Sound Non-Statistical Clustering of Static Analysis Alarms

WOOSUK LEE, Seoul National University
WONCHAN LEE, Stanford University
DONGOK KANG and KIHONG HEO, Seoul National University
HAKJOO OH, Korea University
KWANGKEUN YI, Seoul National University

We present a sound method for clustering alarms from static analyzers. 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. We have implemented our clustering algorithm on top of a realistic buffer-overflow analyzer and proved that our method reduces 45% of alarm reports. Our framework is applicable to any abstract interpretation-based static analysis and orthogonal to abstraction refinements and statistical ranking schemes.

CCS Concepts: • **Theory of computation → Program analysis**; **Abstraction**; • **Software and its engineering → Formal software verification**;

Additional Key Words and Phrases: Static analysis, abstract interpretation, false alarms

ACM Reference format:
Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Sound Non-Statistical Clustering of Static Analysis Alarms. *ACM Trans. Program. Lang. Syst.* 39, 4, Article 16 (August 2017), 35 pages.
https://doi.org/10.1145/3095021

1 INTRODUCTION

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

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. B0717-16-0098 and No. R0190-16-2011, Development of Vulnerability Discovery Technologies for IoT Software Security) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1C1B2014062). This research was also supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning(MSIP) / National Research Foundation of Korea(NRF) (Grant NRF-2008-0062609), and by Samsung Electronics Software Center.
Authors' addresses: W. Lee, D. Kang, K. Heo, and K. Yi, Room 312-2, Building 302, Seoul National University, 1 Kwanak-ro, Kwanak-gu, Seoul 151-744, Korea; emails: {wslee, dokang, khheo, kwang}@ropas.snu.ac.kr; W. Lee, 416 Gates, 353 Serra Mall, Stanford, California 94305, USA; email: wonchan@cs.stanford.edu; H. Oh (Corresponding author), Room 616c, Science Library Bldg, College of Informatics, Korea University, Anam-dong 5-ga, Seongbuk-gu, Seoul 136-713, Korea.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 ACM 0164-0925/2017/08-ART16 $15.00
https://doi.org/10.1145/3095021

commercial software, we have found only one real error in 273 buffer-overflow alarms after tedious and time-consuming alarm investigation efforts [16].

Statistical ranking schemes [16, 20] 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.

1.2 Our Solution

Our solution is to reduce the alarm-investigation burden by clustering alarms according to their sound dependence information. We say that alarm A has (sound) dependence on alarm B whenever 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 article, 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 that have their own dominant alarms. If the dominant alarms turn out to be false (true, respectively), then we can assure that all the others in the same cluster are also false (true, respectively).

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

```
1  void residual(SYSTEM *sys, double *upad, double *r) {
2    nx = 50;
3    u = &upad[nx+2];
4    ...
5    for (k = 0; k < ny; k++) {
6      u++;
7      for(j = 0; j < nx; j++) {
8        r[0] = ac[0]*u[0] - ax[0]*u[-1] - ax[1]*u[1] - ay[0]*u[-nx-2]
9          - ay[nx]*u[nx+2] - q[0];
10       r++; u++; q++; ac++; ax++; ay++;
11     }
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, then 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, then 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 that 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 construct a true-alarm cluster with the same set; the buffer access u[-nx-2] at line 8 is the dominant alarm of the cluster. Thus, 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 reported alarms. On the other hand, finding the access u[-nx-2] unsafe will help to spot other potential vulnerabilities accordingly.

Example 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, arrays invmergerules and invmergerulesnn have the same size 8. The function apply_rule is the only one caller to the other functions lookup_mergearcs and rule_mergearcs.

```
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 that if either one of the alarms is true (or false), so are the others, since all the alarms access the same array with the same index for the following reasons:

(1) There is no update on the value of i between the two accesses at line 3 and 4.
(2) The value of i at line 3 flows to the variable rule at line 10 through function calls and returns (5 → 15 → 16 → 10).

We can find false- and true-alarm clusters in a similar manner as we did in the example 1.1. Instead of inspecting all of the alarms, checking either one of the alarms (e.g., the one at line 3) is sufficient to determine if the other remaining alarms are true or false.

Example 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 that 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 that will be described in this article can be generalized to other languages and safety properties as well, because we are based on a general model of programs and static analyses.

1.4 Contributions

In this article, 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.

This article is an extension of [22]. Compared to the previous version, the current article presents a new clustering algorithm that guarantees to find a minimal set of dominant alarms (Section 4.1), provides a new instance of the framework based on a symbolic domain (Section 5.4), shows experimentally that alarm-clustering with the symbolic domain outperforms the previous octagon-based method in [22], and formally proves the soundness of the proposed alarm-clustering framework and algorithms (Appendix).

2 OVERVIEW

Before formally presenting our alarm clustering approach (Section 3, 4, and 5), we illustrate key aspects of our approach with an example. In this section, we consider a flow-sensitive interval analysis for buffer-overflow detection. However, our method is general and applicable to any trace-partitioning strategy, for example, context-sensitivity. In Section 3, we present our approach in a general setting.

Example Program. Consider the following code snippet:

```
φ1  : int* a = init_array(0); // a.size = [7, 7]
φ2  : int* b = init_array(1); // b.size = [-oo, +oo]
φ3  : if (!*b)
φ4  :   exit (*b);
φ5  : int sum =0;
φ6  : int i = read_int(); // i = [0, +oo]
φ7  : while (*) {
φ8  :   sum += a[i-1];
φ9  :   sum += a[i+2];
φ10 :   sum += a[i-2];
φ11 :   sum += a[i+1]; }
```

The analysis computes interval values for each variable at each program point. Suppose the analysis reports five buffer-overflow alarms: Alarms are underlined, and the values of variables in intervals are annotated in comments. Throughout this section, we will use program point and alarm interchangeably; alarm φ_i means the one at the program point φ_i. Assume that a gets allocated by an array of size 7 at line φ_1 but the analysis cannot precisely infer the size of b at line φ_2, so b gets allocated by an array of size $[-\infty, +\infty]$ during the analysis.

Key Idea. The key idea of our alarm clustering method is what we call *sound refinement by refutation* (Section 3.4); if we can kill an alarm φ_j from the abstract semantics refined under the assumption that alarm φ_i is false, then the falsehood of φ_j is determined by that of φ_i. Suppose alarm φ_9 is false. Then `i` at φ_9 should have interval $[-2, 4]$, because the value of `i+2` should lie in $[0, 6]$. Similarly, suppose alarm φ_{10} is false. Then `i` at φ_{10} should hold $[2, 8]$, because the value of `i-2` should lie in $[0, 6]$. If we re-analyze the program under those assumptions, then the interval value of `i` will be $[2, 4]$ throughout the loop ($\varphi_7 - \varphi_{11}$), which removes the other alarms in the loop (φ_8 and φ_{11}). We can soundly conclude that if the dominant alarms φ_9 and φ_{10} are false, so are the other alarms φ_8 and φ_{11} in the loop. In Section 3.4, we show that, given an abstract domain equipped with a *sound abstract slice operator* used to slice out the erroneous states, our framework provides a sound method to find a small set of dominant alarms.

By varying the abstract domain used for the refinement by refutation, we can have different tradeoffs between the cost and the number of final alarms. Note that, with the interval domain, we cannot find that φ_3 dominates φ_4, because the erroneous state at φ_3 cannot be expressed as intervals as the size of b is unbounded. Therefore, the final alarms in the interval-domain-based clustering will be

$$\{ \varphi_3, \varphi_4, \varphi_9, \varphi_{10} \}.$$

Using a more powerful abstract domain will cluster more alarms. Suppose we use the octagon domain [28] in the refinement phase. Then the non-erroneous state at φ_3 will be expressed as a numerical constraint:

$$0 \leq \texttt{b.offset} \wedge \texttt{b.offset} < \texttt{b.size}.$$

With octagon, we can find the dependency as the falsehood assumption of φ_3 will be propagated and kill φ_4. Therefore, the final alarms will be $\{ \varphi_3, \varphi_9, \varphi_{10} \}$. But this fewer number of final alarms comes with a scalability loss as the octagon analysis is generally more expensive than the interval analysis. In Section 5, we provide designs of three concrete instances of different powers and costs, which are based on interval, octagon, and symbolic domains, respectively.

Alarm Clustering Algorithms. Now we present two algorithms to find dominant alarms. The details of these algorithm will be presented in Section 4. The two algorithms have different tradeoffs between the cost and the number of final alarms. The first algorithm, presented in Section 4.1, guarantees to find a set of *minimal* dominant alarms: the set dominates all alarms and does not contain unnecessaries. However, the algorithm's running time is proportional to the number of total alarms. On the other hand, the algorithm in Section 4.2 quickly finds a dominant alarm set regardless of the number of alarms. Instead, the result may not be minimal. Now, we will describe how the two algorithms based on the interval domain work on the example program.

Minimal Algorithm. This algorithm begins with finding alarms that can be clustered together with other alarms. It re-analyzes the program assuming all of the reported alarms are false. Then we have the following two alarms:

$$\{ \varphi_3, \varphi_4 \}.$$

These two alarms are beyond the power of the interval domain. In other words, we cannot find dependencies involving them as they survive even after refuting all the alarms. Setting aside the

two alarms, it will try to find dependencies among the other four alarms in the loop. To suppress all the alarms in the loop ($\{ \varphi_8, \varphi_9, \varphi_{10}, \varphi_{11} \}$) false, i at the loophead should hold $[2, 4]$, and we will find minimal refutations leading to the interval value. For each alarm, the algorithm refutes all but that alarm and reanalyze the program. The following table shows each refutation and its result.

Refuted alarms	i at φ_7 after re-analysis
$\{ \varphi_9, \varphi_{10}, \varphi_{11} \}$	$[2, 4]$
$\{ \varphi_8, \varphi_{10}, \varphi_{11} \}$	$[2, 5]$
$\{ \varphi_8, \varphi_9, \varphi_{11} \}$	$[1, 4]$
$\{ \varphi_8, \varphi_9, \varphi_{10} \}$	$[2, 4]$

In the second and the third rows, we do not refute φ_9 and φ_{10}, respectively, and we fail to get $[2, 4]$. In the first and the last rows, we do not refute φ_8 and φ_{11}, respectively, but still we get $[2, 4]$. Therefore, refuting φ_9 and φ_{10} is a minimal requirement to suppress all the alarms. With the two alarms beyond the capability of interval, the algorithm reports the following final alarms:

$$\{ \varphi_3, \varphi_4, \varphi_9, \varphi_{10} \}.$$

We explain the algorithm in more detail in Section 4.1. Note that the algorithm requires to run the analysis multiple times.

Non-minimal But Efficient Algorithm. We also present more efficient algorithm that finds a subset of all alarm dependencies in a single fixpoint computation (Section 4.2). The idea is to run the analysis after refuting all alarms and track which alarm's falsehood assumption kills which alarm. After slicing out erroneous states at each program point, the refined states will be propagated through the program by the narrowing operation. First, we ignore φ_3 and φ_4, since the erroneous states are beyond the capability of interval. At φ_8, i holds $[1, 7]$ by assuming alarm φ_8 false. We record the fact that refuting alarm φ_8 contributes to the current value of i. This refined state is propagated further. At φ_9, i initially holds $[-2, 4]$. We conjoin the incoming state from φ_8 and this value obtaining the following result:

$$[1, 7] \sqcap [-2, 4] = [1, 4].$$

We record that refuting alarms φ_8 and φ_9 contribute to the current value of i. At φ_{10}, i initially holds $[2, 8]$. We conjoin the incoming state from φ_{10} and this value obtaining the following result:

$$[1, 4] \sqcap [2, 8] = [2, 4].$$

We record alarms refuting φ_8, φ_9, and φ_{10} contribute to the current value of i. At φ_{11}, i initially holds $[-1, 5]$. Conjoining this state with the incoming state from φ_{10} does not result in a narrowed state, since $[-1, 5] \sqcap [2, 4] = [2, 4]$. We do not add φ_{11} to the list of refuted alarms that contribute to the current value of i. After analyzing the loop once again, i holds $[2, 4]$ at every program points in the loop and the fixpoint is reached. With the new fixpoint, we have the alarms $\{ \varphi_3, \varphi_4 \}$. In addition to this set, we additionally report the dominant alarms. We know that alarms φ_8, φ_9, and φ_{10} dominate φ_{11}. The final alarms will be

$$\{ \varphi_3, \varphi_4, \varphi_8, \varphi_9, \varphi_{10} \}.$$

Note that alarm φ_8 is additionally reported compared to the minimal algorithm. When analyzing φ_8, we do not know in advance that the refutations of φ_9 and φ_{10} will completely eclipse the effect of refuting φ_8. For this reason, the algorithm may report redundant dominant alarms.

3 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. The input to the framework is a static analyzer that has two assumptions: (1) we assume that the analyzer is defined with a trace-partitioning function δ and (2) the abstract domain of the analyzer comes with a meet operator and a sound abstract slice operator. These requirements will be explained in Sections 3.1 and 3.4, respectively.

3.1 Static Analysis

We first define a class of static analyses that we consider in this article. The analysis is used to prove safety properties about programs. It is defined by abstract interpretation of trace semantics based on the trace partitioning [26]. We begin with basic notions used in this article.

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, then 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, then we write $\tau \preceq \sigma$.

We say a sequence σ is a *trace* if σ is a (partial) execution sequence, that is, $\sigma_0 \in \mathbb{S}_\iota \wedge \forall k. \sigma_k \rightarrow \sigma_{k+1}$. 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:

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

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

Abstract Semantics. The class of static analyzers that this article considers is obtained 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 that are maps from a pre-defined set, called "partitioning indices" (e.g., program points) Φ to the set of concrete states. 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 $\hat{\mathbb{D}} = \Phi \rightarrow \hat{\mathbb{S}}$. The overall abstraction is formalized by the following two-step Galois-connection:

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

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 partitioning indices and a partitioning function

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

which maps each partitioning index (Φ) to a set of traces. We assume that the partitioning function is well formed in a sense that it covers all the traces, that is,

$$\bigcup_{\varphi \in \Phi} \delta(\varphi) = \mathbb{S}^+$$

and all the associated sets are disjoint, that is,

$$\forall \varphi_1, \varphi_2. \varphi_1 \neq \varphi_2 \Rightarrow \delta(\varphi_1) \cap \delta(\varphi_2) = \emptyset.$$

Example 3.1. The most popular strategy for partitioning is the so-called flow-sensitivity that partitions the set of traces based on the program points of the final states. When a state is a pair of program point ($\mathbb{C}$) and a memory state ($\mathbb{M}$), that is, $\mathbb{S} = \mathbb{C} \times \mathbb{M}$, this final program point partitioning is defined by the partitioning function $\delta_p(c) = \{\sigma \mid \exists m.\, \sigma_{\dashv} = (c, m)\}$; 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:

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

We write $[\![P]\!]_{/\delta}$ for the concrete semantics $[\![P]\!]$ modulo the partitioning abstraction by δ, that is, $[\![P]\!]_{/\delta} \in \Phi \to \wp(\mathbb{S})$.

(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 $\mathsf{lfp}^{\#}$ is a sound, abstract post-fixpoint operator and the function $\hat{F} : \hat{\mathbb{D}} \to \hat{\mathbb{D}}$ is a monotone or an extensive abstract transfer function such that $\alpha \circ F_P \sqsubseteq \hat{F} \circ \alpha$. The soundness of the static analysis follows from the fixpoint transfer theorem [8].

3.2 Alarm Dependencies

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, that is,

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

In the rest of the article, 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, that is, $[\![P]\!]_{/\delta}(\varphi) \cap \Omega(\varphi) \neq \varnothing$, the alarm is true.

Alarm Dependencies. 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, that is,

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

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

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

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, then it means that φ_2 has concrete dependence on φ_1.

We will describe a simple example that conveys the idea.

Example 3.2 (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, then so is φ_2. Also, if φ_2 is a true alarm, then so is φ_1. The soundness is guaranteed by our alarm clustering framework.

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.

3.3 Computing Alarm Dependencies

Refinement by Refutation. Our key idea for computing the alarm dependence is refinement by refutation; we refine the original fixpoint with the assumption that an alarm is false and then propagate that information to see which other alarms are filtered out as the consequence of the refinement.

Our alarm clustering framework requires the following:

- $[\![\hat{P}]\!] : \Phi \to \hat{\mathbb{S}}$: the abstract semantics of program P, that is, the analysis result.
- $\hat{\Omega} : \Phi \to \hat{\mathbb{S}}$, an underapproximation of the erroneous states, that is,

  $$\forall \varphi \in \Phi.\ \hat{\Omega}(\varphi) \sqsubseteq \alpha_S(\Omega(\varphi)),$$

  where $\Omega : \Phi \to \wp(\mathbb{S})$ specifies erroneous states at each partitioning index.
- $\hat{\ominus} : \hat{\mathbb{S}} \times \hat{\mathbb{S}} \to \hat{\mathbb{S}}$: an abstract slice operator such that it is sound with respect to the concrete slicing:

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

where $\ominus : \wp(\mathbb{S}) \times \wp(\mathbb{S}) \to \wp(\mathbb{S})$ is the concrete slicing operator defined as the set difference, that is, $S_1 \ominus S_2 = S_1 \setminus S_2$. We require that the abstract domain $\hat{\mathbb{S}}$ comes with a meet operator ($\sqcap$) and a sound abstract slice operator ($\hat{\ominus}$). In Section 5, we describe abstract slice operators of the interval, octagon, and symbolic domains.

Given an alarm φ, our alarm clustering method works in the following three steps:

(1) We slice out the erroneous states at φ from the original fixpoint $[\![\hat{P}]\!]$:

$$[\![\hat{P}]\!]_{\neg\varphi} = [\![\hat{P}]\!][\varphi \mapsto [\![\hat{P}]\!](\varphi) \,\hat{\ominus}\, \hat{\Omega}(\varphi)].$$

Here, $[\![\hat{P}]\!]_{\neg\varphi}$ denotes the resulting sliced abstract semantics, which is the same as the original fixpoint $[\![\hat{P}]\!]$ except that an underapproximation of the erroneous states at partitioning index φ is sliced out. This step corresponds to assuming that the alarm φ is false.

(2) Next, we propagate the refined information through the program. This is done by computing the following "narrowing" operation with the abstract semantic function $\hat{F}$ of the target program:

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

where $\mathsf{fix}^{\#}$ is a fixpoint operator. $[\![\tilde{P}]\!]_{\varphi}$ denotes the final analysis result where the information about φ being false is propagated along the entire program.

(3) We conclude that alarms that disappear from $[\![\tilde{P}]\!]_{\varphi}$ has abstract alarm dependence on φ. This step will be formalized shortly (Definition 1).

Example 3.3. Consider the program in Example 3.2. The abstract value of `i` at φ_1 from the original fixpoint $[\![\hat{P}]\!]$ is $[0, +\infty]$:

$$[\![\hat{P}]\!](\varphi_1)(\mathtt{i}) = [0, +\infty].$$

Under the assumption that alarm φ_1 is false, an underapproximation of the erroneous state satisfies the following:

$$\hat{\Omega}(\varphi_1)(\mathtt{i}) = [21, +\infty].$$

Here the slice operator $\hat{\ominus}$ simply rules out the erroneous interval from the original one for each variable in the abstract state:

$$\Big([\![\hat{P}]\!](\varphi_1) \hat{\ominus} \hat{\Omega}(\varphi_1)\Big)(\mathtt{i}) = [0, 20].$$

After slicing out the erroneous state, the sliced abstract semantics satisfies the following (Step 1):

$$[\![\hat{P}]\!]_{\neg\varphi_1}(\varphi_1)(\mathtt{i}) = [\![\hat{P}]\!][\varphi_1 \mapsto [\![\hat{P}]\!](\varphi_1) \,\hat{\ominus}\, \hat{\Omega}(\varphi_1)](\varphi_i)(\mathtt{i}) = [0, 20].$$

The refined state is propagated through the program by the narrowing operation and then the abstract value of `j` at φ_3 after the refinement is as follows (Step 2):

$$[\![\tilde{P}]\!]_{\varphi_1}(\varphi_3)(\mathtt{j}) = [0, 10].$$

Finally we observe that the alarm at φ_3 disappears by assuming the alarm at φ_1 to be false (Step 3).

It is easy to extend this refinement algorithm 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 $[\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} = \sqcap_{\varphi_i \in \overrightarrow{\varphi}} [\![\hat{P}]\!]_{\neg\varphi_i}$.

Abstract Alarm Dependence. We now define abstract alarm dependence based on the refinement by refutation. The dependence between alarm φ_1 and φ_2, written as $\varphi_1 \rightsquigarrow \varphi_2$ 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; that is,

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

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. We show the refinement by refutation of alarm φ_1 (i.e., $[\![\tilde{P}]\!]_{\varphi_1}$) still soundly approximates the concrete semantics (i.e., $\alpha([\![P]\!]) \sqsubseteq [\![\tilde{P}]\!]_{\varphi_1}$) if alarm φ_1 is false. Then we can conclude alarm φ_2 if the refinement removes alarm φ_2, because the refinement is sound with respect to the concrete semantics. We prove the lemma by induction and the soundness of abstract slice operator. The details are available in the appendix. □

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 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. The proof is similar to the proof of Lemma 1 except that we refute multiple alarms. The details are available in the 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.

3.4 Alarm Clustering

Alarm Cluster. Using abstract alarm dependencies, we can build false- and true-alarm clusters. Suppose that we are given a set of dominant alarms $\overrightarrow{\varphi}$ (how to choose such dominant alarms will be discussed in the next section), 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 $C^F_{\overrightarrow{\varphi}} \subseteq \mathcal{A}$ 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 $C^F_{\overrightarrow{\varphi}}$ 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 $C^T_{\varphi} \subseteq \mathcal{A}$ with its dominant alarm 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 example, if the alarm at line 9 is true in Example 1.3, our framework just guarantees that one of the alarms at line 3 or 4 is 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 C^T_{φ} 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 article, a cluster $C_{\overrightarrow{\varphi}}$ means a false-alarm cluster $C^F_{\overrightarrow{\varphi}}$.

3.5 Final Alarm Report

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 ($\overrightarrow{\varphi}$):

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

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

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

Ungroupable alarms exist because (i) the power of the underlying abstract domain of the clustering analysis is not sufficient to detect alarm dependencies for them or (ii) abstract slice operator is imprecise. For example, suppose that analysis developer specifies abstract slice operator does not slice out any abstract states. Although such operator is sound, every alarm would be ungroupable with the operator.

Given a set of alarms $\overrightarrow{\varphi}$ that dominates all groupable alarms (i.e., $C_{\overrightarrow{\varphi}} = \mathcal{G}$), the final alarm reports that users have to examine is as follows:

$$\overrightarrow{\varphi} \cup \mathcal{U}. \tag{1}$$

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 ($\mathcal{U}$).

Example 3.4 (Final alarm report). Suppose we cluster alarms in the following example using the interval domain.

```
// a.size = [10, 10] and i = [0, +oo]
φ1 : a[i] = ...;
φ2 : ... = a[i];

// b.size = [10, 10] and j = [0, +oo]
φ3 : b[j] = ...;
φ4 : ... = b[j];

// c.size = [10, +oo] and k = [0, +oo]
φ5 : c[k] = ...;
```

Alarms $\varphi_1, \varphi_2, \varphi_3$, and φ_4 are groupable, because

$$C_{\varphi_1} = \{\varphi_1, \varphi_2\} \qquad C_{\varphi_3} = \{\varphi_3, \varphi_4\}.$$

On the other hand, the remaining alarm φ_5 is ungroupable, since the alarm is not dominated even by itself. Because both the value of `c.size` and `k` involve $+\infty$, the alarm cannot be soundly refuted using the interval domain. If we use richer domains such as the octagon that can express linear inequalities, then φ_5 can be refuted as `k < c.size`, so is groupable.

In this example, it is sufficient for users to inspect φ_5, which is ungroupable, and φ_1, φ_3, which dominates all groupable ones (i.e., $C_{\varphi_1,\varphi_3} = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\} = \mathcal{G}$). The example suggests that although there are multiple clusters, each of which owns its dominant alarms, user only has to inspect dominant alarms of the largest cluster that comprises all groupable alarms.

4 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.

We present two algorithms, which have different tradeoffs between the cost and the number of final alarm reports. The first algorithm, presented in Section 4.1, 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 4.2 quickly finds a dominant alarm set regardless of the number of alarms. Instead, the set found is not guaranteed to be minimal.

4.1 Algorithm 1: Finding Minimal Dominant Alarms

The first algorithm finds minimal dominant alarms so that we can minimize the number of final alarms (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, that is, $C_{\overrightarrow{\varphi}} = \mathcal{G}$, and
(2) $\overrightarrow{\varphi}$ is a minimal such set, that is, $\forall \overrightarrow{\varphi}' \subseteq \mathcal{A}.\, C_{\overrightarrow{\varphi}'} = \mathcal{G} \wedge \overrightarrow{\varphi} \subseteq \overrightarrow{\varphi}' \Rightarrow \overrightarrow{\varphi} = \overrightarrow{\varphi}'$.

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

ALGORITHM 1: Algorithm for Finding Groupable and Ungroupable Alarms.

```
1:  procedure CATEGORIZE([[P̂]], 𝒜)
2:      ⟨𝒰, 𝒢⟩ := ⟨∅, ∅⟩                          ▷ ungroupable and groupable alarms
3:      for all c ∈ 𝒜 do
4:          if γ_S([[P̃]]_𝒜(c)) ∩ Ω(c) ≠ ∅ then
5:              𝒰 := 𝒰 ∪ {c}
6:          end if
7:      end for
8:      𝒢 := 𝒜 − 𝒰
9:      return ⟨𝒰, 𝒢⟩
10: end procedure
```

Basic Algorithm. We utilize existing algorithms that are initially developed for finding minimal abstractions [23]. They proposed algorithm ScanCoarsen to find a program abstraction that is 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 $\mathbf{F} : \wp(\mathcal{A}) \to \{\mathbf{0}, \mathbf{1}\}$ be the clustering analysis defined as follows:

$$\mathbf{F}(\overrightarrow{\varphi}) = (C_{\overrightarrow{\varphi}} = \mathcal{G}),$$

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 [23]:

Lemma 3. $\overrightarrow{\varphi} \subseteq \overrightarrow{\varphi}' \Rightarrow C_{\overrightarrow{\varphi}} \subseteq C_{\overrightarrow{\varphi}'}$.

Proof. Available in the appendix. □

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

Our goal is to find a minimal $\overrightarrow{\varphi}$ such that $\mathbf{F}(\overrightarrow{\varphi}) = \mathbf{1}$. 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. $\mathcal{U} = \{\varphi \in \mathcal{A} \mid \varphi \notin C_{\mathcal{A}}\}$.

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 $C_{\mathcal{A}}$. The groupable alarms are computed simply by $\mathcal{G} = \mathcal{A} \setminus \mathcal{U}$. 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, then we put 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 [23]. □

ALGORITHM 2: Clustering via Scanning

```
procedure ScanCluster(L, U)
    If L = U then return U
    end if
    choose φ ∈ U \ L
    if F(U \ {φ}) = 1 then                          ▷ try removing φ
        return ScanCluster(L, U − {φ})              ▷ φ is not necessary
    else
        return ScanCluster(L ∪ {φ}, U)              ▷ φ is necessary
    end if
end procedure
```

Example 4.1 (Minimal Algorithm). Consider the following code, which is a simplified version of the example in Section 2, and suppose an interval domain-based analyzer reports a set of alarms $\mathcal{A} = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$, because of the unknown input before the loop.

```
// a.size=7
sum = 0;
i = read(); // i = [−∞, +∞]
while (...) {
φ1 : sum += a[i-1];
φ2 : sum += a[i+2];
φ3 : sum += a[i-2];
φ4 : sum += a[i+1];
}
```

The minimal algorithm begins with refuting all alarms. We find $C_{\mathcal{A}} = \mathcal{A}$, because it suppresses all alarms for the following reasoning. First, we slice out the erroneous states for each alarm. The values of i at each alarm point are as follows:

$$
\begin{aligned}
[\![\hat{P}]\!]_{\neg\varphi_1}(\varphi_1)(\mathtt{i}) &= [1, 7] \\
[\![\hat{P}]\!]_{\neg\varphi_2}(\varphi_2)(\mathtt{i}) &= [-2, 4] \\
[\![\hat{P}]\!]_{\neg\varphi_3}(\varphi_3)(\mathtt{i}) &= [2, 8] \\
[\![\hat{P}]\!]_{\neg\varphi_4}(\varphi_4)(\mathtt{i}) &= [-1, 5].
\end{aligned}
$$

Next, we propagate the refined states through the program and identify the following invariant:

$$\forall \varphi \in \mathcal{A}.\ [\![\tilde{P}]\!]_{\mathcal{A}}(\varphi)(\mathtt{i}) = [2, 4].$$

Finally, all the alarms disappear with $[\![\tilde{P}]\!]_{\mathcal{A}}$ that implies $C_{\mathcal{A}} = \mathcal{A}$.

Now the algorithm removes dominant alarms one by one to remove redundant ones. The following table represents each iteration of procedure ScanCluster in Algorithm 2.

iter	L	U	φ	$\mathbf{F}(U \setminus \{\varphi\})$
1	$\emptyset$	$\{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$	φ_1	1
2	$\emptyset$	$\{\varphi_2, \varphi_3, \varphi_4\}$	φ_2	0
3	$\{\varphi_2\}$	$\{\varphi_2, \varphi_3, \varphi_4\}$	φ_3	0
4	$\{\varphi_2, \varphi_3\}$	$\{\varphi_2, \varphi_3, \varphi_4\}$	φ_4	1
5	$\{\varphi_2, \varphi_3\}$	$\{\varphi_2, \varphi_3\}$	—	—

The algorithm ends with $L = U = \{\varphi_2, \varphi_3\}$. Thus, we conclude φ_2 and φ_3 dominate all alarms (i.e., $C_{\{\varphi_2, \varphi_3\}} = \mathcal{A}$).

There is also another method ActiveCoarsen that applies randomization into ScanCoarsen [23], but the algorithm is not effective in our case. 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. However, it is effective only if a small subset of alarms matters for clustering all groupable alarms. In other words, minimal dominant alarms should be sparse. In ActiveCoarsen, the expected number of calls to **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, then the number of calls becomes close to $O(|\mathcal{A}| \log |\mathcal{A}|)$, which is greater than $|\mathcal{A}|$ calls to **F** in ScanCluster. For this reason, our clustering algorithm is only based on ScanCoarsen.

Further Optimization. We further improve ScanCluster by considering only *refutable* alarms candidates of dominant alarms. Let R be the set of refutable alarms (Let $\hat{T} \in \Phi \to \hat{\mathbb{S}}$ be the analysis result):

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

We say an alarm φ is refutable if some erroneous states at φ can be sliced out in the underlying abstract domain. It means that only refutable alarms have possibilities to dominate other alarms. Therefore, we exclude non-refutable alarms from the initial set of alarms ($\mathcal{A}$) in running ScanCluster. That is, we run ScanCluster($\emptyset$, R) instead of ScanCluster($\emptyset$, $\mathcal{A}$). Note that refutable alarms are independent of 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 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., $\hat{T}(\varphi) \mathbin{\hat{\ominus}} \hat{\Omega}(\varphi) = \hat{T}(\varphi)$), then φ is not included in any set of minimal dominant alarms.*

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

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 1.5 times (1/0.68) faster than non-optimized.

4.2 Algorithm 2: Non-Minimal But Efficient

In this section, we present a more appropriate clustering algorithm in case we have limited time budgets. This algorithm is more efficient than the other one as it finds a subset of all abstract alarm dependencies by a single fixpoint computation. By contrast, the algorithm in Section 4.1 requires us to run the analysis multiple times. 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 that 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 on which the algorithm is based. We assume that a program is represented by a control-flow graph. Φ is the set of nodes (or program points) and every node

ALGORITHM 3: Clustering algorithm

1: $w \in Work = \Phi \quad W \in Worklist = 2^{Work}$
2: pred $\in Predecessors = \Phi \to 2^{\Phi}$
3: succ $\in Successors = \Phi \to 2^{\Phi}$
4: $\hat{f} \in \Phi \to \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ ▷ *abstract transfer function for each program point*
5: $T \in Table = \Phi \to \hat{\mathbb{S}}$ ▷ *abstract state indexed by program point*
6: $\overrightarrow{\varphi} \in DomCand = 2^{\Phi}$ ▷ *dominant alarm candidate. set of alarms.*
7: $R \in RefinedBy = \Phi \to DomCand$ ▷ $\{\varphi \mapsto \overrightarrow{\varphi}\} \in R : T(\varphi)$*is refined by* $\overrightarrow{\varphi}$
8: $\hat{\Omega} \in ErrorInfo = \Phi \to \hat{\mathbb{S}}$ ▷ *abstract erroneous state information*
9: $C \in Clusters = DomCand \to 2^{\Phi}$ ▷ *alarm clusters indexed by dominant alarms*
10: **procedure** FixpointIterate(W, T, R)
11: **repeat**
12: φ := choose(W) ▷ *pick a work from worklist*
13: $\hat{s} := T(\varphi)$ ▷ *previous abstract state*
14: $\hat{s}' := \hat{f}(\varphi)(\bigsqcup_{\varphi_i \in \text{pred}(\varphi)} T(\varphi_i))$ ▷ *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: $\overrightarrow{\varphi}' := \bigcup_{\varphi_i \in \text{pred}(\varphi)} R(\varphi_i)$ ▷ *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: $W := W \cup \text{succ}(\varphi); T(\varphi) := \hat{s}_{new}; R(\varphi) := \overrightarrow{\varphi}_{new}$
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: $C := C\{R(\varphi) \mapsto C(R(\varphi)) \cup \{\varphi\}\}$
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)

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:

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

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; that is, 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, then 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, and, 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 lines 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, then it means that the abstract state at φ is refined by some dominant alarm candidate $\overrightarrow{\varphi}$, and 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).

Alongside the usual fixpoint computation, we iteratively compute the information R of dominant alarm candidates. At line 17, we store the previous information of R at φ in $\overrightarrow{\varphi}$. At line 18, we update that information as follows:

$$\overrightarrow{\varphi}' = \bigcup_{\varphi_i \in \mathsf{pred}(\varphi)} R(\varphi_i).$$

That is, if φ_i is a predecessor of φ on the control-flow graph and φ_i is dominated by $R(\varphi_i)$, then φ is also dominated by $R(\varphi_i)$. 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), then $\hat{s}_{new}$ 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), then 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, then 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 dependencies.*

Proof. We show that $\forall \varphi \in \Phi.\, T(\varphi) = [\![\tilde{P}]\!]_{R(\varphi)}(\varphi)$ at line 27. Then abstract dependence $R(\varphi) \rightsquigarrow \varphi$ added at line 28 is sound as it is found only if $[\![\tilde{P}]\!]_{R(\varphi)}(\varphi) \sqcap \hat{\Omega}(\varphi) = \bot$. The details are available in the appendix. □

Example 4.2 (Heuristic Algorithm). Consider the same code in Example 4.1. The following table represents each iteration of procedure FixpointIterate in Algorithm 3. We begin with analyzing φ_1.

iter	φ	$\hat{s}(\mathtt{i})$	$\hat{s}'(\mathtt{i})$	$\hat{s}_{new}(\mathtt{i})$	$\overrightarrow{\varphi}$	$\overrightarrow{\varphi}'$	$\overrightarrow{\varphi}_{new}$
1	φ_1	$[1,7]$	$[1,7]$	$[1,7]$	φ_1	φ_1	φ_1
2	φ_2	$[-2,4]$	$[1,7]$	$[1,4]$	φ_2	φ_1	φ_1, φ_2
3	φ_3	$[2,8]$	$[1,4]$	$[2,4]$	φ_3	φ_1, φ_2	$\varphi_1, \varphi_2, \varphi_3$
4	φ_4	$[-1,5]$	$[2,4]$	$[2,4]$	φ_4	$\varphi_1, \varphi_2, \varphi_3$	$\varphi_1, \varphi_2, \varphi_3$
5	φ_1	$[1,7]$	$[2,4]$	$[2,4]$	φ_1	$\varphi_1, \varphi_2, \varphi_3$	$\varphi_1, \varphi_2, \varphi_3$
6	φ_2	$[1,4]$	$[2,4]$	$[2,4]$	φ_1, φ_2	$\varphi_1, \varphi_2, \varphi_3$	$\varphi_1, \varphi_2, \varphi_3$
7	φ_3	$[2,4]$	$[2,4]$	$[2,4]$	$\varphi_1, \varphi_2, \varphi_3$	$\varphi_1, \varphi_2, \varphi_3$	$\varphi_1, \varphi_2, \varphi_3$

Finally, this algorithm reports $\{\varphi_1, \varphi_2, \varphi_3\}$ as dominant alarms, that is, $C_{\{\varphi_1, \varphi_2, \varphi_3\}} = \mathcal{A}$. Note that the heuristic algorithm computes more dominant alarms than the minimal algorithm in Example 4.1. But the heuristic algorithm each alarm node is visited twice during analysis whereas each alarm node is visited 4 times in the minimal algorithm.

5 INSTANCES

In this section, we show how to use our framework to design alarm clustering 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 [32]. The key component we have to define to use our framework is the abstract slice operator described in Section 3.

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* [10] 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 $\wp(\mathbb{S}) \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \hat{\mathbb{S}}$. 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 $\wp(\mathbb{S}) \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \hat{\mathbb{S}}$, 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.

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

□

5.1 Setting: Baseline Analyzer

Now we describe a baseline analyzer Sparrow [32] on which our clustering methods are implemented. The analyzer is a realistic buffer-overflow detector performing sound and inter-procedural analysis. Sparrow basically performs a flow-sensitive and context-insensitive analysis with the interval abstract domain. Sparrow performs a sparse analysis [29, 30] that scales to analyze up to one million lines of C programs.

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 Representation. We assume that a program is represented by a control-flow graph. Each command in a node (or program point) $\varphi \in \Phi$ in the graph has one of the following command, denoted $\mathsf{cmd}(\varphi)$:

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

An (side-effect-free) expression is either constant integer (n), binary operation ($e + e$), or variable (x). The command $\mathsf{x} := e$ assigns the value of e into x. The command $\{\!\{\mathsf{x} \leq \mathsf{n}\}\!\}$ makes the program continue only when the condition evaluates to true. The command $\mathsf{x} := \mathsf{unknown}()$ assigns an arbitrary integer into x. Edges are assembled by function $\mathsf{predof} \in \Phi \rightarrow 2^{\Phi}$, which maps each node to its predecessors.

Collecting Semantics. Collecting semantics of a program P is an invariant $[\![P]\!]_{/\delta} : \Phi \rightarrow \wp(\mathbb{S})$, where δ is the final program point partitioning function described in Section 3. 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 ($\hat{\mathbb{S}}_{\mathbb{I}} = Var \overset{\text{fin}}{\rightarrow} \mathbb{I}$), 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 $\hat{f}_{\mathbb{I}} : \Phi \to \hat{\mathbb{S}}_{\mathbb{I}} \to \hat{\mathbb{S}}_{\mathbb{I}}$ that, given an input memory state, computes the effect of the assignment in the node on the input state:

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

The effect of node $\{\{\mathsf{x} \leq \mathsf{n}\}\}$ is to confine the interval value of x according to the condition. The effect of node $\mathsf{x} := e$ is to assign the abstract value of e into variable x. The effect of node $\mathsf{x} := \mathsf{unknown}()$ 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:

$$\hat{\mathcal{V}}(e)\ :\ \hat{\mathbb{S}}_{\mathbb{I}} \to \hat{Val}$$

$$\begin{aligned} \hat{\mathcal{V}}(\mathsf{n})(\hat{m}) &= [n, n] \\ \hat{\mathcal{V}}(e_1 + e_2)(\hat{m}) &= \hat{\mathcal{V}}(e_1)(\hat{m}) \hat{+} \hat{\mathcal{V}}(e_2)(\hat{m}) \\ \hat{\mathcal{V}}(\mathsf{x})(\hat{m}) &= \hat{m}(x). \end{aligned}$$

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

The analyzer computes a fixpoint table $[\![\hat{P}]\!]^{\mathbb{I}} \in \Phi \to \hat{\mathbb{S}}_{\mathbb{I}}$ 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:

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

The fixpoint table $[\![\hat{P}]\!]^{\mathbb{I}}$ is a sound approximation of the collecting semantics of the program, that is, $\forall\varphi \in \Phi.\gamma_{\mathbb{I}}([\![\hat{P}]\!]^{\mathbb{I}}(\varphi)) \sqsupseteq [\![P]\!]_{/\delta}(\varphi)$.

Alarms. We define erroneous states and alarms of the static analysis. We assume queries, triples in $Q \subseteq \Phi \times Var \times Var$, 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:

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

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

5.2 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:

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

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:

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

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 $\hat{\Omega}(\varphi, x, y)$:

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

Example 5.1. Consider the following code. The code is simply adapted from Example 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,

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

The variable sz refers to the size of cboard and ephash in Example 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:

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

The following are the underapproximation of the erroneous states:

$$\hat{\Omega}(\varphi_2, \mathtt{f}, \mathtt{sz}) = \bot_{\hat{\mathbb{S}}_{\mathbb{I}}}[\mathtt{f} \mapsto [64, +\infty], \mathtt{sz} \mapsto [-\infty, 63]]$$
$$\hat{\Omega}(\varphi_3, \mathtt{t}, \mathtt{sz}) = \bot_{\hat{\mathbb{S}}_{\mathbb{I}}}[\mathtt{t} \mapsto [64, +\infty], \mathtt{sz} \mapsto [-\infty, 63]],$$

and the following are the precise complements:

$$\overline{\hat{\Omega}(\varphi_2, \mathtt{f}, \mathtt{sz})} = \top_{\hat{\mathbb{S}}_{\mathbb{I}}}[\mathtt{f} \mapsto [-\infty, 63], \mathtt{sz} \mapsto [64, \infty]]$$
$$\overline{\hat{\Omega}(\varphi_3, \mathtt{t}, \mathtt{sz})} = \top_{\hat{\mathbb{S}}_{\mathbb{I}}}[\mathtt{t} \mapsto [-\infty, 63], \mathtt{sz} \mapsto [64, \infty]].$$

The sliced abstract semantics is

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

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. We show $\hat{\Omega}(\varphi, x, y))$ is an under-approximation of the erroneous states. By Theorem 5 and because $\hat{\Omega}(\varphi, x, y))$ is precisely complementable, we prove the theorem. The details are available in the appendix. □

5.3 Clustering Using Octagon Domain

Now we present another alarm clustering technique using the octagon abstract domain [28] that captures relational properties between variables. Our octagon-based clustering finds 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 $x, y \in Var$ 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 $\alpha_{\mathbb{O}}$:

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

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

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

where $\hat{f}_{\mathbb{O}}$ functions as the standard octagon transfer function for the abstract assignment or the abstract test [28] according to an associated command.

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:

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

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

Example 5.2. Consider the following code, which has been slightly modified from Example 5.1.

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

Suppose we are given the same set of queries as in Example 5.1,

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

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 $[\![\hat{P}]\!]^{\mathbb{O}}$ 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:

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

The sliced abstract semantics is

$$\begin{aligned} [\![\hat{P}]\!]^{\mathbb{O}}_{\neg\varphi_2}(\varphi_2) &= [\![\hat{P}]\!]^{\mathbb{O}}(\varphi_2) \,\hat{\ominus}_{\hat{\mathbb{S}}_{\mathbb{O}}}\, \left(\overline{\alpha_{\mathbb{O}}(\Omega(\varphi_2, \mathtt{f}, \mathtt{sz}))}\right) = \top_{\hat{\mathbb{S}}_{\mathbb{O}}} \sqcap \left(\overline{\alpha_{\mathbb{O}}(\Omega(\varphi_2, \mathtt{f}, \mathtt{sz}))}\right) \\ &= \left(\overline{\alpha_{\mathbb{O}}(\Omega(\varphi_2, \mathtt{f}, \mathtt{sz}))}\right). \end{aligned}$$

By propagating the refinement, we obtain

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

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

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

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

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

5.4 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 [17]. 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 $\hat{Mem} = \hat{Addr} \xrightarrow{\text{fin}} \hat{Val}$ is a finite map from symbolic addresses ($\hat{Addr}$) to symbolic values ($\hat{Val}$):

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

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 (conjunction $\wedge$). Symbols (*Symbol*) are used to indicate symbolic values. A symbolic value can be a number ($\mathbb{Z}$), an address ($\hat{Addr}$), or a binary value ($\hat{Val} \times \mathsf{Bop} \times \hat{Val}$). 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:

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

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., $[\![\hat{P}]\!]^{\mathbb{SE}} = \bigsqcap_{i \in \mathbb{N}} {F_{\mathbb{SE}}}^i(\top_{\Phi \Rightarrow \hat{\mathbb{S}}_{\mathbb{SE}}})$):

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

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

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

and the evaluation $[\![e]\!]$ of an expression e in a memory $\hat{m}$ is defined as usual : $[\![\mathsf{n}]\!](\hat{m}) = \mathsf{n}$, $[\![\mathsf{x}]\!](\hat{m}) = \hat{m}(\mathsf{x})$, and $[\![e_1 + e_2]\!](\hat{m}) = [\![e_1]\!](\hat{m}) + [\![e_2]\!](\hat{m})$. 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 $[\![\hat{P}]\!]^{\mathbb{I}}(\varphi)(\mathsf{x}) = [-\infty, 3]$. Then we insert a node φ' such that $\mathsf{cmd}(\varphi') = \{\{\mathsf{x} \leq 3\}\}$ between φ and all nodes in $\mathsf{predof}(\varphi)$. We do this because our symbolic execution and interval analysis have incomparable precision; for example, the symbolic execution uses a widening operator that changes the unstable abstract states to $\top$ after a finite number of iterations of a loop. In such a case, we aim to improve the precision of the symbolic execution by using the invariant obtained from the interval analysis.

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:

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

Example 5.3. Consider the following code (slightly modified from Example 5.1).

```
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 5.1.

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

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

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

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

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

By propagating the refinement, we obtain

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

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

$$(\mathtt{f} < \mathtt{sz}) \wedge (\mathtt{t} < \mathtt{sz}) \wedge (\mathtt{sq} = (\mathtt{f} + \mathtt{t})/2) \Rightarrow \mathtt{sq} < \mathtt{sz}.$$ □

6 EXPERIMENTS

We apply our clustering methods on 14 packages from three different categories (Bugbench [7], GNU software, and SourceForge open source projects). Table 1 shows the benchmark programs. We implemented our alarm clustering technique on Sparrow [32], an industrial-strength static buffer overrun detector for C programs. The baseline analyzer is flow sensitive, field sensitive, and context insensitive and uses the interval domain. The core semantics of the analyzer is described in Section 5.1

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 baseline 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 [22], the octagon-based clustering reduced 8% of alarms, but our new symbolic execution-based clustering reduces 12% with a smaller cost. We use the ScanCluster algorithm for interval domain-based clustering and the heuristic algorithm for symbolic execution-based clustering because each of symbolic executions requires significant overhead.

Table 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	1.8	0.6
polymorph-0.4.0	21	15	14	29%	5%	0.1	0.01	0.01
ncompress-4.2.4	82	70	52	15%	22%	1.7	2.3	0.9
sbm-0.0.4	269	231	189	14%	16%	4.3	115.4	2.8
stripcc-0.2.0	190	132	110	31%	12%	3.1	3.4	0.5
barcode-0.9.6	416	355	287	15%	16%	3.3	7.0	3.5
129.compress	66	49	35	26%	21%	91.6	951.5	0.2
archimedes-0.7.0	119	24	24	80%	0%	16.6	19.5	2.8
man-1.5h1	287	234	191	18%	15%	31.4	59.7	1.5
gzip-1.2.4	390	325	294	17%	8%	15.6	91.0	5.7
combine-0.3.3	836	485	318	42%	20%	21.8	290.9	117.9
gnuchess-5.05	1040	427	329	59%	9%	67.4	2189.8	154.3
bc-1.06	730	482	337	34%	20%	50.6	1511.7	22.1
grep-2.5.1	948	819	811	14%	1%	35.6	216.9	0.1
TOTAL	5518	3714	3057	33%	12%	343.4	5460.9	313.1

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

In Table 1, the column labeled "**# Alarms**" shows the numbers of alarms reported by the baseline analyzer (**B**), after the clustering using the interval analysis (**I**), and after the clustering using both the interval analysis and the symbolic execution (**S+I**), respectively. The next columns labeled "**% Reduc.**" show the reduction ratios by the interval clustering (**I**) and the further reduction by the symbolic-execution-based clustering (**+S**). As shown in Table 1, our method identifies 45% of the alarms non-dominating. This reduction is in the number to be examined by the user.

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 that 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, and 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 2.8GHz Intel Xeon processor and 24GB of memory. In Table 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**). For each benchmark, we repeat the experiment 10 times and average the running time. The standard deviations do not exceed 7% of the average times.

Table 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

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 3s, 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 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.

7 RELATED WORK

To the best of our knowledge, Le et al.'s work [21] 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 that 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.

There is a large body of work on error cause localization related to our work. A lot of work on locating the sources of type errors in higher-order languages with let-polymorphism [3, 6, 11, 15, 33, 34] identify the source of a type error in the form of program points. Our work is not limited to locate the sources and, moreover, soundly clusters the alarms of the same origins. Error cause localization techniques in model checkers [2, 12] also can be viewed as clustering algorithms. They analyze the common and different features between erroneous and safe traces and provide succinct and useful information about the error traces to the user.

Statistical ranking schemes [16, 19, 20] 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, our technique can be combined with them for a more sophisticated alarm reporting interface as proposed by Mangal et al. [24].

Mangal et al. combine alarm clustering with statistical learning. They propose EUGENE that allows user feedback to guide datalog analysis towards producing the desired output. User feedbacks are about which analysis results an user dislikes (or likes). With the feedbacks, EUGENE derives desired reports after re-running the analysis. To this aim, datalog rules are selectively applied to suppress alarms the user dislikes. In this process, other similar alarms of the same origins are also suppressed, because they are dependent on the same intermediate tuples, which can be seen as alarm clustering. Statistical learning plays a key role in selecting datalog rules to be applied. Datalog rules and initial tuples are equipped with learned weights, and the analysis derives tuples maximizing the sum of total weights. Therefore, this work can be considered a good combination of alarm clustering and statistical learning.

Our work resembles that of Rival's work [31] 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, then 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. The similarity also applies to Gogul's work [1]. Similarly to Rival's work, they refine the abstraction by slicing out non-error states and perform a sequence of many forward and backward runs.

Our clustering method can be integrated with other refinement approaches [1, 4, 9, 13, 14, 18, 31]. Their goal is to remove false alarms by abstraction refinement, whereas our work seeks to reduce the number of alarms to investigate. Our work can also reduce the number of targets to do the refinement.

Our work is more general than error recovery techniques that are used for reducing false alarms in many commercial static analysis tools [5, 25, 27]. For each alarm found, these commercial analyzers recover from those alarms; that is, whenever an alarm is found, they report the alarm, slice the abstract erroneous states, and continue the fixpoint computation. In contrast to the error recovery techniques, we can use a more expressive domain for clustering purposes than the one used in the baseline (as shown in Section 5.4), which can be more precise or cost-effective. Additionally, our method can derive true clusters that cannot be done by the above error recovery techniques.

8 CONCLUSION

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.

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 Section 3.4.)

Proof. We will show the refinement by refutation of alarm φ_1 (i.e., $[\![\tilde{P}]\!]_{\varphi_1}$) still soundly approximates the collecting semantics of P (i.e., $\alpha([\![P]\!]) \sqsubseteq [\![\tilde{P}]\!]_{\varphi_1}$) if alarm φ_1 is false. Then we can conclude alarm φ_2 if the refinement removes alarm φ_2, because the refinement is sound with respect to the collecting semantics. We prove the lemma by induction and the soundness of abstract slice operator.

We begin with proving that $\forall i \in \mathbb{N}.\ \alpha({F_P}^i \bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1}$.

$$
\begin{array}{ll}
\alpha_S([\![P]\!]_{/\delta}(\varphi_1) \ominus \Omega(\varphi_1)) \sqsubseteq [\![\hat{P}]\!](\varphi_1) \hat{\ominus} \alpha_S(\Omega(\varphi_1)) & (\alpha_S \circ \ominus \sqsubseteq \hat{\ominus} \circ \alpha_{S\times S}) \\
\alpha_S([\![P]\!]_{/\delta}(\varphi_1)) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1}(\varphi_1) & (\text{Def. of } [\![\hat{P}]\!]_{\neg\varphi_1} \text{ and } [\![P]\!]_{/\delta}(\varphi_1) \cap \Omega(\varphi_1) = \emptyset) \\
\alpha([\![P]\!]) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} & (\forall \varphi \in \Phi \backslash \{\varphi_1\}.\ [\![\hat{P}]\!](\varphi) = [\![\hat{P}]\!]_{\neg\varphi_1}(\varphi)) \\
\alpha(\bigsqcup_{i\in\mathbb{N}} {F_P}^i \bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} & (\alpha([\![P]\!]) = \alpha(\bigsqcup_{i\in\mathbb{N}} {F_P}^i \bot))
\end{array}
$$

By definition of lub and that α is monotone,

$$\forall i \in \mathbb{N}.\ \alpha({F_P}^i \bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} \tag{2}$$

To show $\alpha([\![P]\!]) \sqsubseteq [\![\tilde{P}]\!]_{\varphi_1} = \text{fix}^{\#}\, \hat{H}$ where $\hat{H} = \lambda \hat{X}.[\![\hat{P}]\!]_{\neg\varphi_1} \sqcap \hat{F}(\hat{X})$, we first show

$$\forall i \in \mathbb{N}.\ \alpha({F_P}^i \bot) \sqsubseteq \hat{H}^i(\hat{\bot}) \tag{3}$$

by induction.

- Basis:

$$
\begin{array}{ll}
\alpha(F_P(\bot)) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} & (\text{By } 2) \\
\alpha(F_P(\bot)) \sqsubseteq \hat{F}(\bot) & (\alpha \circ F_P \sqsubseteq \hat{F} \circ \alpha) \\
\therefore \alpha(F_P(\bot)) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} \sqcap \hat{F}(\hat{\bot}) = \hat{H}(\hat{\bot}) &
\end{array}
$$

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

$$
\begin{array}{ll}
\alpha({F_P}^{k+1} \bot) = \alpha(F_P \circ {F_P}^k \bot) & \\
\sqsubseteq \alpha(F_P \circ \gamma \circ \alpha \circ {F_P}^k \bot) & (\alpha \circ F_P \text{ is monotone,} \\
 & \quad \text{and } id \sqsubseteq \gamma \circ \alpha) \\
\sqsubseteq \alpha \circ F_P \circ \gamma(\hat{H}^k(\hat{\bot})) & (\text{By IH}) \\
\sqsubseteq \hat{F}(\hat{H}^k(\hat{\bot})) & (\alpha \circ F_P \sqsubseteq \hat{F} \circ \alpha) \\
 & \\
\alpha({F_P}^{k+1} \bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} & (\text{By } 2) \\
\therefore \alpha({F_P}^{k+1} \bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi_1} \sqcap \hat{F}(\hat{H}^k(\hat{\bot})) = \hat{H}^{k+1}(\hat{\bot}) &
\end{array}
$$

Now, we can show $\alpha([\![P]\!]) \sqsubseteq \mathsf{fix}^{\#}\,\hat{H} = [\![\tilde{P}]\!]_{\varphi_1}$ as follows:

$$\begin{array}{ll} \bigsqcup_{i\in\mathbb{N}} \alpha({F_P}^i\bot) \sqsubseteq \bigsqcup_{i\in\mathbb{N}} \hat{H}^i(\hat{\bot}) & \text{(By 3)} \\ \alpha(\bigsqcup_{i\in\mathbb{N}} {F_P}^i\bot) \sqsubseteq \bigsqcup_{i\in\mathbb{N}} \hat{H}^i(\hat{\bot}) & (\alpha \text{ is continuous.}) \\ \alpha([\![P]\!]) \sqsubseteq \mathsf{fix}^{\#}\,\hat{H} = [\![\tilde{P}]\!]_{\varphi_1}. & \end{array}$$

Finally, we can conclude alarm φ_2 is false as follows, because the refinement is sound.

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

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 Section 3.4.)

Proof. The proof is similar to the proof of Lemma 1 except that we refute multiple alarms. We begin with proving that $\forall i \in \mathbb{N}.\, \alpha({F_P}^i\bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}}$.

$$\begin{array}{ll} \forall\varphi \in \overrightarrow{\varphi}.\, \alpha_S([\![P]\!]_{/\delta}(\varphi) \ominus \Omega(\varphi)) \sqsubseteq [\![\hat{P}]\!](\varphi) \hat{\ominus} \alpha_S(\Omega(\varphi)) & (\alpha_S \circ \ominus \sqsubseteq \hat{\ominus} \circ \alpha_{S\times S}) \\ \forall\varphi \in \overrightarrow{\varphi}.\, \alpha_S([\![P]\!]_{/\delta}(\varphi)) \sqsubseteq [\![\hat{P}]\!]_{\neg\varphi}(\varphi) & (\forall\varphi \in \overrightarrow{\varphi}.\, [\![P]\!]_{/\delta}(\varphi) \cap \Omega(\varphi) = \emptyset) \\ \forall\varphi \in \overrightarrow{\varphi}.\, \alpha_S([\![P]\!]_{/\delta}(\varphi)) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}}(\varphi) & ([\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} = \sqcap_{\varphi\in\overrightarrow{\varphi}} [\![\hat{P}]\!]_{\neg\varphi}) \\ \alpha([\![P]\!]) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} & \\ \alpha(\bigsqcup_{i\in\mathbb{N}} {F_P}^i\bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} & (\alpha([\![P]\!]) = \alpha(\bigsqcup_{i\in\mathbb{N}} {F_P}^i\bot)) \end{array}$$

By definition of lub and that α is monotone,

$$\forall i \in \mathbb{N}.\, \alpha({F_P}^i\bot) \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} \tag{4}$$

The remaining part is similar to the corresponding part in the proof of Lemma 1; simply substituting φ_1 for $\overrightarrow{\varphi}$ and φ_2 for φ_0 completes the proof. □

Lemma 3. $\overrightarrow{\varphi} \subseteq \overrightarrow{\varphi}' \Rightarrow C_{\overrightarrow{\varphi}} \subseteq C_{\overrightarrow{\varphi}'}$
(Stated in Section 4.1.)

Proof.

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

Therefore, $C_{\overrightarrow{\varphi}} = \{\varphi \in \mathcal{A} \mid \overrightarrow{\varphi} \rightsquigarrow \varphi\} \subseteq C_{\overrightarrow{\varphi}'} = \{\varphi \in \mathcal{A} \mid \overrightarrow{\varphi}' \rightsquigarrow \varphi\}$ □

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

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

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 $H' \sqsubseteq H$, because $[\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}} \sqsubseteq [\![\hat{P}]\!]_{\neg\overrightarrow{\varphi}'}$. We conclude $\mathsf{fix}^{\#}\,H' \sqsubseteq \mathsf{fix}^{\#}\,H$, because $\bigsqcup_{i\in\mathbb{N}} H'^i\bot \sqsubseteq \bigsqcup_{i\in\mathbb{N}} H^i\bot$. Therfore, $[\![\tilde{P}]\!]_{\overrightarrow{\varphi}'} \sqsubseteq [\![\tilde{P}]\!]_{\overrightarrow{\varphi}}$. □

Theorem 4. Algorithm 3 computes sound alarm dependencies.
(Stated in Section 4.2.)

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

Now we show $\forall\varphi \in \Phi.\, T(\varphi) = [\![\tilde{P}]\!]_{R(\varphi)}(\varphi)$. At line 33 after the function FixpointIterate is called, $T = [\![\tilde{P}]\!]_{\Phi}$, because we refute all alarms and compute the refinement. In addition, by Lemma 6, $\forall\varphi \in \Phi.\, [\![\tilde{P}]\!]_{\Phi} = [\![\tilde{P}]\!]_{R(\varphi)}$. Therefore $\forall\varphi \in \Phi.\, T(\varphi) = [\![\tilde{P}]\!]_{R(\varphi)}(\varphi)$. □

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

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

$$\forall\varphi \in \Phi.\, [\![\tilde{P}]\!]_{R(\varphi)}(\varphi) \sqsubseteq T(\varphi). \tag{5}$$

As the base case, the loop invariant holds as follows at the first entrance to the loop:

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

As the inductive step, assuming the loop invariant (5) currently holds, we show the loop 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 (lines 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, our goal is to show that $[\![\tilde{P}]\!]_{\overrightarrow{\varphi}_{new}}(\varphi) \sqsubseteq \hat{s}_{new}$ in every case.

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

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

  By the inductive hypothesis 5, $\forall\varphi_i \in \mathsf{pred}(\varphi).\, [\![\tilde{P}]\!]_{R(\varphi)}(\varphi) \sqsubseteq T(\varphi)$ and that $\hat{f}(\varphi)$ is monotone,

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

  Therefore, $[\![\tilde{P}]\!]_{\overrightarrow{\varphi}_{new}}(\varphi) \sqsubseteq \hat{s}_{new}$.
- Case $\hat{s} \sqsubseteq \hat{s}'$: immediate from the inductive hypothesis (5).
- Case $\hat{s} \not\sqsupset \hat{s}', \hat{s} \not\sqsubseteq \hat{s}'$:
  Let $R' = \bigcup_{\varphi_i \in \mathsf{pred}(\varphi)} R(\varphi_i)$. From the above two previous cases, we have concluded that $[\![\tilde{P}]\!]_{R'}(\varphi) \sqsubseteq \hat{s}'$ and $[\![\tilde{P}]\!]_{R(\varphi)}(\varphi) \sqsubseteq \hat{s}$. By Lemma 7, $[\![\tilde{P}]\!]_{R(\varphi)\cup R'}(\varphi) \sqsubseteq \hat{s} \sqcap \hat{s}'$. Because $R(\varphi) \cup R' = \overrightarrow{\varphi}_{new}$ and $\hat{s} \sqcap \hat{s}' = \hat{s}_{new}$, 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 ($\forall\varphi \in \Phi.\, R(\varphi) \subseteq \Phi$). And by the loop invariant 5, we conclude $\forall\varphi \in \Phi.\, [\![\tilde{P}]\!]_{\Phi}(\varphi) = [\![\tilde{P}]\!]_{R(\varphi)}(\varphi)$. □

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

Proof.

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

□

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 Section 5.2.)

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 $\hat{\Omega}(\varphi, x, y) = \{x \mapsto [y_{max}, +\infty], y \mapsto [-\infty, y_{min} - 1]\}$:
  $\forall s \in \gamma_{\mathbb{I}}(\hat{\Omega}(\varphi, x, y)).\, s(x) \geq s(y)$, because $y_{max} \geq y_{min} - 1$.

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,

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

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

In addition, because $\gamma_{\mathbb{I}}(\hat{\Omega}(\varphi, x, y)) \subseteq \Omega(\varphi, x, y)$,

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

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

REFERENCES

[1] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, Ou Wei, and Aarti Gupta. 2008. SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *Proceedings of the 15th International Symposium on Static Analysis (SAS'08)*. Springer-Verlag, Berlin, 238–254. DOI: http://dx.doi.org/10.1007/978-3-540-69166-2_16

[2] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. ACM, New York, NY, 97–105. DOI: http://dx.doi.org/10.1145/604131.604140

[3] Mike Beaven and Ryan Stansifer. 1993. Explaining type errors in polymorphic languages. *ACM Lett. Program. Lang. Syst.* 2, 1–4 (March 1993), 17–30. DOI: http://dx.doi.org/10.1145/176454.176460

[4] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise refutations for heap reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 275–286. DOI: http://dx.doi.org/10.1145/2491956.2462186

[5] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. ACM, New York, NY, 196–207. DOI: http://dx.doi.org/10.1145/781131.781153

[6] Olaf Chitil. 2001. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*. ACM, New York, NY, 193–204. DOI: http://dx.doi.org/10.1145/507635.507659

[7] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. 2009. 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)*. ACM, New York, NY, 16–20. DOI: http://dx.doi.org/10.1145/1555860.1555866

[8] Patrick Cousot and Rahida Cousot. 1992. Abstract interpretation and application to logic programs. *J. Log. Program.* 13, 2–3 (July 1992), 103–179. DOI: http://dx.doi.org/10.1016/0743-1066(92)90030-7

[9] P. Cousot, P. Ganty, and J.-F. Raskin. 2007. Fixpoint-guided abstraction refinements. In *Proceedings of the 14th International Symposium on Static Analysis, SAS '07*, G. Filé and H. Riis Nielson (Eds.). Springer, Berlin, 333–348.

[10] Vijay D'Silva, Leopold Haller, Daniel Kroening, and Michael Tautschnig. 2012. 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)*. Springer-Verlag, Berlin, 48–63. DOI: http://dx.doi.org/10.1007/978-3-642-28756-5_5

[11] Dominic Duggan and Frederick Bent. 1995. Explaining type inference. In *Science of Computer Programming*. 37–83.

[12] Alex Groce and Willem Visser. 2003. What went wrong: Explaining counterexamples. In *Proceedings of the 10th International Conference on Model Checking Software (SPIN'03)*. Springer-Verlag, Berlin, 121–136. http://dl.acm.org/citation.cfm?id=1767111.1767119

[13] Bhargav S. Gulavani and Sriram K. Rajamani. 2006. Counterexample driven refinement for abstract interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Holger Hermannsand Jens Palsberg (Eds.). Lecture Notes in Computer Science, Vol. 3920. Springer, Berlin, 474–488. DOI: http://dx.doi.org/10.1007/11691372_34

[14] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. 2008. 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)*. Springer-Verlag, Berlin, 443–458.

[15] Gregory F. Johnson and Janet A. Walz. 1986. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'86)*. ACM, New York, NY, 44–57. DOI: http://dx.doi.org/10.1145/512644.512649

[16] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. 2005. 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)*. Springer-Verlag, Berlin, 203–217. DOI: http://dx.doi.org/10.1007/11547662_15

[17] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. 2011. MeCC: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 301–310. DOI: http://dx.doi.org/10.1145/1985793.1985835

[18] Youil Kim, Jooyong Lee, Hwansoo Han, and Kwang-Moo Choe. 2010. Filtering false alarms of buffer overflow analysis using SMT solvers. *Inf. Softw. Technol.* 52, 2 (Feb. 2010), 210–219. DOI: http://dx.doi.org/10.1016/j.infsof.2009.10.004

[19] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. 2004. Correlation exploitation in error ranking. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT'04/FSE-12)*. ACM, New York, NY, 83–93. DOI: http://dx.doi.org/10.1145/1029894.1029909

[20] Ted Kremenek and Dawson Engler. 2003. 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)*. Springer-Verlag, Berlin, 295–315.

[21] Wei Le and Mary Lou Soffa. 2010. Path-based fault correlations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. ACM, New York, NY, 307–316. DOI: http://dx.doi.org/10.1145/1882291.1882336

[22] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. 2012. Sound non-statistical clustering of static analysis alarms. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12)*. Springer-Verlag, Berlin, 299–314. DOI: http://dx.doi.org/10.1007/978-3-642-27940-9_20

[23] Percy Liang, Omer Tripp, and Mayur Naik. 2011. Learning minimal abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, NY, 31–42. DOI: http://dx.doi.org/10.1145/1926385.1926391

[24] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, 462–473. DOI: http://dx.doi.org/10.1145/2786805.2786851

[25] MathWorks. 2015. Polyspace Embedded Software Verification. Retrieved from http://www.mathworks.com/products/polyspace/index.html.

[26] Laurent Mauborgne and Xavier Rival. 2005. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the 14th European Conference on Programming Languages and Systems (ESOP'05)*. Springer-Verlag, Berlin, 5–20. DOI: http://dx.doi.org/10.1007/978-3-540-31987-0_2

[27] Microsoft. 2015. Code Contracts. Retrieved from http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx.

[28] Antoine Miné. 2006. The octagon abstract domain. *Higher Order Symbol. Comput.* 19, 1 (March 2006), 31–100. DOI: http://dx.doi.org/10.1007/s10990-006-8609-1

[29] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. 2014. Global sparse analysis framework. *ACM Trans. Program. Lang. Syst.* 36, 3, Article 8 (Sept. 2014), 44 pages. DOI: http://dx.doi.org/10.1145/2590811

[30] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. 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)*. ACM, New York, NY, 229–238. DOI: http://dx.doi.org/10.1145/2254064.2254092

[31] Xavier Rival. 2005. Understanding the origin of alarms in ASTRÉE. In *Proceedings of the 12th International Conference on Static Analysis (SAS'05)*. Springer-Verlag, Berlin, 303–319. DOI: http://dx.doi.org/10.1007/11547662_21

[32] ROPAS. 2017. The Sparrow Static Analyzer. Retrieved from https://github.com/ropas/sparrow.

[33] F. Tip and T. B. Dinesh. 2001. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.* 10, 1 (Jan. 2001), 5–55. DOI: http://dx.doi.org/10.1145/366378.366379

[34] Mitchell Wand. 1986. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'86)*. ACM, New York, NY, 38–43. DOI: http://dx.doi.org/10.1145/512644.512648

Received June 2015; revised December 2016; accepted May 2017