bot})\\
&\alpha(\textstyle\bigsqcup_{i\in\mathbb{N}} {F_P}^i\bot) \sqsubseteq \bigsqcup_{i\in\mathbb{N}} \hat{H}^i(\hat{\bot}) \quad (\alpha \text{ is continuous.})\\
&\therefore \alpha([\![P]\!]) \sqsubseteq \mathsf{fix}^\# \ \hat{H} = [\![\tilde{P}]\!]_{\varphi_1}
\end{aligned}$$

$$\begin{aligned}
&[\![P]\!]/_\delta(\varphi_2) \subseteq \gamma_S([\![\tilde{P}]\!]_{\varphi_1}(\varphi_2)) && (\alpha([\![P]\!]) \sqsubseteq [\![\tilde{P}]\!]_{\varphi_1})\\
&\therefore [\![P]\!]/_\delta(\varphi_2) \cap \Omega(\varphi_2) = \varnothing && (\gamma_S([\![\tilde{P}]\!]_{\varphi_1}(\varphi_2)) \cap \Omega(\varphi_2) = \varnothing)
\end{aligned}$$

□

Lemma 2. Given set $\overrightarrow{\varphi}$ of alarms and alarm φ_0, if $\overrightarrow{\varphi} \rightsquigarrow \varphi_0$, then alarm φ_0 is false whenever all alarms in $\overrightarrow{\varphi}$ are false.
(Stated in § 3.2.)

Proof. Similar to the proof of Lemma 1.

$$\begin{array}{ll}
\forall \varphi \in \overrightarrow{\varphi}.\ \alpha_S([\![P]\!]_{/\delta}(\varphi) \ominus \Omega(\varphi)) \sqsubseteq [\![\hat{P}]\!](\varphi)\ \hat{\ominus}\ \alpha_S(\Omega(\varphi)) & (\alpha_S \circ \ominus \sqsubseteq \hat{\ominus} \circ \alpha_{S\times S}) \\
\therefore \forall \varphi \in \overrightarrow{\varphi}.\ \alpha_S([\![P]\!]_{/\delta}(\varphi)) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi}(\varphi) & (\forall \varphi \in \overrightarrow{\varphi}.[\![P]\!]_{/\delta}(\varphi) \cap \Omega(\varphi) = \emptyset) \\
\therefore \forall \varphi \in \overrightarrow{\varphi}.\ \alpha_S([\![P]\!]_{/\delta}(\varphi)) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}}(\varphi) & ([\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} = \bigsqcap_{\varphi\in\overrightarrow{\varphi}} [\![\hat{P}]\!]_{\neg\varphi})
\end{array}$$

Because $\forall \varphi \in \Phi - \overrightarrow{\varphi}.\ [\![\hat{P}]\!](\varphi) = [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}}(\varphi)$, $\alpha([\![P]\!]) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}}$.

$$\begin{array}{ll}
\alpha([\![P]\!]) = \alpha(\bigsqcup_{i\in\mathbb{N}} {F_P}^i\bot) \sqsubseteq [\![\hat{P}]\!]_{\overrightarrow{\varphi}} & \\
\bigsqcup_{i\in\mathbb{N}} {F_P}^i\bot \sqsubseteq \gamma([\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}}) & (id \sqsubseteq \gamma \circ \alpha) \\
\forall i \in \mathbb{N}.\ {F_P}^i\bot \sqsubseteq \gamma([\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}}) & (\text{By definition of lub})
\end{array}$$

Because $\gamma \circ \alpha \sqsubseteq id$,

$$\forall i \in \mathbb{N}.\ \alpha({F_P}^i\bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} \tag{8.2}$$

Let $\hat{H} = \lambda\hat{X}.[\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} \sqcap \hat{F}(\hat{X})$. We will show $\alpha([\![P]\!]) \sqsubseteq \mathsf{fix}^{\#}\ \hat{H} = [\![\tilde{P}]\!]_{\overrightarrow{\varphi}}$.
For the purpose, we first show $\forall i \in \mathbb{N}.\ \alpha({F_P}^i\bot) \sqsubseteq \hat{H}^i(\hat{\bot})$.

- Basis :

$$\begin{array}{ll}
\alpha(F_P(\bot)) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} & (\text{By 8.2}) \\
\alpha(F_P(\bot)) \sqsubseteq \hat{F}(\hat{\bot}) & (\alpha \circ F_P \sqsubseteq \hat{F} \circ \alpha) \\
\therefore \alpha(F_P(\bot)) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} \sqcap \hat{F}(\hat{\bot}) = \hat{H}(\hat{\bot}) &
\end{array}$$

- Induction step :

$$\text{IH} : \alpha({F_P}^k\bot) \sqsubseteq \hat{H}^k(\bot)$$

$$\begin{aligned}
&\alpha({F_P}^{k+1}\bot) = \alpha(F_P \circ {F_P}^k\bot) \\
&\sqsubseteq \alpha(F_P \circ \gamma \circ \alpha \circ {F_P}^k\bot) && (\alpha \circ F_P \text{ is monotone,} \\
& && \text{and } id \sqsubseteq \gamma \circ \alpha) \\
&\sqsubseteq \alpha \circ F_P \circ \gamma(\hat{H}^k(\hat{\bot})) && (\text{By IH}) \\
&\sqsubseteq \hat{F}(\hat{H}^k(\hat{\bot})) && (\alpha \circ F_P \sqsubseteq \hat{F} \circ \alpha)
\end{aligned}$$

$$\begin{aligned}
&\alpha({F_P}^{k+1}\bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} && (\text{By 8.2}) \\
&\therefore \alpha({F_P}^{k+1}\bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} \sqcap \hat{F}(\hat{H}^k(\hat{\bot})) = \hat{H}^{k+1}(\hat{\bot})
\end{aligned}$$

Therefore

$$\begin{aligned}
&\textstyle\bigsqcup_{i\in\mathbb{N}} \alpha({F_P}^i\bot) \sqsubseteq \bigsqcup_{i\in\mathbb{N}} \hat{H}^i(\hat{\bot}) \\
&\alpha(\textstyle\bigsqcup_{i\in\mathbb{N}} {F_P}^i\bot) \sqsubseteq \bigsqcup_{i\in\mathbb{N}} \hat{H}^i(\hat{\bot}) \quad (\alpha \text{ is continuous.}) \\
&\therefore \alpha([\![P]\!]) \sqsubseteq \mathsf{fix}^{\#}\ \hat{H} = [\![\tilde{P}]\!]_{\varphi_1}
\end{aligned}$$

$$\begin{aligned}
&[\![P]\!]/_\delta(\varphi_0) \subseteq \gamma_S([\![\tilde{P}]\!]_{\overrightarrow{\varphi}}(\varphi_0)) && (\alpha([\![P]\!]) \sqsubseteq [\![\tilde{P}]\!]_{\overrightarrow{\varphi}}) \\
&\therefore [\![P]\!]/_\delta(\varphi_0) \cap \Omega(\varphi_0) = \varnothing && (\gamma_S([\![\tilde{P}]\!]_{\overrightarrow{\varphi}}(\varphi_0)) \cap \Omega(\varphi_0) = \varnothing)
\end{aligned}$$

□

Lemma 3. $\overrightarrow{\varphi} \subseteq \overrightarrow{\varphi}' \implies \mathcal{C}_{\overrightarrow{\varphi}} \subseteq \mathcal{C}_{\overrightarrow{\varphi}'}$
(Stated in § 3.3.1.)

Proof.

$$\begin{aligned}
&[\![\tilde{P}]\!]_{\overrightarrow{\varphi}} \sqsupseteq [\![\tilde{P}]\!]_{\overrightarrow{\varphi}'} && (\text{By Lemma 5}) \\
&\forall\varphi \in \Phi.\ \gamma_S([\![\tilde{P}]\!]_{\overrightarrow{\varphi}}(\varphi)) \cap \Omega(\varphi) = \varnothing \implies \gamma_S([\![\tilde{P}]\!]_{\overrightarrow{\varphi}'}(\varphi)) \cap \Omega(\varphi) = \varnothing && (\gamma_S \text{ is monotone.})
\end{aligned}$$

Thus, $\mathcal{C}_{\overrightarrow{\varphi}} = \{\varphi \in \mathcal{A} \mid \overrightarrow{\varphi} \rightsquigarrow \varphi\} \subseteq \mathcal{C}_{\overrightarrow{\varphi}'} = \{\varphi \in \mathcal{A} \mid \overrightarrow{\varphi}' \rightsquigarrow \varphi\}$

□

Lemma 5. $\overrightarrow{\varphi} \subseteq \overrightarrow{\varphi}' \implies [\![\tilde{P}]\!]_{\overrightarrow{\varphi}'} \sqsubseteq [\![\tilde{P}]\!]_{\overrightarrow{\varphi}}$

PROOF. Note that $[\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} = \bigsqcap_{\varphi_i \in \overrightarrow{\varphi}} [\![\hat{P}]\!]_{\neg\varphi_i} \sqsupseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}'} = \bigsqcap_{\varphi_i \in \overrightarrow{\varphi}'} [\![\hat{P}]\!]_{\neg\varphi_i}$.

Let $H = \lambda Z.[\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} \sqcap \hat{F}(Z)$ and $H' = \lambda Z.[\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}'} \sqcap \hat{F}(Z)$. Then

$$\begin{array}{ll} H \sqsupseteq H' & ([\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}'}) \\ [\![\tilde{P}]\!]_{\overrightarrow{\varphi}} = \mathsf{fix}^{\#} \lambda Z.[\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} \sqcap \hat{F}(Z) & \\ \sqsupseteq \mathsf{fix}^{\#} \lambda Z.[\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}'} \sqcap \hat{F}(Z) = [\![\tilde{P}]\!]_{\overrightarrow{\varphi}'} & \text{(By the fixpoint transfer theorem [17])} \end{array}$$

□

THEOREM 4. Algorithm 3 computes sound alarm dependences.
(Stated in § 3.3.2.)

Proof. At line 28, an abstract dependence $R(\varphi) \rightsquigarrow \varphi$ is added if $T(\varphi) \sqcap \hat{\Omega}(\varphi) = \bot$. It is correct because $T = [\![\tilde{P}]\!]_{R(\varphi)}$.

Now we show $T = [\![\tilde{P}]\!]_{R(\varphi)}$. At line 33 after the function FIXPOINTITERATE is called, $T = [\![\tilde{P}]\!]_{\Phi}$. In addition, by Lemma 6, $[\![\tilde{P}]\!]_{\Phi} = [\![\tilde{P}]\!]_{R(\varphi)}$. Therefore $T = [\![\tilde{P}]\!]_{R(\varphi)}$.

□

LEMMA 6. *In algorithm 3, after the function* FIXPOINTITERATE *is called,* $[\![\tilde{P}]\!]_{\Phi} = [\![\tilde{P}]\!]_{R(\varphi)}$.

Proof. We first show that the loop invariant in the function FIXPOINTITERATE is

$$\forall \varphi \in \Phi.\ [\![\tilde{P}]\!]_{R(\varphi)}(\varphi) \sqsubseteq T(\varphi). \tag{8.3}$$

At the first entrance to the loop, $T = [\![\hat{P}]\!]_{\neg\Phi}$ and $\forall \varphi \in \Phi. R(\varphi) = \{\varphi\}$.

$$\begin{array}{ll} \forall \varphi \in \Phi.\ [\![\tilde{P}]\!]_{R(\varphi)}(\varphi) = [\![\tilde{P}]\!]_{\varphi}(\varphi) & \\ \forall \varphi \in \Phi.\ [\![\tilde{P}]\!]_{\varphi}(\varphi) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi}(\varphi) & \text{(By def. of } [\![\tilde{P}]\!]_{\varphi}) \\ \forall \varphi \in \Phi.\ [\![\hat{P}]\!]_{\neg\varphi}(\varphi) = [\![\hat{P}]\!]_{\neg\Phi}(\varphi) & \end{array}$$

Thus, $\forall \varphi \in \Phi.\ [\![\tilde{P}]\!]_{R(\varphi)}(\varphi) \sqsubseteq [\![\hat{P}]\!]_{\neg\Phi}(\varphi)$. Finally, we derive $\forall \varphi \in \Phi.\ [\![\tilde{P}]\!]_{R(\varphi)}(\varphi) \sqsubseteq T(\varphi)$ because $T = [\![\hat{P}]\!]_{\neg\Phi}$.

Next, assuming the loop invariant (8.3) holds, we show the invariant still holds after a single iteration.The following table shows the values of $\overrightarrow{\varphi}_{new}$ and $\hat{s}_{new}$ respectively at the begin of line 22 for each of cases in line 19-21.

Case	$\overrightarrow{\varphi}_{new}$	$\hat{s}_{new}$
$\hat{s} \sqsupset \hat{s}'$	$\bigcup_{\varphi_i \in \mathsf{pred}(\varphi)} R(\varphi_i)$	$\hat{f}(\varphi)(\bigsqcup_{\varphi_i \in \mathsf{pred}(\varphi)} T(\varphi_i))$
$\hat{s} \sqsubseteq \hat{s}'$	$R(\varphi)$	$T(\varphi)$
otherwise	$R(\varphi) \cup \bigcup_{\varphi_i \in \mathsf{pred}(\varphi)} R(\varphi_i)$	$T(\varphi) \sqcap \hat{f}(\varphi)(\bigsqcup_{\varphi_i \in \mathsf{pred}(\varphi)} T(\varphi_i))$

Because $\overrightarrow{\varphi}_{new}$ and $\hat{s}_{new}$ will be assigned to $T(\varphi)$ and $R(\varphi)$ respectively at line 23, now our goal is to show that $[\![\tilde{P}]\!]_{\overrightarrow{\varphi}_{new}}(\varphi) \sqsubseteq \hat{s}_{new}$ in each case.

- Case $\hat{s} \sqsupset \hat{s}'$: Let $R' = \bigcup_{\varphi_i \in \mathsf{pred}(\varphi)} R(\varphi_i)$ and $\hat{H} = \lambda Z.[\![\hat{P}]\!]_{\neg R'} \sqcap \hat{F}(Z)$.

$$\begin{array}{ll} [\![\tilde{P}]\!]_{R'}(\varphi) = \hat{H}([\![\tilde{P}]\!]_{R'})(\varphi) & ([\![\tilde{P}]\!]_{R'} = \mathsf{fix}^{\#}\hat{H}) \\ \sqsubseteq \hat{F}([\![\tilde{P}]\!]_{R'})(\varphi) & (\hat{H} \sqsubseteq \hat{F}) \\ = \hat{f}(\varphi)(\bigsqcup_{\varphi_i \in \mathsf{pred}(\varphi)} [\![\tilde{P}]\!]_{R'}(\varphi_i)) & (\text{By def. of } \hat{F}) \\ \sqsubseteq \hat{f}(\varphi)(\bigsqcup_{\varphi_i \in \mathsf{pred}(\varphi)} [\![\tilde{P}]\!]_{R(\varphi_i)}(\varphi_i)) & (\text{By Lemma 5 and the monotonicity of } \hat{f}(\varphi)) \end{array}$$

  By the inductive hypothesis, $\forall \varphi_i \in \mathsf{pred}(\varphi).\ [\![\tilde{P}]\!]_{R(\varphi)}(\varphi) \sqsubseteq T(\varphi)$. Because $\hat{f}(\varphi)$ is monotone,

$$\hat{f}(\varphi)(\bigsqcup_{\varphi_i \in \mathsf{pred}(\varphi)} [\![\tilde{P}]\!]_{R(\varphi_i)}(\varphi_i)) \sqsubseteq \hat{f}(\varphi)(\bigsqcup_{\varphi_i \in \mathsf{pred}(\varphi)} T(\varphi_i))$$

  Therefore, $[\![\tilde{P}]\!]_{\overrightarrow{\varphi}_{new}}(\varphi) \sqsubseteq \hat{s}_{new}$.

- Case $\hat{s} \sqsubseteq \hat{s}'$: immediate from the inductive hypothesis (8.3).

- Case $\hat{s} \not\sqsupset \hat{s}', \hat{s} \not\sqsubseteq \hat{s}'$:

  Let $R' = \bigcup_{\varphi_i \in \mathsf{pred}(\varphi)} R(\varphi_i)$ and $\hat{s} = \hat{f}(\varphi)(\bigsqcup_{\varphi_i \in \mathsf{pred}(\varphi)} T(\varphi_i))$. So far, we have derived that the followings hold:

$$[\![\tilde{P}]\!]_{R'}(\varphi) \sqsubseteq \hat{s}$$
$$[\![\tilde{P}]\!]_{R(\varphi)}(\varphi) \sqsubseteq T(\varphi)$$

By Lemma 7, $[\![\tilde{P}]\!]_{R(\varphi)\cup R'}(\varphi) \sqsubseteq T(\varphi) \sqcap \hat{s}$. Because $R(\varphi) \cup R' = \overrightarrow{\varphi}_{new}$ and $T(\varphi) \sqcap \hat{s} = \hat{s}_{new}$, we conclude $[\![\tilde{P}]\!]_{\overrightarrow{\varphi}_{new}}(\varphi) \sqsubseteq \hat{s}_{new}$.

At the exit of the loop, $T = [\![\tilde{P}]\!]_\Phi$ by the correctness of the worklist algorithm. On the other hand, $\forall\varphi \in \Phi.\ [\![\tilde{P}]\!]_\Phi(\varphi) \sqsubseteq [\![\tilde{P}]\!]_{R(\varphi)}(\varphi)$ by Lemma 5. Because $\forall\varphi \in \Phi.\ [\![\tilde{P}]\!]_{R(\varphi)}(\varphi) \sqsubseteq T(\varphi) = [\![\tilde{P}]\!]_\Phi(\varphi)$ and $\forall\varphi \in \Phi.\ [\![\tilde{P}]\!]_\Phi(\varphi) \sqsubseteq [\![\tilde{P}]\!]_{R(\varphi)}(\varphi)$, we conclude $\forall\varphi \in \Phi.\ [\![\tilde{P}]\!]_\Phi(\varphi) = [\![\tilde{P}]\!]_{R(\varphi)}(\varphi)$. □

LEMMA 7. *If* $[\![\tilde{P}]\!]_R(\varphi) \sqsubseteq s, [\![\tilde{P}]\!]_{R'}(\varphi) \sqsubseteq s'$ *then* $[\![\tilde{P}]\!]_{R\cup R'}(\varphi) \sqsubseteq s \sqcap s'$.

Proof.

$$\begin{array}{ll} [\![\tilde{P}]\!]_{R\cup R'}(\varphi) \sqsubseteq [\![\tilde{P}]\!]_R(\varphi) \sqsubseteq s & \text{(By Lemma 5)} \\ [\![\tilde{P}]\!]_{R\cup R'}(\varphi) \sqsubseteq [\![\tilde{P}]\!]_{R'}(\varphi) \sqsubseteq s' & \\ [\![\tilde{P}]\!]_{R\cup R'}(\varphi) \sqsubseteq s \sqcap s' & \text{(By definition of glb.)} \end{array}$$

□

THEOREM 6. $\forall\varphi \in \Phi.\ \gamma_{\mathbb{I}}([\![\hat{P}]\!]^{\mathbb{I}}(\varphi)) \ominus \Omega(\varphi, x, y) \sqsubseteq \gamma_{\mathbb{I}}([\![\hat{P}]\!]^{\mathbb{I}}(\varphi) \ominus_{\hat{\mathbb{S}}_{\mathbb{I}}} \hat{\Omega}(\varphi, x, y))$
(Stated in § 3.4.2.)

PROOF. We first show $\gamma_{\mathbb{I}}(\hat{\Omega}(\varphi, x, y)) \subseteq \Omega(\varphi, x, y)$.

- Case $\hat{\Omega}(\varphi, x, y) = \bot_{\hat{\mathbb{S}}_{\mathbb{I}}}$: trivial.
- Case $\hat{\Omega}(\varphi, x, y) = \bot_{\hat{\mathbb{S}}_{\mathbb{I}}}[x \mapsto [y_{max}, +\infty], y \mapsto [-\infty, x_{min}]]$:
  $\forall s \in \gamma_{\mathbb{I}}(\hat{\Omega}(\varphi, x, y)).\ s(x) \geq s(y)$ because $y_{max} \geq x_{min}$.

Therefore, $\hat{\Omega}(\varphi, x, y)$ is an underapproximation of the erroneous states.

Next, we show $\hat{\Omega}(\varphi, x, y)$ is precisely complementable. In other words,

$$\gamma_{\hat{\mathbb{S}}_{\mathbb{I}}}(\hat{\Omega}(\varphi, x, y)) = \wp(\mathbb{S}) \setminus \gamma_{\hat{\mathbb{S}}_{\mathbb{I}}}(\overline{\hat{\Omega}(\varphi, x, y)}).$$

$\gamma_{\hat{\mathbb{S}}_{\mathbb{I}}}(\hat{\Omega}(\varphi, x, y)) = \{x \mapsto n_x, y \mapsto n_y \mid n_x \geq y_{max}, n_y \leq x_{min}\}$
$\gamma_{\hat{\mathbb{S}}_{\mathbb{I}}}(\overline{\hat{\Omega}(\varphi, x, y)}) = \{x \mapsto n_x, y \mapsto n_y, z \mapsto n_z \mid z \in \mathit{Var}, n_z \in \mathbb{Z}, n_x < y_{max}, n_y > x_{min}\}$
$\therefore \gamma_{\hat{\mathbb{S}}_{\mathbb{I}}}(\hat{\Omega}(\varphi, x, y)) = \wp(\mathbb{S}) \setminus \gamma_{\hat{\mathbb{S}}_{\mathbb{I}}}(\overline{\hat{\Omega}(\varphi, x, y)})$

In addition,

$$\forall \varphi \in \Phi.\ \gamma_{\mathbb{I}}(\llbracket \hat{P} \rrbracket(\varphi)) \ominus \Omega(\varphi, x, y) \sqsubseteq \gamma_{\mathbb{I}}(\llbracket \hat{P} \rrbracket(\varphi)) \ominus \gamma_{\mathbb{I}}(\hat{\Omega}(\varphi, x, y)) \quad (\because \gamma_{\mathbb{I}}(\hat{\Omega}(\varphi, x, y)) \subseteq \Omega(\varphi, x, y))$$

By the fact that $\hat{\Omega}(\varphi, x, y)$ is precisely complementable and Theorem 5, the theorem holds.

□

B Progress Graphs

In this appendix, progress graphs are presented. Figure 8.1, 8.2, and 8.3 present the resulting interval, pointer, and octagon analysis progress bars respectively. Dotted diagonal line denotes the ideal progress bar.

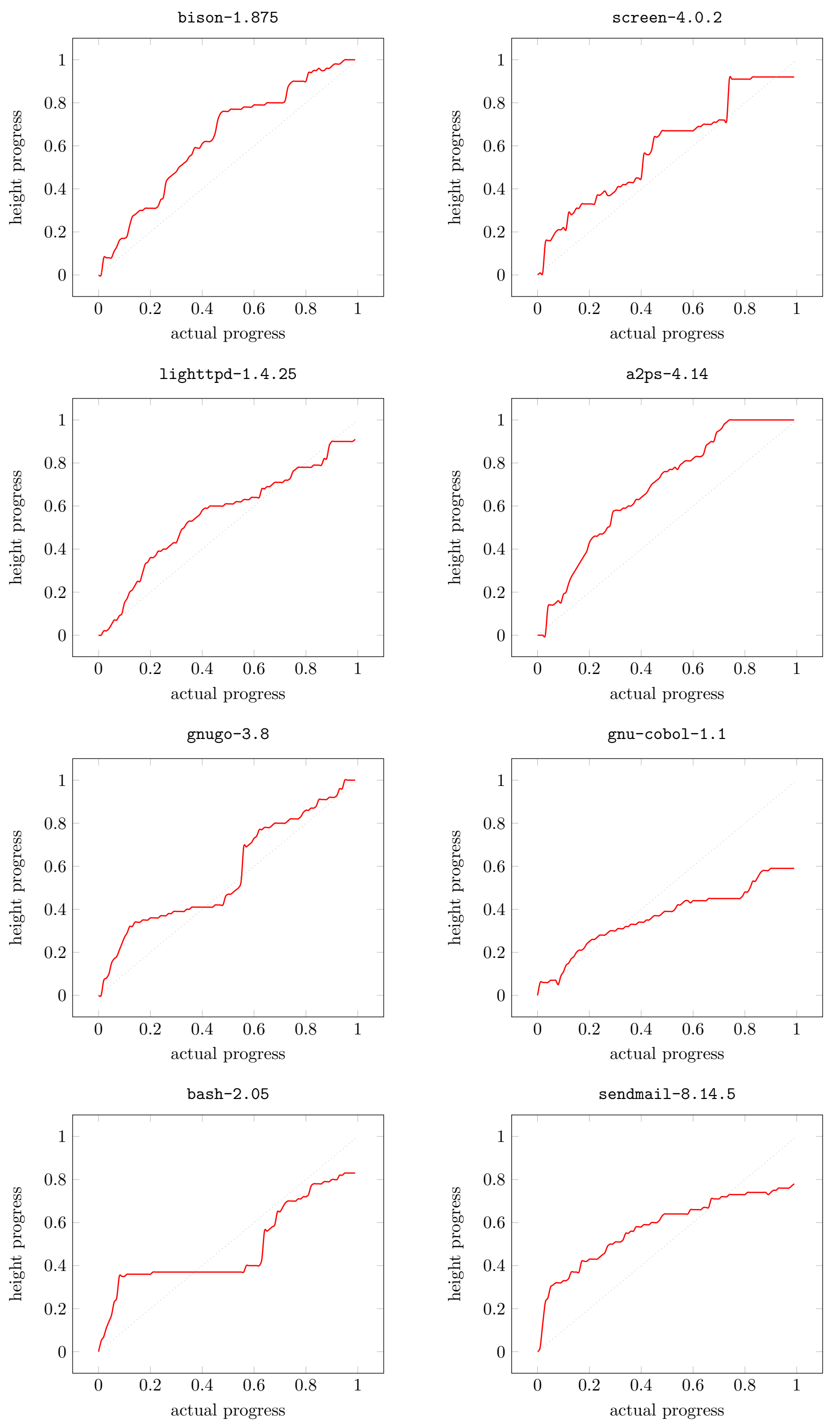


Figure 8.1: Our progress estimation for interval analysis (when *depth* = 1).

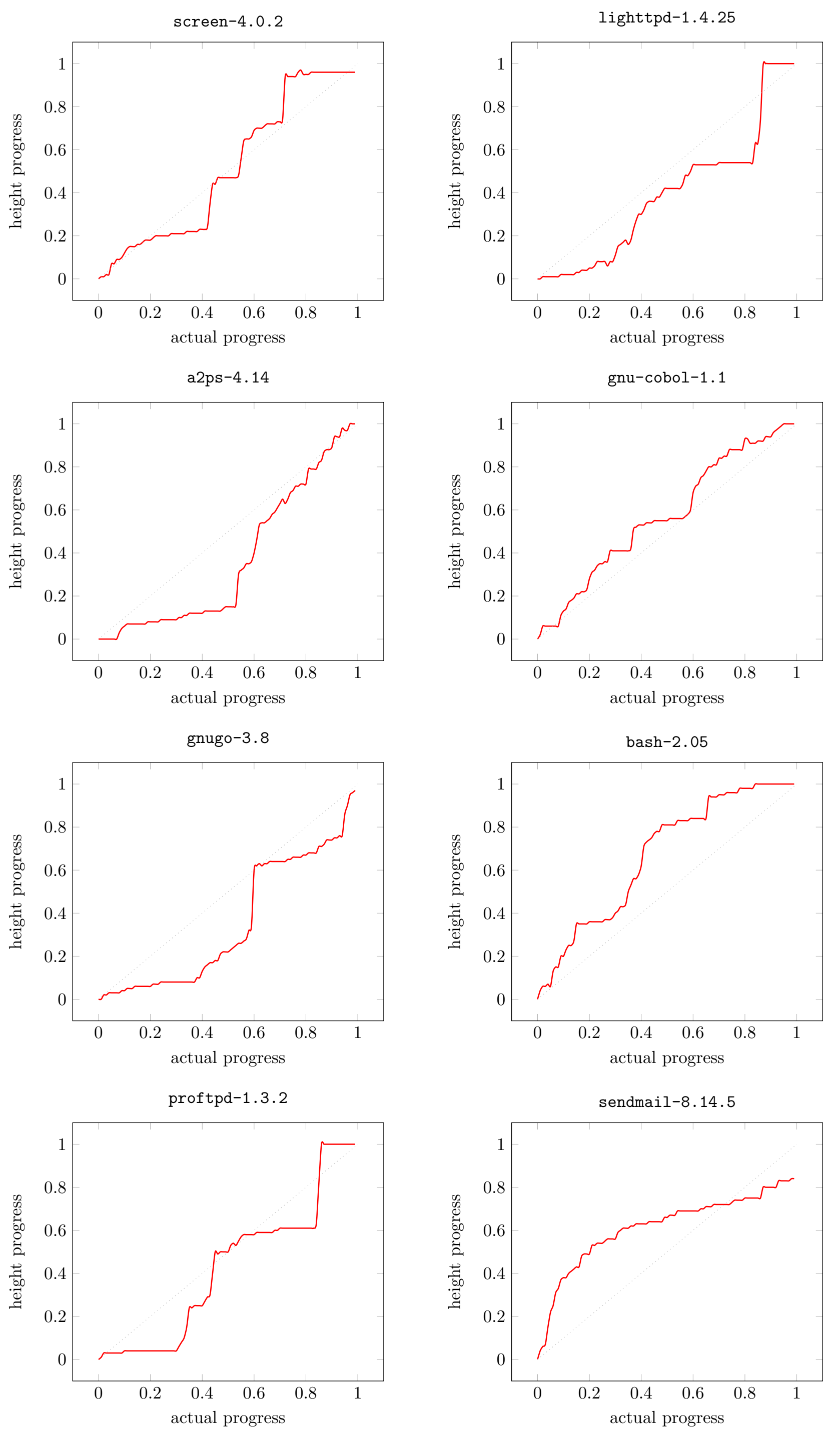


Figure 8.2: Our progress estimation for pointer analysis (when $depth = 1$).

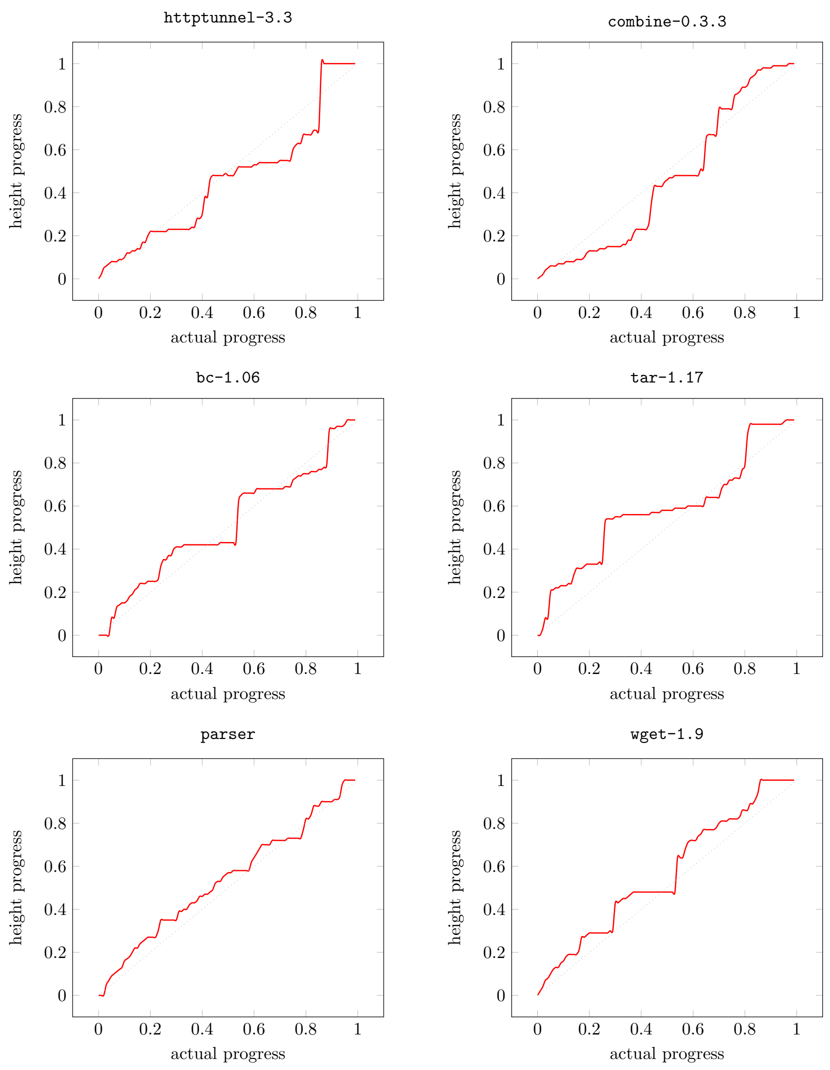


Figure 8.3: Progress estimation for octagon analysis.

C Algorithms for the Pointer Analysis in Secrecy

Fig. 8.4 describes the protocol. Fig. 8.5 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 5.3.2 and 5.4.3. 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. 8.5.

– **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 $\left\{\mathtt{x_i} \longrightarrow \&\mathtt{x_j} \mid \delta_{i,j}^{(\ell)} \ne 0, 1 \le i,j \le m, 1 \le \ell \le n\right\}$.
```

Figure 8.4: 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  $\bar{Z} \leftarrow \bar{A}$;   for $i = 1$ to $\lceil \log k \rceil$ do        $\bar{Z} \leftarrow \mathsf{HE.MatrixMult}\left(\bar{Z}, \bar{Z}\right)$
   2  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.
// In the following functions, $\bar{\mathbf{c}}$ refers to the encryption of $(\mu_1, \cdots, \mu_m)$
HE.Replicate($\bar{\mathbf{c}}, k$)
  return the ciphertext $\mathsf{BGV.Enc}(\mu_k, \cdots, \mu_k)$

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

HE.Rotate($\bar{\mathbf{c}}, k$)
  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 constant vector $\mathbf{r} = (r_1, \cdots, r_m) \in \mathbb{Z}_p \times \cdots \times \mathbb{Z}_p$.
  return the ciphertext $\mathsf{BGV.Enc}(r_1\mu_1, \cdots, r_m\mu_m)$
```

Figure 8.5: Pseudocode for the Homomorphic Matrix Operations

초 록

정적 분석기의 사용자들이 흔히 겪는 세 가지 문제들 - 허위 경보, 진행정도 예측 불가, 대상 프로그램의 저작권 침해 우려 - 각각에 대한 해결책들을 제시한다. 첫 번째로, 분석기가 발생시킬 수 있는 다수의 허위 경보들을 보다 쉽게 걸러낼 수 있는 방법을 제시한다. 이 기술은 같은 발생 원인을 공유하는 경보들을 묶어, 그 중 대표 경보만을 사용자에게 제시함으로써 사용자가 허위여부를 판별해야 하는 경보 숫자를 줄인다. 둘째로, 복잡한 프로그램들에 대해서 분석이 오래 걸림에도 불구하고 진행율을 알 수 없었던 기존 문제에 대한 해결책을 제시한다. 마지막으로, 암호화된 대상 프로그램에 대해 분석을 수행할 수 있는 방법을 제시함으로써 분석 서비스 사용시 발생할 수 있는 저작권 침해 가능성을 차단하는 해결책을 제시한다. 본 논문에서는 위의 기술들을 엄밀히 정의하고 그 기술들이 실제 C 프로그램 분석에서 성공적으로 적용될 수 있음을 실험적으로 보인다.

주요어 : 프로그래밍 언어, 요약 해석, 정적 분석, 사용자 편의성, 동형 암호, 허위 경보, 진행율 예측, 버퍼오버런 탐지, 포인터 분석, 비통계적 클러스터링

학 번 : 2009-20866