ISSN 1330-3651 (Print), ISSN 1848-6339 (Online)
DOI: 10.17559/TV-20150317102804

RESEARCH OF ALARM CORRELATIONS BASED ON STATIC DEFECT DETECTION

Dalin Zhang, Dahai Jin, Yunzhan Gong, Siru Chen, Chengcheng Wang

Original scientific paper
Traditional static defect detection tools can detect software defects and report alarms, but the correlations among alarms are not identified and massive independent alarms are against the understanding. Helping users in the alarm verification task is a major challenge for current static defect detection tools. In this paper, we formally introduce alarm correlations. If the occurrence of one alarm causes another alarm, we say that they are correlated. If one dominant alarm is uniquely correlated with another, we know verifying the first will also verify the others. Guided by the correlation, we can reduce the number of alarms required for verification. Our algorithms are inter-procedural, path-sensitive, and scalable. We present a correlation procedure summary model for inter-procedural alarm correlation calculation. The underlying algorithms are implemented inside our defect detection tools. We chose one common semantic fault as a case study and proved that our method has the effect of reducing 34,23 % of workload. Using correlation information, we are able to automate the alarm verification that previously had to be done manually.

Keywords: *abstract interpretation; alarm correlations; alarm verification; correlation summary; state slicing*

Istraživanje korelacija alarma na temelju detekcije statičkog kvara

Izvorni znanstveni članak
Tradicionalni alati za detekciju statičkog kvara mogu detektirati kvarove softvera i objaviti alarm, ali korelacije između alarma nisu identificirane i masivni nezavisni alarmi protivni su razumijevanju. Pomaganje korisnicima u verifikaciji alarma predstavlja veliki izazov postojećim alatima za detekciju statičke greške. U ovom radu mi formalno uvodimo korelacije alarma. Ako postojanje jednog alarma uzrokuje drugi, kažemo da su u korelaciji. Ako je jedan dominantni alarm jedinstveno povezan s drugim, znamo da će se verifikacijom jednoga također verificirati drugi. Na osnovu korelacije možemo reducirati broj alarma potrebnih za verifikaciju. Naši su algoritmi inter-proceduralni, osjetljivi na putanju i podesivi (scalable). Mi prikazujemo sumarni model postupka korelacije za računanje inter-proceduralne korelacije alarma. Osnovni algoritmi su implementirani u naše alate za detekciju kvara. Izabrali smo jednu uobičajenu semantičku pogrešku za analizu slučaja i dokazali da naša metoda rezultira smanjenjem radnog opterećenja za 34,23 %. Primjenom korelacijeske informacije možemo automatizirati verifikaciju alarma, što se ranije moralo raditi ručno.

Ključne riječi: *apstraktna interpretacija; korelacije alarma; verifikacija alarma; pregled korelacija; provjeravanje (slicing) stanja*

1 Introduction

Traditional static defect detection tools can detect software defects [1 ÷ 4], e.g., null-pointer dereference (NPD), invalid arithmetic operations (IAO), and memory leak (ML), and generate independent atomic warnings automatically, but they do not take the influences among different alarms into account and identify the correlations among alarms [5]. In an actual software testing process, the verification of software warnings is still done manually while the code around the alarm can be complex sometimes and it is time-consuming. For the large newly developed software, hundreds, or even thousands of warnings are generated by tools, but generated reports are processed at a very low speed. Excessive warning generation and a large proportion of incorrect warnings may cause developers to reject the use of static analysis tools. Helping users in the alarm verification is a major challenge for current static defect detection tools [6, 7].

In this paper, we propose a framework for the investigation of the alarms, so as to help classify them by their correlations. We explore relationships among alarms for verification, show that a correlation can exist between alarms. Once we find these groups of alarms, we only need to check whether their dominant alarm is false positive or a real defect.

Our goal is to provide support in the alarm investigation process, especially in how to reduce alarm identification burden. We propose an approach resorting to automatic, sound static analysis techniques to refine an initial abstract semantics by an alarm reported by our static analyzer. If another alarm does not take place in a refined semantics, we believe they are correlated.

The contribution of the paper is both theoretical and practical. The details are in the following:

- We provide two rigorous definitions of alarm correlation and abstract alarm correlation, and then prove the soundness based on abstract interpretation.
- We propose an approach to automatically computing correlations based on forward analysis, state slicing and input constraint. Meanwhile, a sound alarm correlation framework is presented. It is general and applicable to any semantic-based static defect detection tools.
- For the inter-procedural alarm correlation calculation, we present a procedure correlation summary model and describe a general technique for constructing and instantiating, which makes our approach distinguished from others.
- A prototype of our framework is implemented as an extension to DTS [8 − 10], a general automatic static defect detection tool for GCC programs developed by ourselves. Preliminary experimental results are encouraging. We choose one common semantic alarm as a case study and prove that the approach can effectively reduce 34,23 % of the workload to identify all alarms.

The rest of this paper is organized as follows: Section 2 gives one illustrative motivation example. Section 3 presents the fundamentals of alarm correlation, introducing a procedure correlation summary model for inter-procedural alarm correlation calculation. Algorithms for computing alarm correlation are presented in Section4. In Section 5, we introduce the experimental results.

Related work and conclusion are presented in Section 6 and Section 7.

2 Motivating example and insight

In this section, we show that a correlation can exist among alarms. We provide an informal overview of the various challenges faced with alarm verification while leveraging alarm correlations technique. We present main ideas at a semi-technical level using a motivation example.

```
static SSLConnRec *ssl_init_connection_ctx(conn_rec *c)
{
//...
sslconn = apr_pcalloc(c->pool,sizeof(*sslconn));//line359
sslconn->server = c->base_server; //npd1,line361
sslconn->verify_depth = UNSET; //npd2,line362
//...
return sslconn; // line 367
}
                                    (a)
int ssl_engine_disable(conn_rec *c)
{
//...
sslconn = ssl_init_connection_ctx(c);//line406
sslconn->disabled = 1; //npd3,line408
//...
}
                                    (b)
```

Figure 1 Three Alarms from httpd-2.4.4

Fig. 1 shows an example of alarm correlations found in httpd-2.4.4. It presents how alarm investigation efforts are greatly reduced. These 3 alarms are reported by the static defect detection tool (DTS) and these correlation relationships are discovered automatically by correlation algorithms.

In the program snippet (a) of Fig. 1, the procedure $apr\_pcalloc()$ is summarized first with a return value $[null\ or\ notnull]$ by the analyzer at line 359. At line 361 and 362, two NPD alarms are reported. The variable $sslconn'$s value is returned on line 367, and procedure *ssl_init_connection_ctx()* is summarized with a return value $[null\ or\ notnull]$ by the analyzer. Then the *ssl_init_connection_ctx()* function is called to allocate the memory in other procedures (see the program snippet (b)) and the analyzer reports the alarm *npd3*. However, in fact, the procedure *ssl_init_connection_ctx()*will never fail to allocate memory, since in the case of httpd-2.4.4. If the allocation fails, a function is registered that gracefully shuts down the particular server process, and *apr_palloc()* is in the process guaranteed to never return$[null]$. So these 3 alarms are all false positives. If they are reported separately, the end user will have to investigate them for 3 times while the code around the alarm can be complex sometimes.

Obviously, in procedure *ssl_init_connection_ctx()* (Fig. 1(a)), if *npd1* is a false positive, the *npd2* will also be the one. We call them as the intra-procedural correlated. Meanwhile, *npd1* has an inter-procedural correlation with *npd3*, and *npd1* is the predominant alarm. If the predominant alarm is proved false, then others correlated with it are also false. So we only need to identify *npd1* instead of all alarms, seeing that alarm correlation can improve the efficiency of warning verification.

According to the analysis above, the key technical challenges facing alarm correlation concern two aspects. First, how to calculate the alarm correlation between two alarms with uniformity and accuracy, where indicates the isomorphic representation whatever alarms are false positives or true errors. Meanwhile, accuracy implies computational results are reliable. Second, how to construct a general-purpose procedure correlation summary for the inter-procedural alarm correlation calculation is a key issue. Our algorithm overcomes these challenges with the following insights.

DTS analyzes programs based on a forward dataflow analysis and takes the external input into account in a real world program, which also might affect the results at a precise time in the execution. DTS dealing with these external inputs is based on over-approximating techniques. It is difficult to capture the impact of one alarm on another directly. In this study, we take the advantage of the ideas of external input constraints. More specifically, by giving two alarms α and β, we slice α's error state and treat the $state\ slicing$ at a program point as an external input, then get a refined semantic based on an assumed external input constraints. We just judge whether β is reported at the refined semantic. If β is reported, we call β is correlated with α.

Each alarm's $state\ slicing$ might affect its procedures and its concrete call site contexts in two aspects: (1) the $state\ slicing$ in one call procedure might cause side effects to actual-parameters and global variables, (2) the $state\ slicing$ might affect the procedure's return value, potential interrupt instructions, the caller's dataflow and control flow. As a result, by concentrating on the above information, a unified procedure correlation summary model could be constructed, which can uniformly and precisely reflect the call effect of each alarm's $state\ slicing$ in every procedure.

3 Alarm correlation

In this section, we first briefly describe the background of our static analysis which is the basis of the subsequent sections. In the following, concrete program semantics refers to trace semantics [11, 12], which observes the history of each possible computation step by step. Abstract semantics refers to any approximations of trace semantics. According to the research of the pioneers, one single cause of imprecision at some program point often leads to many false alarms in the code reachable from that program point later [6], so a single refinement typically eliminates many false alarms. Furthermore, the same root cause of a defect appears several times in the program. In this section, an alarm correlation notion was formally introduced to optimize static analysis reports, providing the definition of alarm correlation and abstract alarm correlation.

A. Preliminaries

We describe a program with a transition system and assume that a procedure is defined by the data of a tuple

$(\mathbb{L}, \mathbb{X}, \mathbb{V}, \rightarrow, \mathbb{S}^i)$. $\mathbb{L}$ stands for the set of program point of execution; $\mathbb{X}$denotes a finite set of variables; $\mathbb{V}$ denotes a set of values stored in the memory; a function $\sigma \in \mathbb{M}$ describes the variables and values correspondingly, where $\mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$. We usually write $\mathbb{S}$ for the set of execution states and $\mathbb{S}^i$ for the set of initial states,$\mathbb{S}^i = \{l^i\} \times \mathbb{M}$ where $l^i \in \mathbb{L}$ are the entry program points. We should add a special error state Ω since real world programs may crash. We also consider inter-procedural programs, that a control state is defined by a pair (k, l), where k is a calling function name and l a syntactic control point. A state in an inter-procedural program is a tuple in$(\mathbb{K} \times \mathbb{L}) \times \mathbb{M}$.

The program semantics formalizes its behaviour and the precision of its description depends on the level of abstraction of the considered semantics and the level of abstraction domains over which the semantics is computed.

An execution of a program is represented with a sequence of execution states, called as a trace; This semantics is accurate but not decidable in abstract interpretation semantics systems. The written $[\![P]\!]$ stands for the trace semantics of a program P, including more detailed introduction of trace semantics see[11].

By giving a program and a safety property, we wish to either validate that the code respects the property, or find an execution path that shows how the code violates the property. The static analyzer over-approximates the set of reachable dangerous states and reports corresponding alarms.

Alarm. At a program point l of program P , if$\gamma_{\mathbb{M}} T(l) \cap \mathcal{F}(l) \neq \emptyset$, then, the static analysis tool reports an alarmψ_l, where T and $\mathcal{F}$ are the abstract semantics of $[\![P]\!]$ and Ω separately, $\mathcal{F}(l)$ is an error state function, and$\mathcal{F}(l) : \mathbb{L} \rightarrow \mathcal{F}$.

Alarm is a major issue for end users. Indeed, when the analyzer reports an alarm, it could be either a false alarm or a real bug that should be fixed. Currently, the alarm investigation process mainly relies on the manual inspection of variants, partly with a graphical interface. This process turns out to be cumbersome, since even simple alarms may take days to classify [7]. In the rest of this section, we explore relationships among alarms for investigation. We define the notion of alarm correlation and abstract correlation. We also prove the soundness of the abstract correlation among these correlated alarms. Once we find these groups of alarms, we only need to check whether their predominant alarm is false positive or a real defect.

B. Correlation Definition

We first give the definition of *alarm correlation* in concrete semantics, and formally introduce two notions *semantic slicing* and*error state slicing*, then prove the soundness based on abstract interpretation. Finally, this subsection gives the definition of *abstract alarm correlation*, which establishes the basis for the following implementation in our static analyzer.

Alarm Correlation. In program$P$, if a static analysis tool reports two alarms ψ_m and ψ_n respectively from m and n along a trace σ of$P$, $m, n \in \mathbb{L}$, suppose alarm ψ_n will never occur when alarm ψ_m does not occur, we say that ψ_m have a correlation relationship withψ_n, and write $\psi_m \rightarrow \psi_n$.

In static analysis, real faults are often mixed with an overwhelming number of false alarms. By being given two alarms ψ_m and ψ_n, there are two situations for alarm correlations: (1) if alarm ψ_n is always false alarm when alarm ψ_m is false alarm, we say that ψ_n is correlated with ψ_m . We denote the correlation as $\psi_m \rightarrow \psi_n$, i.e. $[\![P]\!](m) \cap \Omega(m) = \emptyset \Rightarrow [\![P]\!](n) \cap \Omega(n) = \emptyset$, $\psi_m \rightarrow \psi_n$, and (2) if alarm ψ_n is always real fault when alarm ψ_m is real fault, we say that ψ_n is correlated with ψ_m . We denote the correlation as $\psi_m \rightarrow \psi_n$, i.e. $[\![P]\!](m) \cap \Omega(m) \neq \emptyset \Rightarrow [\![P]\!](n) \cap \Omega(n) \neq \emptyset$, $\psi_m \rightarrow \psi_n$.

Since the concrete alarm correlation is not computable in general, we use abstract alarm correlation which is sound in replacement of concrete alarm correlation. The idea is that if static analysis tools do not report the alarm ψ_n from the abstract semantics refined under the assumption that alarm ψ_m does not occur, we can say that ψ_m has a concrete correlation withψ_n. It is easy to notice 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 holds.

At some abstract domain, if we cut the error state, the static analysis tool will not report this alarm at this point in the program. In the rest of the section, we define the notion of state slicing and abstract correlation. We also prove the soundness of abstract correlation.

State Slicing. In the program P, if a static analysis tool reports an alarm ψ_l at a program point$l$, we can get a state slicing by slicing the error state at this program point$l$. We can define the *state slicing* $\bar{\rho}_{\psi_l} = [\![P]\!](l) \ominus \Omega_{\psi}(l)$, we let $\Omega_{\psi}(l)$ for the concrete error state at program point l in the execution of the program, $\ominus$ denotes a concrete operation of error state slicing.

Accordingly, we can define an *abstract state slicing* $\overline{d}_{\psi_l}$: $\overline{d}_{\psi_l} = T(l) \mathbin{\widehat{\ominus}} \mathcal{F}(\psi_l)$, and we denote $\widehat{\ominus}$ for the sound abstract error state slicing operation. In the concrete practice, static analysis tools need to implement this abstract resection operation at some abstract domain.

In a real world application, there can be a lot of external input, which also might affect the results at precise time in the execution of the program. Most present tools deal with these external inputs based on over-approximation techniques. In this study, we take advantage of the ideas of external input constraints, treating the state slicing at a program point as an external input, and then get a *refined semantic* based on an assumed external input constraint.

Input. In program P, given a set of *input* program points: $\mathbb{L}_{in} = \{l \in \mathbb{L} | l : input(x_l)\}$. An *input* program point is a function $\delta : \mathbb{L} \rightarrow \mathcal{P}(\mathbb{V})$, mapping a program point to the set of values that may be read at this point. δallows to select different inputs for different execution contexts at the same program point. The denotation of the input function δ is the set of traces $\gamma_{\mathbb{V}}(\delta) = \{\langle (l_0, \rho_0), \ldots, (l_n, \rho_n) \rangle | \forall i, l_i \in \mathbb{L}_{in} \Rightarrow \rho_i(x) \in \delta(l_i)\}$,

and such traces satisfy the property that reading an input at label l yields a value in$\delta(l_i)$.

We do not consider the real application of external input. Indeed, we slice the error states as a kind of external input constraints, replacing the original program semantics, denoted as $\mathbb{L}_{in} = \{l \in \mathbb{L}_{\mathcal{F}} | l : input(x_l)\}$ and $\mathbb{L}_{\mathcal{F}}$ denotes a finite set of alarm program points, accordingly, input function $\delta_{\mathcal{F}}$ can be further described:

$\gamma_{\mathbb{V}}(\delta_{\mathcal{F}}) = \{\langle (l_0, \rho_0), \ldots, (l_n, \rho_n)\rangle | \forall i, l_i \in \mathbb{L}_{\mathcal{F}} \Rightarrow \bar{\rho}_{\psi_i}(x) \in \delta_{\mathcal{F}}(l_i)\}$.

*Refined semantic.*Consider the case where we slice the error states as a kind of external input constraints, the semantic T can be further refined, the refined semantic $\widehat{T}_{\psi_l}$ can be presented as $\widehat{T}_{\psi_l} = T \cap \gamma_{\mathbb{V}}(\delta_{\mathcal{F}}(l))$. Suppose $\delta^{\sharp}_{\mathcal{F}}(l)$ is a sound abstraction of $\delta_{\mathcal{F}}(l)$, i.e. $\{\bar{\rho} \in \mathbb{M} | \bar{\rho}_{\psi_l}(x) \in \delta_{\mathcal{F}}(l)\} \subseteq \gamma(\delta^{\sharp}_{\mathcal{F}}(l))$.

Abstract refined semantic. The abstraction of the $refined\ semantic$ can be defined: $\widehat{T}'_{\psi_l} = \mathrm{lfp}^{\sharp} F^{\sharp} \sqcap \delta^{\sharp}_{\mathcal{F}}$, $\widehat{T}'_{\psi_l}$ is a sound approximation of $\widehat{T}_{\psi_l}$, i.e. $\widehat{T}_{\psi_l} \subseteq \gamma(\mathrm{lfp}^{\sharp} F^{\sharp} \sqcap \delta^{\sharp}_{\mathcal{F}})$.

Since concrete correlation is not computable in general, we use an approximation correlation which is sound with respect to concrete correlation. The approximation of alarm correlation is based on *abstract state slicing* and *abstract refined semantic*.

*Abstract correlation.*The static analysis tool reports two alarms ψ_m andψ_n respectively from m and n along a trace σ of P, $m, n \in \mathbb{L}$, $\dashrightarrow$ denote an abstract correlation relationship between two alarms. Suppose $\psi_m \dashrightarrow \psi_n$, if and only if ψ_n is not reported by static analysis tool under the *Abstract refined semantic* $\widehat{T}'_{\psi_m}$, i.e., if and only if $\gamma(\widehat{T}'_{\psi_m}(\psi_n)) \cap \Omega(n) = \emptyset$, $\psi_m \dashrightarrow \psi_n$. The following proves the soundness of *abstract correlation*.

Demonstrate: Suppose the static analysis tool reports two alarms ψ_m and ψ_n, and $\psi_m \dashrightarrow \psi_n$. Conveniently, by the definition of *abstract correlation*, then $\gamma(\widehat{T}'_{\psi_m}(\psi_n)) \cap \Omega(n) = \emptyset$. By the definition of the abstract interpretation, we can see $\widehat{T}_{\varphi_m}(\varphi_n) \sqsubseteq \gamma(\widehat{T}'_{\varphi_m}(\varphi_n))$, so we can get $\widehat{T}_{\varphi_m}(\varphi_n)) \cap \Omega(\varphi_n) = \emptyset$. Based on the definition of *alarm correlation*, it is easy to see that $\widehat{T}_{\varphi_m}(\varphi_n)) \cap \Omega(\varphi_n) = \emptyset \Leftrightarrow \varphi_m \rightarrow \varphi_n$. So $\psi_m \dashrightarrow \psi_n$ is a sound abstraction of $\psi_m \rightarrow \psi_n$.

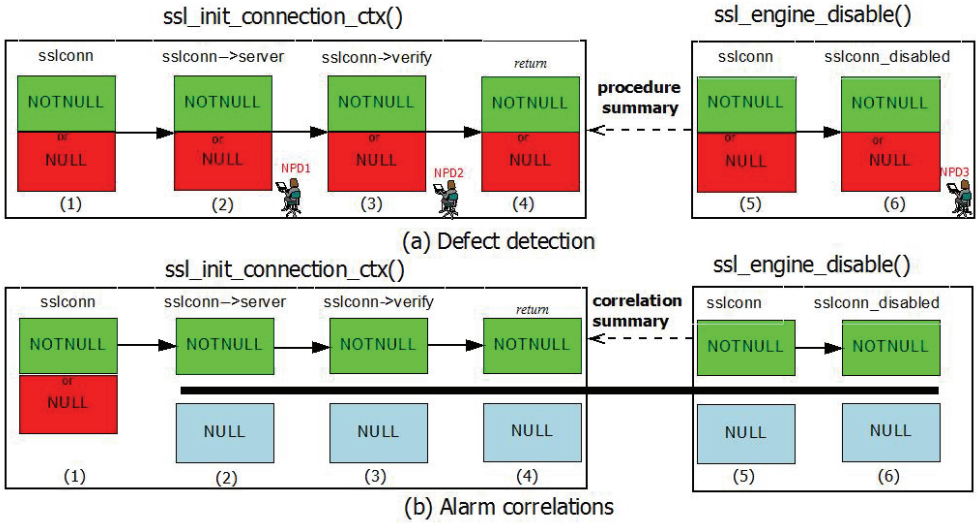

Figure 2 Defect detection and alarm correlation of 3 alarms

Taking the procedure *ssl_init_connection_ctx()* as an example (Snippet (a) of Fig. 1), the process of defect detection and alarm correlation is shown in the following Figs. 2(a) and 2(b). In the left of Fig. 2(a), there are two alarms reported (*npd1* and *npd2*) at node (2) and (3) separately in the procedure *ssl_init_connection_ctx()*, because the variable *sslconn* contains a [*null*] value. In the left of Fig. 2(b), when we slice the [*null*] value of *sslconn*, the alarm will not be reported at node (2), also at node (3). So, we can say that npd1 has an intra-procedural correlation with *npd2*, which denotes as *npd1*→ *npd2*.

C. Inter-procedural alarm correlations

DTS achieves both path-sensitive intra-procedural analysis and context-sensitive inter-procedural analysis. DTS addresses inter-procedural problem by introducing a unified symbolic procedure summary model and describes a general technique for constructing and instantiating. What information to capture in each procedure summary has been carefully tuned so that the summary should not lose any common defect-related behaviour? Because of limited space, we do not explain our DTS inter-procedural analysis, see [8, 10] for details.

(1) Procedure correlation summary model

As shown in the right of Fig. 3(a), although the procedure summary technology has been successfully used inDTS, it does not take the influence among alarms into account between two procedures. Such as in Fig. 3(b), we slice the error state in parallel when *npd1* is reported at the node (2). It becomes even more obvious that the *state slicing* affects procedure *ssl_init_connection_ctx()*'s return value and the caller *ssl_engine_disable()*'s dataflow value in the right of figure 3 (b). To be specific, if the return value of *ssl_init_connection_ctx()* is *notnull*, the value of $sslconnl- > disabled$ is also *notnull* in

ssl_engine_disable(). So, we can say that *npd1* have an inter-procedural correlation with *npd3*, which is denoted as $npd1 \rightarrow npd3$.

As discussed in Section 2, an alarm's state slicing might affect its procedure and further lead to two types of call site context transformation, which corresponds to two aspects of our *correlation summary model*. In a specific invocation to procedure Q, there is a set of alarms Ψ reported. The construction of our alarm *correlation summary model* follows the following description:

$$PostCond(Q,\Psi) = \bigcup_{g \in globals}^{p \in params} \overline{d}(exit_Q, \overline{d}_\psi, p, g) | \psi \in \Psi, \quad (1)$$

$$ReFeature(Q,\Psi) = \bigcup_{ret \in ReturnStmt}^{\psi \in \Psi} (\overline{d}(ret, \overline{d}_\psi) \wedge \overline{d}_{cc}(ret, \overline{d}_\psi)), \quad (2)$$

$$IntFeature(Q,\Psi) = \bigcup_{inter \in InterruptStmt}^{\psi \in \Psi} \overline{d}_{cc}(inter, \overline{d}_\psi). \quad (3)$$

$PostCond(Q,\Psi)$ means that the ψ's error state slicing $\overline{d}_\psi$ in one callee procedure might cause side effects to actual-parameters and global variables. We consider the error state slicing $\overline{d}_\psi$ as a kind of external input constraints, the domain of parameters and global variables can be further refined, the $\overline{d}(exit_Q, \overline{d}_\psi, p, g)$ means the parameters and global refined domain information.

$RetFeature(Q,\Psi)$ and $IntFeature(Q,\Psi)$ indicates that the ψ's error state slicing $\overline{d}_\psi$ might affect the procedure return value and potential interrupt instructions respectively. The $\overline{d}(ret, \overline{d}_\psi)$ and $\overline{d}_{cc}(ret, \overline{d}_\psi)$ are the refined information of return value $d(ret)$ and interrupt information $d_{cc}(ret)$ accordingly. Each procedure is summarized by convergent fixpoint iteration based on abstract interpretation, so the analyzer can naturally handle arbitrary call cycles from direct or indirect recursive calls.

(2) Correlation summary instantiation

We now show how to perform context-sensitive correlation summary instantiation at concrete call sites. Consider a procedure call $P \xrightarrow{n} Q$, and we have got the correlation summary of callee Q. The call site context (CSC) consists of all dataflow information right before the procedure call sites. While encountering Q at call site v, the precise summary is extracted by comparing $CSC(n)$ and the conditional constraints binding with the summary. Then we leverage the refined correlation summary in two different manners: (1) if the summary information is concrete deterministic abstract domains, we will update the call site context using these fresh dataflows for subsequent detection and (2) if the summary is represented by some symbolic expressions, we employ the reversed mapping function to decide which symbol should be substituted by the actual parameter, and the dataflow iterates continually in this manner [8].

4 Computing alarm correlation

As shown in the above section, we need to calculate the abstract alarm correlation to reveal the concrete correlations between alarms which are reported by static analyzers. Our approach has three major phases. In the first phase, we employ a data flow analysis to update the state information and refined state information at one node of a *Control Flow Graph* (CFG). The phase of state updating is shown on the top side of Fig. 3. Next, we run defect detection, and if the static analysis tool reports an alarm, we will go to the final phase and run the alarm correlation computing. We simulate the propagation of the new state along one trace to determine its impact on the occurrence of other alarms. If the alarm reported at the second is not reported under the refined state information set, we report the alarm correlation information and slice out the error state at this alarm point where the alarm occurs. The goal of the third phase is to identify whether an error state that was removed at an alarm point can influence another alarm reported in the second phase. The second and third phases are shown on the bottom side of Fig. 3.

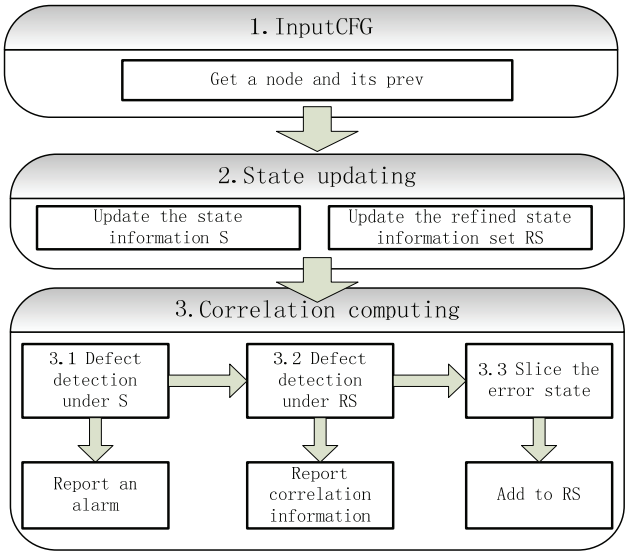

Figure 3 The flowchart of our correlation algorithm

Based on the above idea, we give a specific algorithm to calculate the alarm correlation in this section. We embed our alarm correlation algorithm in the process of defect detection. It takes as input a safety policy set and a CFG. Our goal is to identify all the correlations for the alarms reported by our static analysis tool at each node on CFG. We now describe these algorithms formally. High-level pseudocode for the correlation algorithms is given in the following.

Algorithm 1 Compute Alarm Correlations

Input: Defect finite State AutoMaton(DFSM) $dfsm$;
Control Flow Graph(CFG) $cfg < N, E >$
Output: Alarms and alarms correlation information

1. *OUT*[entry]: =*dfsm*{$*start* :⊥}
2. set *begin*: =false, *firstdefect*: =*true*;
3. set *RefSemSet* **: =***null***;** $\hat{T}'_{\varphi_i}(n)$ **: =***null***;**

4. for each *n in N* except *entry* do
5. *call DefectDetection*(*dfsm,n*);
6. if(*begin*) then
7. *call DefectCorrelation*(*n,RefSemSet*);
8. if (*firstdefect*) then
9. *Stateslice*(*def*), add $\hat{T}'_{def}(n)$ into *RefSemSet*;
10. set *firstdefect*: =*false*;
11. end do

In Algorithm 1, we set the state of $dfsm$ to $start$ and set the $state\ conditions$ of $dfsm$ to $\perp$(line 1) at the entry node of cfg. In our static analysis tool, property state conditions are represented by abstract domain of variables, and infeasible paths can be identified when some variables' abstract value range is empty. This method avoids the combination explosion of full path analysis by merging the conditions of identical property state at join points in the CFG. Lines 2-5 initialize the abstract domain of variables $domain$, the slicing of abstract state $\overline{d}_{\psi_l}$, and refined abstract domain set $refdomainset$, by setting them to $null$.

Algorithm 2 Compute abstract state

Input: Defect finite State AutoMaton(DFSM) $dfsm$;
Control Flow Graph(CFG) $cfg < N, E >$
Output: Abstract domain value
1. procedure *CallDomainSet*(*Node n*){
2. *calulateIn*(*n*);
3. *calulateOut*(*n*);
4. }

In Algorithm 2, the procedure CALLDOMAINSET applies a traditional data flow analysis to compute a prefixpoint of a decreasing chain. We get a previous abstract state by calculating the $domain$ of $In\text{-}Edge$ and compute a new abstract state $domain'$ at the $Out\text{-}Edge$ (line 2 and 3).

Algorithm 3 Defect detection

Input: Defect finite State AutoMaton(DFSM) $dfsm$;
Control Flow Graph(CFG) $cfg < N, E >$
Output: Alarms information
1. procedure *DefectDetection*(*dfsm,n*){
2. call *CalDomainSet*(*n*);
3. *update state conditions and remove contrary states*;
4. if(*$error*) then
5. report defect *def*;
6. if *RefSemSet !=null* then
7. set *begin*:= *true*;
8. }

In Algorithm 3, procedure DEFECTDETECTION is one of the core functional components of our static analysis tool. The finite state machine is a formal description of defect patterns. States in the automaton correspond to typestates that dataflow variants can occupy during execution. A designated typestate *error* corresponds to an erroneous state. Transitions in the automaton correspond to potential operations that might change the variant's typestate. Once transiting to the *error* state, there exists a potential defect. The $Defect\ Finite\ State\ autoMaton\ (DFSM)$ is a simple and common used model to extract semantics of the procedure execution. We transit the $DFSM$ states via the interval analysis results, until encountering the *error* state which indicates a potential defect, or transit to the *end* state while the $DFSM$ destroy automatically.

Algorithm 4 Abstract Correlations

Input: Defect finite State AutoMaton(DFSM) $dfsm$;
Control Flow Graph(CFG) $cfg < N, E >$
Output: Alarms correlation information
1. procedure *AlarmCorrelation*(*Node n, RefSemSet*){
2. for each $\hat{T}'_{\varphi_i}(n)$ in *RefSemSet* do
3. call *CalDomainSet*(*n*);
4. *DefectDetection*(*dfsm,n*);
5. if not report defect *def* then
6. report *defect correlation information*;
7. end do
8. }

In Algorithm 4, procedure ALARMCORRELATION is a process of calculating $abstract\ correlation$ at the abstract refined semantic $refdomaint$ (line 2). As described earlier, if an alarm ψ_n reported at the defect detection phase is not reported by the procedure CALLDOMAINSET (line 3) under the abstract refined semantic $refdomaint$ with an alarm ψ_m(line 5-6), we say ψ_m abstractly correlates with ψ_n and is denoted as $\psi_m \dashrightarrow \psi_n$.

5 Experimental results

To evaluate the effectiveness and efficiency, we have implemented a prototype of this novel technique as an extension to DTS, a static analysis tool developed by ourselves. DTS analyzes programs in four stages. First, source codes are preprocessed to an intermediate format by GCC compiler. Second, a compiler-front-end like analyzer generates the CFG and procedure call graph. Third, at the same time with a symbolic interval analysis for generating variants' abstract domain, we summarize each procedure and instantiate the callee summaries at concrete call sites. Finally, the $DFSM$ resolver detects potential defects along the control flow graph of the procedure, using the above dataflow information. In this section, we first present the experimental setup and show the results. Then, we analyze the results and give our experimental discussion.

D. Experimental Setup

We choose one type of common software warning null-pointer dereference and ten open source C projects with source code sizes ranging from several to hundreds of thousands of lines for evaluating our method, all of our experiments are done on a 2,4 GHz Intel Xeon Server with 3 GB main memory, running Windows Server 2003. We measured running time using enough repetitions to avoid timer resolution errors.

To quantify the efficiency and precision, the experiments are conducted in two different configurations.

The first configuration employs the original DTS, whereas the second leverages our alarm correlation method. In Tab. 1, the *LOC(S)* column indicates the total lines of source codes in all source files with suffixes ".c" and ".h", whereas the *LOC(I)* entries indicate the total lines in the preprocessed files, generated by GCC preprocessor with ".i" suffix. The *Original* and *New Method* entries respectively indicate the analyzing time consumption of original DTS and our alarm correlation method.

The alarms point (in column *Alarms*) indicates a potential defect, and the *Correlation* entries represent the number of correlated alarms. The intra-procedural correlated alarms are listed in column *Intra* and the inter-procedural correlated alarms are shown in Column *Inter*. The correlated alarms are grouped by their predominant alarm. The statistic entries with the form "%" stand for the investigation decrement ratio after manually distinguished. Finally, we give a statistics on the last line. For the two configurations, we are more interested in whether some of the unknown alarm correlations could be detected by our alarm correlation technique at a very little consuming time.

E. Experimentation Analysis and Discussion

As shown in TABLE 1, according to the statistics, the 1760 files contain over 0,65 million lines of source code (a parallel 5,4 million lines of intermediate code). All benchmarks were analyzed in 4,2 hours by our original DTS, and reported 2810 alarms, comparison to 4,57 hours by alarm correlation method with an average investigation decrement ratio $idr = 34.23\%$. Generally, the results of two configurations deliver on their promises: the *Original* was fast but the reported alarms are separate; the *New Method* reported 962 correlated alarms. Next, we will analyze the results of our alarm correlation method from the aspects of effectiveness and efficiency.

Table 1 Results of comparison experiment

Benchmark	Files	LOC(S)	LOC(I)	Time(sec)		Alarms	Correlation			Reduction
				Original	New Method		Intra	Inter	Total	%
antiword-0.37	78	20 213	126 925	121	159	54	5	0	5	9,26
barcode-0.98	18	3 409	19 591	34	44	17	1	0	1	5,88
spell-1.0	6	1 991	4 847	5	6	40	9	2	11	27,50
sphinxbase-0.3	120	22 517	157 332	196	224	285	37	12	49	17,19
uucp-1.07	252	52 595	518 067	1575	1626	237	48	0	48	20,25
sudo-1.8.6	191	56 866	207 123	360	399	103	12	2	14	13,59
acpid-1.0.8	6	1 680	13 491	12	12	0	0	0	0	0
ffmpeg-0.4.8	247	124 911	11 342	1717	1900	83	32	6	38	45,78
git-1.8.2	468	164 500	2 998 401	6205	6565	144	1	0	1	0,69
httpd-2.4.4	374	204 229	1 369 624	4902	5653	1847	725	70	795	43,04
Total:	1760	652 911	5 426 743	15127	16588	2810	870	92	962	34,23

Our alarm correlation algorithm turns out to be the most effective for ffmpeg-0.4.8 and httpd-2.4.4 (reduced by 45,78 % and 43,04 %) because of the following factors. First, these two projects both contain many dereferences of structure-typed variables. Second, it is common in these projects that valued dependencies of alarm-related variables propagate from callee to caller. For the example in figure 1, the structure-typed variable *sslconn* (line 359) is dereferenced at line 361 and 362, and its value is returned at the end of function *ssl_init_connection_ctx()*, which would further affect the callers. Our approach to resolve this problem is based on a unified procedure correlation summary model.

Suppose that a tool reports 3000 warnings and each warning requires 3 minutes for inspection, the time to inspect all the warnings will take 18,75 uninterrupted 8-h workdays [13]. Originally without the improvements by alarm correlation, the 2810 alarms will take about 18 workdays. Naturally, by applying our approach, there are 962 correlated alarms outputted by the proposed algorithm with only 23 minutes, that is, we save one-third of the total time for identifying all alarms. In summary, the experimental results demonstrate that our alarm correlation method scales well in practice, with an affordable efficiency.

6 Related work

In this paper, we provide a support in the alarm verification process to identify correlate relationships between alarms and propose a framework for the investigation of the alarms produced by DTS, so as to help classifying them by their correlations [13].

To help with diagnosis, Wei Le *et al.* [5] show that identifying a causal relationship among faults helps understand fault propagation and group faults of related causes. To our best knowledge, they are the first time to introduce the definition of fault correlation. By propagating the effects of the error state along the program path, they detect the correlation of pairs of alarms. However, Wei Le *et al.*'s method is not sound, their method is based on symbolic execution techniques, they introduce an external constraint solver to resolve integer constraints stored in the query. Our method is based on abstract interpretation theory and sound.

Rival proposed a method for the investigation of the alarms produced by their static analysis tool [6], so as to help classifying them as true errors or false alarms that are due to the approximation inherent in the static analysis. They refine an initial static analysis into an approximation of a subset of traces that actually lead to an error. If a combination of forward and backward refining analysis allows proving that this set is empty, they can conclude the alarm is false. In this paper, we do not check an alarm.

Indeed, we judge a correlation relationship between two alarms.

Our work resembles to Lee *et al.*'s work in the sense that both works refine the abstraction by exploiting the information about error state[14]. We are both trying to classify the alarms by their correlate relationship, but Lee *et al.*'s method is based on a super control flow graph. They cluster alarms based on a (not all) subset of possible dependencies, and if $\widehat{T}'_{\psi_n} \sqsubseteq \widehat{T}'_{\psi_m}$ at the program point$n$, they will abandon the abstract refined alarm trace semantic $\widehat{T}'_{\psi_m}$ and alarm information aboutψ_m, which will end up with missing more correlation information. On the other hand, we perform an intra-procedural alarm correlation computing among all alarms in one procedure. Also, we introduce a $head$ tag to record the alarm duplication in one procedure. If one alarm was reported many times, it will only be counted once. Our inter-procedural alarm correlation computing is based on an expanded unified symbolic procedure summary model. More details of our analysis can be found in our early works.

7 Conclusion

This paper addresses an important issue in the usability of static analysis tools: such tools tend to produce voluminous outputs and potential users are often dissuaded from using them because reviewing this output (warnings of different levels of severity) appears to be overwhelming. We present an approach that uses alarm correlation techniques to efficiently identify correlated warnings. Our approach is general enough to be applicable to any static analyzers based on abstract interpretation. The experimental results prove that our method has the effect of reducing 34,23 % of the amount of alarm identification. Using the correlation information, we are able to automate alarm identification that had to be done manually previously. In the future, we plan to optimize our alarm correlation techniques and reveal more complicated correlation situations.

Acknowledgment

This work is sponsored by the major research plan organized by the National Natural Science Foundation of China under Grant No. 91318301, the National Natural Science Foundation of China under Grant No.61202080, the National High-Tech Research and Development Plan of China under Grant 2012AA011201, the National Key Technology R&D Program under Grant 2012AA011201 and 2012BAH13F02.

8 References

[1] Das, M.; Lerner, S.; Seigle, M. Esp: Pathsensitive program verification in polynomial time. // in PLDI, 2002, pp. 57-68.

[2] Ayewah, N.; Pugh, W.; Morgenthaler, J. D.; Penix, J.; Zhou, Y. Using findbugs on production software. // in OOPSLA Companion, 2007, pp. 805-806.

[3] Bush, W. R.; Pincus, J. D.; Sielaff, D. J. A static analyzer for finding dynamic programming errors. // Software Practice and Experience. 30, 7(2000), pp. 775-802. DOI: 10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H

[4] Bertrane, J.; Cousot, P.; Cousot, R.; Feret, J.; Mauborgne, L.; Miné, A.; Rival, X. Static analysis by abstract interpretation of embedded critical software. // ACM Sigsoft Software Engineering Notes. 36, 1(2011), pp. 1-8. DOI: 10.1145/1921532.1921553

[5] Le, W.; Soffa, M. L. Path-based fault correlations. // in SIGSOFT FSE, 2010, pp. 307-316.

[6] Rival, X. Understanding the origin of alarms in astree. // in SAS, 2005, pp. 303-319.

[7] Dillig, I.; Dillig, T.; Aiken, A. Automated error diagnosis using abductive inference. // in PLDI, 2012, pp. 181-192.

[8] Zhao, Y.; Wang, Y.; Gong, Y.; Chen, H.; Xiao, Q.; Yang, Z. Stvl: Improve the precision of static defect detection with symbolic three-valued logic. // in APSEC, 2011, pp. 179-186.

[9] Wang, Y.; Gong, Y.; Chen, J.; Xiao, Q.; Yang, Z. An application of interval analysis in software static analysis. // in EUC (2), 2008, pp. 367-372.

[10] Zhao, Y.; Gong, Y.; Liu, L.; Xiao, Q.; Yang, Z. Context-sensitive interprocedural defect detection based on a unified symbolic procedure summary model. // in QSIC, 2011, pp. 51-60.

[11] Rival, X.; Mauborgne, L. The trace partitioning abstract domain. // ACM Trans. Program. Lang. Syst. 29, 5(2007). DOI: 10.1145/1275497.1275501

[12] Mauborgne, L.; Rival, X. Trace partitioning in abstract interpretation based static analyzers. // in ESOP, 2005, pp. 5-20.

[13] Heckman, S. S.; Williams, L. A. A systematic literature review of actionable alert identification techniques for automated static code analysis. // Information and Software Technology. 53, 4(2011), pp. 363-387. DOI: 10.1016/j.infsof.2010.12.007

[14] Lee, W.; Lee, W.; Yi, K. Sound non-statistical clustering of static analysis alarms. // in VMCAI, 2012, pp. 299-314.

[15] Xiao, S.; Pham, C. H. Performing high efficiency source code static analysis with intelligent extensions. // in APSEC, 2004, pp. 346-355.

[16] Williams, C. C.; Hollingsworth, J. K. Automatic mining of source code repositories to improve bug finding techniques. // IEEE Transactions on Software Engineering. 31, 6(2005), pp. 466-480. DOI: 10.1109/TSE.2005.63

[17] Heckman, S. S.; Williams, L. A. A model building process for identifying actionable static analysis alerts. // in ICST, 2009, pp. 161-170.

Authors' addresses

Dalin Zhang
National Research Center of Railway Safety Assessment,
Beijing Jiaotong University, Beijing, China
E-mail: dalin@bjtu.edu.cn

Dahai Jin
State Key Laboratory of Networking and Switching, Beijing
University of Posts and Telecommunications, Beijing, China

Yunzhan Gong
State Key Laboratory of Networking and Switching, Beijing
University of Posts and Telecommunications, Beijing, China

Siru Chen
History and Tourism Management, Inner Mongolia University,
Hohhot, China

Chengcheng Wang
Shanghai Stock Exchange, Shanghai, China