Sound Non-Statistical Clustering of Static Analysis Alarms *

Woosuk Lee, Wonchan Lee, and Kwangkeun Yi

Seoul National University

Abstract. 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 alarm of a cluster turns out to be false (respectively true) then it is assured that all others in the same cluster are also false (respectively true). We have implemented our clustering algorithm on top of a realistic buffer-overflow analyzer and proved that our method has the effect of reducing 54% of alarm reports. Our framework is applicable to any abstract interpretation-based static analysis and orthogonal to abstraction refinements and statistical ranking schemes.

1 Introduction

1.1 Problem

Users of sound static analyzers frequently suffer from a large number of false alarms. When we run a static analyzer for realistic software, false alarms often outnumber real errors. For example, in a case of analyzing commercial software, we have found only one error in 273 buffer-overflow alarms after a tedious alarm investigation work [10].

Although statistical ranking schemes [10][13] help to find real errors quickly, ranking schemes do not reduce alarm-investigation burdens. Statistical 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, the number of alarms to investigate is not reduced with ranking. We should examine all the alarms in order to find all the possible errors.

1.2 Our Solution

One way to reduce alarm-investigation burden is to cluster alarms according to their sound dependence information. We say that alarm A has (sound) dependence on alarm B if alarm B turns out to be false (true resp.), then so does

* This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2011-0000971), the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2011, and a research grant from Samsung Electronics DMC R&D Center.

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 (true resp.).

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

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 (Beginning Example). Our analyzer reports 5 buffer-overflow alarms for the following code excerpted from NLKAIN 1.3 (Alarms are 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 buffer access `u[-nx-2]` at line 8 overflows the buffer, so do the others since `-nx-2` is the lowest index among the indices of all the buffer accesses on `u`.
2. If buffer access `u[nx+2]` at line 9 does not overflow the buffer, neither do the others since `nx+2` is the highest index among the indices of all the buffer accesses on `u`.

Using these two facts, we can cluster alarms in two different ways: we can find a false alarm cluster which consists of all the alarms in the example and the dominant alarm is the one of the buffer access `u[nx+2]` at line 9. We can also find the true alarm cluster in the same way, except that the dominant alarm is the one of the buffer access `u[-nx-2]` at line 8. Instead of inspecting all the alarms, checking either the alarm of buffer access `u[-nx-2]` true or the alarm of buffer access `u[nx+2]` false is sufficient for users. □

Example 2 (Inter-procedural alarm dependencies). The following code excerpted from Appcontour 1.1.0 shows inter-procedural alarm dependencies. Our analyzer reports three alarms at line 3, 4, and 10. In the example, array `invmergerules` and `invmergerulesnn` have the same size 8.

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

Note the following two facts in this example:

1. If the alarm of the buffer access `invmergerules[i]` at line 3 is false, so are the others.
   - If alarm at line 3 is false, so is the one at line 4 because the buffer accesses at line 3 and 4 use the same index variable `i` and there is no update on the value between the two.
   - If alarm at line 3 is false, so is the one at line 10 because the value of index variable `i` at line 3 is passed to the index variable `rule` at line 10 without any change by function return and call ($5 \rightarrow 15 \rightarrow 16 \rightarrow 10$).
2. If the buffer access `invmergerules[rule]` at line 10 overflows, so do the others in a similar reason as the first fact.

We can find a false and true alarm cluster in the similar manner as in example 1. Instead of inspecting all the alarms, checking either the alarm at line 10 true or the alarm at line 3 false is sufficient. □

Example 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. Array `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, buffer access at line 9 is also safe. In this example, we have a false cluster whose dominant alarms are the ones at line 3 and 4. □

Contributions.

- We introduce a sound alarm clustering method for static analyzers that can reduce the alarm-investigation cost. Our framework is general in that it is applicable to any semantics-based static analysis. It is orthogonal to both refining approaches and statistical ranking schemes.
- We prove the effectiveness of our clustering method for the benchmark of 16 open-source programs. By our clustering method, we reduce the number of alarms to investigate by 54%.

Organization. Section 2 introduces our alarm clustering framework. Section 3 explains one practical algorithm which is a sound implementation of our alarm clustering method. Section 4 discusses the experiment results. We implemented our clustering algorithm on top of realistic buffer-overflow analyzer and apply it to the benchmark of 16 open-source programs. Section 5 discusses the related work and Section 6 concludes.

2 Alarm Clustering Framework

We describe our general framework of alarm clustering. In the rest of this section, we suppose basic knowledge of the abstract interpretation framework [3] and the trace partitioning abstract domain [16]. We begin by giving some definitions excerpted from [16].

2.1 Definitions

Programs. We define a program P as a transition system $(S, \rightarrow, S_\iota)$ where S is the set of states of the program, $\rightarrow$ is the transition of the possible execution elementary steps and S_ι denotes the set of initial states.

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

A trace of program P is defined as a set $[\![P]\!] \triangleq \{\sigma \in S^* \mid \sigma_0 \in S_\iota \wedge \forall i.\sigma_i \rightarrow \sigma_{i+1}\}$. The set $[\![P]\!]$ is prefix-closed least fixpoint of the semantic function; i.e. $[\![P]\!] = \mathsf{lfp} F_P$ where F_P is the semantic function, defined as:

$$\begin{aligned} F_P &: 2^{S^*} \rightarrow 2^{S^*} \\ F_P(E) &= \{\langle s_\iota \rangle \mid s_\iota \in \mathcal{S}_\iota\} \\ &\quad \cup \{\langle s_0, \cdots, s_{n+1} \rangle \mid \langle s_0, \cdots, s_n \rangle \in E \wedge s_n \rightarrow s_{n+1}\}. \end{aligned}$$

Partitioned Reachable States. Using a well-chosen trace partitioning function $\delta : \Phi \rightarrow 2^{S^*}$, where Φ is the set of partitioning indices, one can model indexed collections of program states. Domain $\Phi \rightarrow 2^S$ is a partitioned reachable-state domain. The involved abstraction is $\alpha_0(\Sigma)(\varphi) \triangleq \{\sigma_\dashv \mid \sigma \in \Sigma \cap \delta(\varphi)\}$ and the concretization is $\gamma_0(f) \triangleq \{\sigma \mid \forall \tau \preceq \sigma. \forall \varphi.\ \tau \in \delta(\varphi) \Rightarrow \tau_\dashv \in f(\varphi)\}$. The pair of functions (α_0, γ_0) forms a Galois connection: $2^{S^*} \underset{\alpha_0}{\overset{\gamma_0}{\leftrightarrows}} \Phi \rightarrow 2^S$. We write concrete semantics $[\![P]\!]$ modulo partitioning function δ as $[\![P]\!]_{/\delta}$.

Abstract Semantics. We think of a static analyzer which is designed over an abstract domain $\hat{D} = \Phi \rightarrow \hat{S}$ with the following Galois connections:

$$2^{S^*} \underset{\alpha_0}{\overset{\gamma_0}{\leftrightarrows}} \Phi \rightarrow 2^S \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \Phi \rightarrow \hat{S}.$$

The galois connection of (α, γ) is easily derived from the one of (α_S, γ_S) between domains 2^S and $\hat{S}$: $2^S \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \hat{S}$.

The abstract semantics of program P computed by the analyzer is a fixpoint $\hat{T} = \mathsf{lfp}^\# \hat{F}$ where $\mathsf{lfp}^\#$ is a sound, abstract post-fixpoint operator and the function $\hat{F} : \hat{D} \rightarrow \hat{D}$ is a monotone or an extensive abstract transfer function such that $\alpha \circ \alpha_0 \circ F_P \sqsubseteq \hat{F} \circ \alpha \circ \alpha_0$. The soundness of the static analysis follows from the fixpoint transfer theorem [2].

Alarms. The static analyzer raises an alarm at trace partitioning index φ if $\gamma_S(\hat{T}(\varphi)) \cap \Omega(\varphi) \neq \varnothing$ where $\hat{T}$ is the abstract semantics of a program P and function $\Omega : \Phi \rightarrow 2^S$ specifies erroneous states at each partitioning index. In the rest of the paper, we use partitioning index and alarm interchangeably; alarm φ means the one at the trace partitioning index φ.

The alarm φ is false alarm (resp. true alarm) when the static analyzer raises the alarm and $[\![P]\!]_{/\delta}(\varphi) \cap \Omega(\varphi) = \varnothing$ (resp. $[\![P]\!]_{/\delta}(\varphi) \cap \Omega(\varphi) \neq \varnothing$).

Alarm Dependence Our goal is to find concrete dependencies between alarms. Given two alarms φ_1 and φ_2, if alarm φ_2 is always false whenever alarm φ_1 is false; i.e.

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

we say that alarm φ_2 has a concrete dependence on alarm φ_1. If we find this concrete dependence of alarm φ_2 on alarm φ_1, we also have another dependence as contraposition.

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

Since concrete dependence is not computable in general, we use abstract dependence which is sound with respect to concrete dependence. The idea is that if we can kill the alarm φ_2 from the abstract semantics refined under the assumption that alarm φ_1 is false, it also means that alarm φ_2 has concrete dependence on alarm φ_1. It is easy to see that this is correct because, even though the refined abstract semantics is smaller than the original fixpoint, it is still sound abstraction of concrete semantics if the assumption of alarm φ_1 false holds.

In the rest of the section, we define the notion of sound refinement by refutation and abstract dependence. We also prove the soundness of abstract dependence.

Refinement by Refutation. Using the assumption of alarm φ being false, we can get a sliced abstract semantics $\tilde{T}_{\varphi}$. The definition of $\tilde{T}_{\varphi}$ is,

$$\tilde{T}_{\varphi} = \mathsf{gfp}^{\#} \lambda Z. \hat{T}_{\neg\varphi} \sqcap \hat{F}(Z)$$

where $\mathsf{gfp}^{\#}$ is a pre-fixpoint operator and $\hat{T}_{\neg\varphi}$ is the same as the original fixpoint $\hat{T}$ except the erroneous states at partitioning index φ sliced out:

$$\hat{T}_{\neg\varphi} = \hat{T}[\varphi \mapsto \hat{T}(\varphi) \mathbin{\hat{\ominus}} \alpha_S(\Omega(\varphi))]$$

where $F[a \mapsto b]$ is the same as F except it maps a to b. The $\hat{\ominus}$ operator should be a sound abstract slice operator such that $\alpha_S \circ \ominus \sqsubseteq \hat{\ominus} \circ \alpha_{S\times S}$ where the operator $\ominus$ is a set difference and $\alpha_{S\times S}$ is an abstraction lifted for pairs. We assume that the abstract domain $\hat{S}$ has meet operator and abstract slice operator.

We can extend this refinement to the case of refuting multiple alarms. Suppose that we assume that set $\{\varphi_1, \cdots, \varphi_n\}$ of alarms is false. The refinement $\tilde{T}_{\{\varphi_1,\cdots,\varphi_n\}}$ of the fixpoint $\hat{T}$ with respect to these assumptions is,

$$\tilde{T}_{\vec{\varphi}} = \mathsf{gfp}^{\#} \lambda Z. \hat{T}_{\neg\{\varphi_1,\cdots,\varphi_n\}} \sqcap \hat{F}(Z)$$

where $\hat{T}_{\neg\{\varphi_1,\cdots,\varphi_n\}} = \bigsqcap_{\varphi_i \in \{\varphi_1,\cdots,\varphi_n\}} \hat{T}_{\neg\varphi_i}$.

Abstract Alarm Dependence. We now define abstract alarm dependence.

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

$$\textit{iff } \gamma_S(\tilde{T}_{\varphi_1}(\varphi_2)) \cap \Omega(\varphi_2) = \varnothing.$$

We write $\varphi_1 \rightsquigarrow \varphi_2$ when an alarm φ_2 has abstract dependence on alarm φ_1. We prove the soundness of abstract alarm dependence as the following lemma.

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

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

Lemma 2 *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 notion of the abstract dependence for more than two alarms.

Definition 2 ($\{\varphi_1, \cdots, \varphi_n\} \rightsquigarrow \varphi_0$) *Given set $\{\varphi_0, \cdots, \varphi_n\}$ of alarms, we write $\{\varphi_1, \cdots, \varphi_n\} \rightsquigarrow \varphi_0$, and say that alarm φ_0 has abstract dependence on set $\{\varphi_1, \cdots, \varphi_n\}$ of alarms, iff the refinement $\tilde{T}_{\{\varphi_1, \cdots, \varphi_n\}}$ by refuting set $\{\varphi_1, \cdots, \varphi_n\}$ of alarms satisfies*

$$\gamma_S(\tilde{T}_{\{\varphi_1, \cdots, \varphi_n\}}(\varphi_0)) \cap \Omega(\varphi_0) = \varnothing.$$

Lemma 3 *Given set $\{\varphi_0, \cdots, \varphi_n\}$ of alarms, if $\{\varphi_1, \cdots, \varphi_n\} \rightsquigarrow \varphi_0$, then alarm φ_0 is false whenever all alarms $\varphi_1, \cdots, \varphi_n$ are false.*

The contraposition of lemma 3 is not quite useful since it specifies only some alarms among set $\{\varphi_1, \cdots, \varphi_n\}$ of alarms are true when $\{\varphi_1, \cdots, \varphi_n\} \rightsquigarrow \varphi_0$ and alarm φ_0 is true.

In the rest of paper, we sometimes write $\overrightarrow{\varphi}$ to denote a set of alarms.

2.2 Alarm Clustering

Using abstract alarm dependencies, we can cluster alarms in two different ways.

Definition 3 (False Alarm Cluster) *Let $\mathcal{A}$ be set of all alarms in program P and $\rightsquigarrow$ be the dependence relation. A false alarm cluster $\mathcal{C}^F_{\overrightarrow{\varphi}} \subseteq \mathcal{A}$ with its dominant alarms $\overrightarrow{\varphi}$ is $\{\varphi \in \mathcal{A} \mid \overrightarrow{\varphi} \rightsquigarrow \varphi\}$.*

Definition 4 (True Alarm Cluster) *Let $\mathcal{A}$ be set of all alarms in program P and $\rightsquigarrow$ be the dependence relation. A true alarm cluster $\mathcal{C}^T_{\varphi} \subseteq \mathcal{A}$ with its dominant alarms φ is $\{\varphi' \in \mathcal{A} \mid \varphi' \overset{+}{\rightsquigarrow} \varphi\}$ ($\overset{+}{\rightsquigarrow}$ is the transitive closure of $\rightsquigarrow$ between only singleton alarms).*

Note that we cannot exploit dependencies like $\{\varphi_1, \cdots, \varphi_n\} \rightsquigarrow \varphi_0$ to make true alarm cluster. As we mentioned in 2.1, it does not tell us exactly which alarms among set $\{\varphi_1, \cdots, \varphi_n\}$ of alarms are true when alarm φ_0 is true.

The soundness of true and false alarm clusters directly follow the soundness of abstract alarm dependence.

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

Theorem 2 *Every alarm in $\mathcal{C}_\varphi^T$ is true whenever alarm φ is true.*

For two reasons, we only focus on false alarm clusters. First, both type of clusters can be found from the same dependence relation $\rightsquigarrow$, so whether to make true or false alarm is simply the matter of interpretation. Second, in our current framework, true alarm clusters can exploit fewer dependencies than false alarm cluster, thus they cluster less alarms. In the rest of the paper, a cluster $\mathcal{C}$ means a false alarm cluster $\mathcal{C}^F$.

3 Alarm Clustering Algorithm

As we explain in section 2.2, we need to compute abstract dependence relation among all the alarms for clustering. A naive way to do this is to enumerate all possible subsets of all the alarms and find the others that are dominated by them. This naive algorithm requires 2^N times of re-computation where N is number of alarms, which is far from practical.

We present one practical alarm clustering algorithm, shown in algorithm 1, which clusters alarms based on a (not all) subset of possible dependencies. By one fixpoint computation, our algorithm finds the subset of possible dependencies. The idea is to slice the static analysis result as much as possible by refuting all alarms and track which dominant alarm candidate possibly kills which alarm. Then, we cluster the alarms which must be killed by the same dominant alarm candidate.

Our algorithm works in the following way: we start by assuming that each alarm is a dominant one of a cluster that clusters only itself. This can be expressed by slicing out the erroneous states at every alarm point but not propagating refinement yet. Then from an alarm point, say φ_1, we start building its cluster. We propagate its sliced, non-erroneous abstract state to another alarm point say φ_2 and see if the propagation further refines the non-erroneous abstract state at φ_2. If the propagated state is smaller than that at φ_2, it means refuting φ_1 will refute alarm φ_2, hence dependence $\varphi_1 \rightsquigarrow \varphi_2$ and thus we add φ_2 to the φ_1-dominating cluster. If the propagated state is larger than that at φ_2, then dependence $\varphi_1 \rightsquigarrow \varphi_2$ is not certain hence, instead of adding φ_2 to the φ_1-dominating cluster, we start building the φ_2-dominating cluster. If the propagated state is incomparable to that at φ_2, then we pick both alarms as dominant ones and start building the φ_1-and-φ_2-dominating cluster by propagating the slicing effect of simultaneously refuting (i.e., taking the meet of refuting) both alarms.

In the algorithm, we assume that Φ is the set of program points and every program point has several predecessors and successors specified by function pred and succ (line 2). For brevity, we also assume that an alarm can be raised at every program point; i.e. for all $\varphi \in \Phi$, $\hat{\Omega}(\varphi) \neq \bot$ where $\hat{\Omega}$ is abstract erroneous information (line 8).

From line 1 to 9, we give definitions used in the algorithm. Everything other than function R at line 7 is trivially explained by the comment on the same line.

Algorithm 1 Clustering algorithm

```
1: w ∈ Work = Φ    W ∈ Worklist = 2^Work
2: pred ∈ Predecessors = Φ → 2^Φ    succ ∈ Successors = Φ → 2^Φ
3: f̂ ∈ Φ → Ŝ → Ŝ                          (* abstract transfer function for each program point *)
4: T ∈ Table = Φ → Ŝ                      (* abstract state indexed by program point *)
5: φ⃗ ∈ DomCand = 2^Φ                      (* dominant alarm candidate. set of alarms. *)
6: Δ ∈ 2^DomCand                          (* set of dominant alarm candidates *)
7: R ∈ RefinedBy = Φ → 2^DomCand          (* {φ ↦ Δ} ∈ R : T(φ) is refined by φ⃗ in Δ *)
8: Ω̂ ∈ ErrorInfo = Φ → Ŝ                 (* abstract erroneous state information *)
9: C ∈ Clusters = DomCand → 2^Partid       (* alarm clusters indexed by dominant alarms *)
10: procedure FixpointIterate(W, T, R)
11:     repeat
12:         φ := choose(W)                (* pick a work from worklist *)
13:         ŝ := T(φ)                     (* previous abstract state *)
14:         ŝ' := f̂(φ)(⨆_{φ_i ∈ pred(φ)} T(φ_i))   (* new abstract state *)
15:         ŝ_new := ŝ' ⊓ ŝ
16:
17:         Δ := R(φ)                     (* previous set of dominant alarm candidates *)
18:         Δ' := ⋃_{φ_i ∈ pred(φ)} R(φ_i)   (* new set of dominant alarm candidates *)
19:         if ŝ ⊐ ŝ' then Δ_new = Δ'
20:         else if ŝ ⊑ ŝ' then Δ_new = Δ
21:         else Δ_new := Δ ⩔ Δ'
22:         if ŝ_new ⊏ ŝ then             (* propagate the change to successors *)
23:             W := W ∪ succ(φ); T(φ) := ŝ_new; R(φ) := Δ_new
24:     until W = ∅
25: procedure ClusterAlarms(T, R)
26:     for all φ ∈ Φ do
27:         if T(φ) ⊓ Ω̂(φ) = ⊥ then
28:             for all φ⃗ ∈ R(φ) do
29:                 C := C{φ⃗ ↦ C(φ⃗) ∪ {φ}}
30: procedure main()
31:     T := T̂_{¬Φ}                       (* T̂ is the original fixpoint *)
32:     R := {φ ↦ {{φ}} | φ ∈ Φ}
33:     FixpointIterate(Φ,T,R); ClusterAlarms(T,R)
```

Function R keeps the information of dominant alarm candidate. As specified in the comment, if $R(\varphi) = \Delta$ for some program point φ and set Δ of dominant alarms, it means that the abstract state at φ is refined by some dominant alarm candidate $\overrightarrow{\varphi}$ in Δ, thus alarm φ can be a member of the $\overrightarrow{\varphi}$-dominating cluster. We keep the set of dominant alarm candidates, not a single dominant alarm candidate, since there are branches where each branch takes different dominant alarm candidate. Line 32 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. To guarantee the termination or to speed up, we can integrate acceleration method (such as widening [4] in the decreasing direction). We start

the fixpoint computation from the fixpoint refined by refuting all alarms (line 32).

Gray-boxed parts are to track which set of dominant alarm candidates refines the abstract state at program point φ. As specified from line 19 to line 21, there are three cases: 1) the new abstract state refines the previous one (line 19), 2) the previous abstract state is smaller than or equal to the new one (line 20), and 3) both abstract states are incomparable (line 21). For the first case, we change the set of dominant alarm candidates to the new one Δ' (line 18). For the second case, we do not change (line 19) since we cannot further refine the abstract state. For the last case, we pick both dominant alarm candidates from set Δ and Δ' (line 20). The new set of dominant alarm candidates is thus computed by the following lifted union $\Cup$:

$$\Delta_1 \Cup \Delta_2 = \{\overrightarrow{\varphi}_1 \cup \overrightarrow{\varphi}_2 \mid \overrightarrow{\varphi}_1 \in \Delta_1 \wedge \overrightarrow{\varphi}_2 \in \Delta_2\}.$$

For each dominant alarm candidate $\overrightarrow{\varphi}_1$ and $\overrightarrow{\varphi}_2$ in set Δ_1 and Δ_2 of alarm candidates, respectively, we union the two.

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

4 Experiments

4.1 Implementation

We have implemented our alarm clustering method on top of Airac [9, 10, 19–21], a realistic buffer-overflow analyzer for C programs. Our static analyzer is a sound, inter-procedural abstract interpreter with interval domain. Because of limited space, we do not explain our baseline analysis; See [20] for the details.

Three different alarm clustering analyses are implemented: 1) syntactic alarm clustering, 2) inter-procedural semantic clustering with interval domain, 3) intra-procedural semantic clustering with octagon domain. As we move from syntactic clustering to semantic clustering with octagon domain, we can cluster more alarms but need to pay more cost for the analysis. Thus, we initially use syntactic clustering to group alarms as many as possible and then apply the semantic clustering analyses to the rest of alarms that are not clustered yet.

In the rest of this section, we explain briefly about the implementation of each clustering analysis.

Syntactic Alarm Clustering. Syntactic alarm clustering is based on syntactically identifiable alarm dependencies. Two alarms are syntactically dependent iff 1) the expressions that raise the alarms are syntactically equivalent and 2) the

variables inside the expressions have the same definition points in the definition-use chain [18].

We implement syntactic alarm clustering as a post-analysis phase. The first check for a syntactic dependence is trivial and the second check can exploit the definition-use chain already computed by our baseline analyzer. Once we find dependencies, the alarm clustering part is the same as algorithm 1.

Note that the syntactic alarm clustering can be explained in our clustering framework. Syntactic alarm dependence is a special type of abstract dependence such that the abstract transfer function between two alarm points is identity upto the alarm-related variables, thus the falsehood of one alarm makes the other also false trivially.

Example 4. Our static analyzer reports four alarms in the following code snippet excerpted from `ftpd.c` in `Wu-ftpd 2.6.2`:

```
1  /* extern char *optarg; */
2  while (*optarg && *optarg >= '0' && *optarg <= '9')
3    val = val * 8 + *optarg++ - '0';
```

We can easily find that three alarms at line 2 have syntactic dependencies on each other. We also find that two alarms in line 2 and 3 are also syntactically dependent; two expressions that raise the alarms are syntactically the same (`*optarg`) and the definitions of `optarg` at line 2 and 3 are always the same (either the one defined before the loop or newly defined at line 3). □

From a practical point of view, syntactic alarm clustering is beneficial for two reasons. First, syntactic alarm clustering is highly cost-effective. It requires only an additional definition-use analysis which does not cost a lot. Especially, our static analyzer has been performing definition-use analysis for its own use. Second, syntactic alarm clustering is precise because it does not involve any abstraction.

Alarm Clustering with Interval and Octagon Domain. We implement algorithm 1 for both interval and octagon domain. The algorithms work after syntactic clustering algorithm find alarm clusters. The octagon domain enables us to find dependencies that are visible only by relational analysis.

One difference between the implementation with interval domain and octagon domain is that we use more fine-grained "refined-by" information (R in algorithm 1) in the implementation with interval domain. We track set of dominant alarm candidates not per each program point, but per each variable. By tracking dominant alarm candidates in this way, we could find more dependencies.

For octagon domain-based analysis, which has not been supported by our baseline analyzer, we integrate a prototype using Apron octagon domain library [8] into our clustering system. We only implement intra-procedural analysis (for cost reduction) and paralleize it. For each function, we do dependence analysis [18] to find the set of alarm-related variables and pack only those variables

Table 1. Alarm clustering results.
B : baseline analysis, **S**: syntactic alarm clustering, **I** : semantic alarm clustering with interval domain, **O** : semantic clustering with octagon domain.

Program	LOC	# Alarms				% Reduction				Time(s)		
		B	S	S+I	S+I+O	S	+I	+O	S+I+O	B	I	O
nlkain-1.3	831	124	118	96	93	5%	18%	2%	25%	0.17	0.03	0.1
polymorph-0.4.0	1,357	25	19	13	13	24%	24%	0%	48%	0.12	0	0.06
ncompress-4.2.4	2,195	66	50	38	30	24%	18%	12%	55%	0.54	0.03	0.69
sbm-0.0.4	2,467	237	230	185	125	3%	19%	25%	47%	2.28	0.3	1.15
stripcc-0.2.0	2,555	194	165	143	127	15%	11%	8%	35%	2.76	0.07	25.44
barcode-0.96	4,460	435	386	329	302	11%	13%	6%	31%	3.23	0.1	2.59
129.compress	5,585	57	56	29	29	2%	47%	0%	49%	2.46	0.02	0.19
archimedes-0.7.0	7,569	711	342	215	132	52%	18%	12%	81%	6.48	0.27	16.11
man-1.5h1	7,232	276	226	189	165	18%	13%	9%	40%	11.65	0.28	1.86
gzip-1.2.4	11,213	385	341	278	263	11%	16%	4%	32%	10.03	0.3	2.92
combine-0.3.3	11,472	733	468	297	294	36%	23%	0%	60%	19.74	0.81	26.93
gnuchess-5.05	11,629	976	744	343	333	24%	41%	1%	66%	42.49	4.78	8.66
bc-1.06	12,830	593	330	320	198	44%	2%	21%	67%	33.75	7.04	27.23
grep-2.5.1	31,154	115	100	96	85	13%	3%	10%	26%	4.19	0.01	11
coan-4.2.2	22,414	461	350	332	291	24%	4%	9%	37%	126.66	1.91	6.14
lsh-2.0.4	110,898	616	387	319	264	37%	11%	9%	57%	115.13	2.12	204.12
TOTAL	245,861	6,004	4,312	3,222	2,744	28%	18%	8%	54%	381.68	15.94	335.19

to make octagons. We use the straightforward translation between the baseline, interval analysis results and their octagon representations.

4.2 Experiment Results

We apply our clustering analyzer on 16 packages from three different categories (Bugbench [14], GNU softwares, and SourceForge open source projects). Table 1 shows our benchmark.

Effectiveness. To evaluate how much our clustering can reduce the alarm-investigation effort, we measure the number of distinct dominant alarms of alarm clusters and compare it to the number of reported alarms. In table 1, the columns labeled "# Alarms" show the numbers of alarms reported by baseline analyzer (B), reduced by syntactic clustering (S), reduced further by semantic clustering with interval domain (S+I), and reduced further by semantic clustering with octagon domain (S+I+O), respectively. The next columns labeled "% Reduction" show the reduction ratios of each additional alarm clustering analysis (S, +I, and +O) and the total (S+I+O).

As shown in table 1, our alarm clustering reduces 54% of alarms on average. Note that even though the syntactic clustering reduces 28% of alarms, the semantic clustering reduces 26% additionally (18% by clustering with interval domain and 8% by the other). This means that semantic clustering analyses successfully find intricate alarm dependencies which can never be found by syntactic clustering.

We investigate the most effective and the least effective cases of the interval domain-based alarm clustering. Our interval domain-based algorithm turned out to be the most effective for gnuchess-5.05 and 129.compress (reduced by 41% and 47%) 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 refinement by refutation, even using interval domain. Second, there were many different buffers of the same size which are accessed using the same index variable. On the other hand, our interval domain-based clustering is least effective for grep-2.5.1 (reduced by 3%). It is because almost all buffers in the program are dynamically allocated, thus the sizes of them were hard to track accurately. Indeed, we found that the interval values of the sizes of buffers 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.

For programs polymorph-0.4.0, 129.compress, combine-0.3.3, and gnuchess-5.0.5, octagon domain-based clustering is not effective. The reason of ineffectiveness for the first three programs is rather originated from our implementation, which has been only doing intra-procedural analysis. Indeed, program polymorph-0.4.0 has many library function calls between alarm points, so that they ruin the refinement. In the case of gnuchess-5.0.5, many buffers were accessed by indices with bit operations on them, which is beyond the reach of octagon domain.

We also investigate the most effective case of the octagon domain-based alarm clustering. The most effective case was program sbm-0.0.4. The program has long consecutive buffer accesses with the indexes having relationship of form $\pm i \pm j = c$. This type of relationship can be precisely expressed and handled by octagon domain.

Clustering Overhead. We measure the analysis time to assess the overhead of clustering analysis. All our experiments are performed on a PC with a 2.4 GHz Intel Core2 Quad processor and 8 GB of memory. In table 1, the columns labeled "Time" present times for the baseline analysis (B) and the additional alarm clustering with interval domain (I) and octagon domain (O). Note that we do not measure the cost of syntactic clustering since it exploits the definition-use chains already generated by the baseline analysis.

The overhead of interval domain-based alarm clustering is on average only 4% of the baseline analysis time. On the other hand, we find that the overhead of octagon domain-based clustering is almost close to, and even surpasses for some cases, the baseline analysis time. This is because octagon domain-based static analysis usually has higher cost than interval domain-based static analysis and our octagon domain-based abstract interpreter is prototypical and far less optimized than interval domain-based one which has been highly optimized [9, 19, 20].

5 Related Work

To our best knowledge, Le et al.'s work [23] is the first one that proposes non-statistical clustering method. They reduce the number of faults (alarms) by detecting correlations (dependencies) between them. By propagating the effects of the error state along the program path, they detect the correlation of pairs

of alarms. They automatically construct a correlation graph which shows how faults are correlated. Based on the graph, we can reduce the number of faults to consider.

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

Statistical ranking schemes [7, 10, 12, 13] may help to find real errors quickly, but ranking schemes do not reduce alarm-investigation burdens as in our work. Since our technique is orthogonal to statistical ranking schemes, we might combine our technique with them for a more sophisticated alarm reporting interface.

Our work resembles to Rival's work [22] in the sense that both work refines the abstraction by exploiting the information about error state. In his work, Rival refines the abstraction by slicing out non-error states and sees if the initial state after refinement still insists that the erroneous states are reachable. If the initial state becomes bottom after refinement, the alarm turns out to be false. On the other hand, in our work, we refine the abstraction by slicing out erroneous states at one point and see if erroneous states at other points become non-reachable, which means that we found the dependence between alarms.

Our work is more general than error recovery technique that is used for reducing false alarms in many commercial static analysis tools [1, 15, 17]. For each alarm found, the commercial analyzers recover from those alarms; i.e. they assume that an alarm is false when they passed the alarm point. Because error recovery is done within the baseline analysis, possible refinements are bounded by the expressiveness of the abstract domain of the baseline. As we show in Section 4, we can use more expressive domain for clustering purpose than the one used in the baseline, which can be more cost-effective than using expensive abstract domain in the baseline. Additionally, our method can derive true clusters for which cannot be done by the error recovery technique.

Our clustering method can be integrated with other refinement approaches [5, 6, 11, 22]. The goal of them is to remove false alarms by abstraction refinement, while our work is to reduce the number of alarms to investigate. Our work can reduce the number of targets to do the refinement.

6 Conclusion

We have presented a new, sound non-statistical alarm clustering method for semantic-based static analyzers. We propose a general framework of alarm clustering. Our technique is general enough to be applicable to any static analysis based on abstract interpretation. By experiment results, we show that our technique can considerably reduce the number of alarms to investigate manually.

Acknowledgment The authors would like to thank Youil Kim, Daejun Park, Hakjoo Oh, Minsik Jin, and the anonymous referees for their comments in improving this work.

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI. pp. 196–207 (2003)
2. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming 13(2–3), 103–179 (1992)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252 (1977)
4. Cousot, P., Cousot, R.: Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In: PLILP. pp. 269–295. Springer-Verlag, London, UK (1992)
5. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: TACAS. Lecture Nodes in Computer Science, vol. 4963, pp. 443–458 (2008)
6. Gulavani, B.S., Rajamani, S.K.: Counterexample driven refinement for abstract interpretation. In: TACAS. LNCS, vol. 3920, pp. 474–488. Springer (2006)
7. Heckman, S.S.: Adaptively ranking alerts generated from automated static analysis. Crossroads 14, 7:1–7:11 (2007)
8. Jeannet, B., Mine, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV. pp. 661–667 (2009)
9. Jhee, Y., Jin, M., Jung, Y., Kim, D., Kong, S., Lee, H., Oh, H., Park, D., Yi., K.: Abstract interpretation + impure catalysts: Our Sparrow experience. Presentation at the Workshop of the 30 Years of Abstract Interpretation, San Francisco (2008)
10. Jung, Y., Kim, J., Shin, J., Yi, K.: Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In: SAS. pp. 203–217 (2005)
11. Kim, Y., Lee, J., Han, H., Choe, K.M.: Filtering false alarms of buffer overflow analysis using smt solvers. Inf. Softw. Technol. 52(2), 210–219 (2010)
12. Kremenek, T., Ashcraft, K., Yang, J., Engler, D.R.: Correlation exploitation in error ranking. In: FSE. pp. 83–93 (2004)
13. Kremenek, T., Engler, D.R.: Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In: Symposium on Static Analysis. pp. 295–315 (2003)
14. Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., Zhou, Y.: Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the Evaluation of Software Defect Detection Tools (2005)
15. MathWorks: Polyspace embedded software verification, `http://www.mathworks.com/products/polyspace/index.html`
16. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP. Lecture Notes in Computer Science, vol. 3444, pp. 5–20. Springer-Verlag (2005)
17. Microsoft: Code contracts, `http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`
18. Muchnick, S.S.: Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)
19. Oh, H.: Large spurious cycle in global static analyses and its algorithmic mitigation. In: APLAS. Lecture Notes in Computer Science, vol. 5904, pp. 14–29. Springer-Verlag, Seoul, Korea (2009)

20. Oh, H., Brutschy, L., Yi, K.: Access analysis-based tight localization of abstract memories. In: VMCAI. pp. 356–370. Springer-Verlag, Berlin, Heidelberg (2011)
21. Oh, H., Yi, K.: An algorithmic mitigation of large spurious interprocedural cycles in static analysis. Software: Practice and Experience 40(8), 585–603 (2010)
22. Rival, X.: Understanding the origin of alarms in astrée. In: SAS. Lecture Notes in Computer Science, vol. 3672, pp. 303–319. Springer (2005)
23. Wei Le, M.L.S.: Path-based fault correlations. In: FSE (2010)