ROPAS
RESEARCH ON PROGRAM ANALYSIS SYSTEM
NATIONAL CREATIVE RESEARCH INITIATIVE CENTER 1998-2003
PROGRAMMING RESEARCH LABORATORY, SCHOOL OF COMPUTER SCIENCE & ENGINEERING
SEOUL NATIONAL UNIVERSITY

ROPAS MEMO
2005-24
April 26, 2005

Soundness by Static Analysis and False-alarm Removal by Statistical Analysis: Our Airac Experience*

Yungbum Jung, Jaehwang Kim, Jaeho Shin, Kwangkeun Yi
{dreameye,jaehwang,netj,kwang}@ropas.snu.ac.kr
Programming Research Laboratory
Seoul National University

April 26, 2005

Abstract

We present our experience of combining, in a realistic setting, a static analysis for soundness and a statistical analysis for false-alarm removal. The static analyzer is Airac that we have developed in the abstract interpretation framework for detecting buffer overruns in ANSI + GNU C programs. Airac is sound (finding all bugs) but with false alarms. Airac raised, for example, 970 buffer-overrun alarms in commercial C programs of 5.3 million lines and 233 among the 970 alarms were true. We addressed the false alarm problem by computing a probability of each alarm being true. We used Bayesian analysis and Monte Carlo method to estimate the probabilities and their credible sets. Depending on the user-provided ratio of the risk of silencing true alarms to that of false alarming, the system selectively present the analysis results (alarms) to the user. Though preliminary, the performance of the combination let us not hastefully trade the analysis soundness for a reduced number of false alarms.

1 Introduction

When one company's software quality assurance department started working with us to build a static analyzer that automatically detect buffer overruns[1] in their C softwares, they challenged us on three aspects: they hoped the analyzer 1) to be sound, detecting all possible buffer overruns; 2) to have a "reasonable" cost-accuracy balance; 3) not to assume a particular set of programming style about the C programs to analyze. Building a C buffer-overrun analyzer that satisfies all the three requirements was a big challenge. In the literature, we have seen impressive static analyzers, but their application targets allow them to drop one of the three aspects [6, 3, 9, 8].

In this article, we present our response that consists of two things: a sound analyzer named Airac and a statistical analysis engine on top of it. Airac collects all the true buffer-overrun points in C programs yet always with false alarms. The soundness is maintained, and the analysis accuracy is stretched to a point where the analysis cost remains acceptable. The statistical engine, given the analysis results (alarms), estimates the probability of each alarm being true. Only the alarms that have true-alarm probabilities higher than a threshold are reported to the user. The threshold is determined by the user-provided ratio of the risk of silencing true alarms to that of false alarming.

*An extended version of this paper will be submitted to a conference. This work was supported by Brain Korea 21 of Korea Ministry of Education and Human Resource Development, and by National Security

	Software	#Lines	Time (sec)	#Airac Alarms		#Real bugs
				#Buffers	#Accesses	
GNU S/W	tar-1.13	20,258	576.79s	24	66	1
	bison-1.875	25,907	809.35s	28	50	0
	sed-4.0.8	6,053	1154.32s	7	29	0
	gzip-1.2.4a	7,327	794.31s	9	17	0
	grep-2.5.1	9,297	603.58s	2	2	0
Linux kernel version 2.6.4	vmax302.c	246	0.28s	1	1	1
	xfrm_user.c	1,201	45.07s	2	2	1
	usb-midi.c	2,206	91.32s	2	10	4
	atkbd.c	811	1.99s	2	2	2
	keyboard.c	1,256	3.36s	2	2	1
	af_inet.c	1,273	1.17s	1	1	1
	eata_pio.c	984	7.50s	3	3	1
	cdc-acm.c	849	3.98s	1	3	3
	ip6_output.c	1,110	1.53s	0	0	0
	mptbase.c	6,158	0.79s	1	1	1
	aty128fb.c	2,466	0.32s	1	1	1
Commercial Softwares	software 1	109,878	4525.02s	16	64	1
	software 2	17,885	463.60s	8	18	9
	software 3	3,254	5.94s	17	57	0
	software 4	29,972	457.38s	10	140	112
	software 5	19,263	8912.86s	7	100	3
	software 6	36,731	43.65s	11	48	4
	software 7	138,305	38328.88s	34	147	47
	software 8	233,536	4285.13s	28	162	6
	software 9	47,268	2458.03s	25	273	1

Table 1: Analysis speed and accuracy of Airac

2 Airac, a Sound Analyzer

Automatically detecting buffer overruns in C programs is not trivial. Arbitrary expressions from simple arithmetics to values returned by function calls can be array indexes. Pointers pointing buffers can be aliased and they can be passed over as function parameters and returned from function calls. Buffers and pointers are equivalent in C. Contents of buffers themselves also can be used as indexes of arrays. Pointer arithmetic complicates the problem once more.

Airac's sound design is based on the abstract interpretation framework[4, 5]. To find out all possible buffer overruns in programs, Airac has to consider all states which can occur during programs executions. Airac computes sound approximation of program state at every program point and reports all possible buffer overruns by examining the approximate program states.

For a given program, Airac computes a map from flow edges to abstract machine states. The abstract machine state consists of abstract stack, abstract memory and abstract dump. Abstract stack, abstract memory and abstract dump are maps of which range domains consist of abstract value. We use interval domain $\widehat{\mathbb{Z}}$ for abstract numeric values. $[a, b] \in \widehat{\mathbb{Z}}$ represents an integer interval that has a as minimum and b as maximum. And this interval means a set of numeric values between a and b. To represent infinite interval, we use $-\infty$ and $+\infty$. $[-\infty, +\infty]$ means all integer values. The abstract array is a triple which consists of its initial address,

Research Institute of Korea Ministry of Information and Communication.

[1]Buffer overruns happen when an index value is out of the target buffer size. They are common bugs in C programs and are main sources of security vulnerability. From 1/2[2] to 2/3[1] of security holes are due to buffer overruns.

offset interval and size interval. We use allocation sites to denote abstract memory locations. An integer array which is allocated at l and has size s is represented as $\langle l, [0,0], [s,s]\rangle$.

2.1 Striking a Cost-Accuracy Balance

Airac has many features designed to decrease false alarms or to speed-up analysis and all techniques don't violate the analysis soundness.

2.1.1 Accuracy Improvement

We use the following techniques to improve the analysis accuracy of Airac:

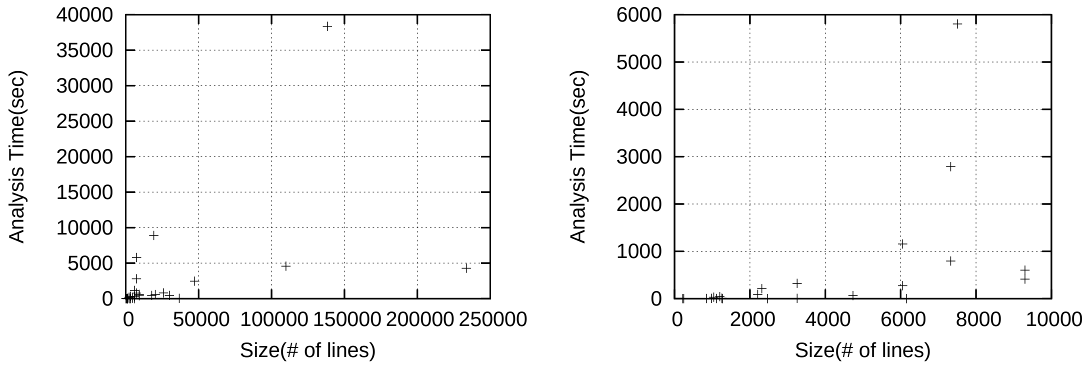
- **Unique Renaming** Memory locations are abstracted by allocation sites. In Airac, sites of variable declarations are represented by variable name and other sites are assigned unique labels. So to prevent interferences among variables, Airac renames all variables to have unique names.
- **Narrowing After Widening** The height of integer interval domain is infinite. Widening operator[4] is essential for the analysis termination. But this operator decreases accuracy of analysis result. Narrowing is used for recovering accuracy.
- **Flow Sensitive Analysis** Destructive assignment is always allowed except for within cyclic flow graphs.
- **Context Pruning** We can confine interval values using conditional expressions of branch statements. Airac use these information to prune interval values and this pruning improve analysis accuracy.
- **Polyvariant Analysis** Function-inlining effect by labeling function-body expressions uniquely to each call-site: the number of different labels for an expression is bound by a value from user. This method is weakened within recursive call cycles.
- **Static Loop Unrolling** Loop-unrolling effect by labeling loop-body expressions uniquely to each iteration: the number of different labels for an expression is bound by a value from the user.

2.1.2 Cost Reduction

When the fixpoint iteration reaches the junction points, we have to check the partial orders of abstract machines and we also commit the join($\sqcup$) operations. These tasks take most of analysis time. The speed of the analysis highly depends on how we handle such operations efficiently.

We developed techniques to reduce time required for partial order checking and join operation.

- **Stack Obviation** We transform the original programs whose effects on stack are reflected by the memory. And this transformation makes Airac avoid scanning abstract stacks during ordering abstract machines.
- **Selective Memory Join** Airac keeps track of information that indicates changed entries in abstract memory. Join operation is applied only to those changed values.
- **Wait-at-Join** For program points where many data flows join, Airac delays the computation for edges starting from the current point until all computations for the incoming edges are done.



(a) (b)

Figure 1: Airac's scalability

3 Performance of Airac

This section presents Airac's performance. Numbers that are before the statistical engine sift out alarms that are probably false.

Airac is implemented in nML[2] and tested to analyze GNU softwares, Linux kernel sources and commercial softwares. The commercial softwares are all embedded softwares. Airac found some fatal bugs in these softwares which were under development. Table 1 shows the result of our experiment. "#Lines" is the number of lines of the C source files before preprocessing them. "Time" is the user CPU time in seconds. "#Buffers" is the number of buffers those may be overrun. "#Accesses" is the number of buffer-access expressions that may overrun. "#Real Bugs" is the number of buffer accesses that are confirmed to be able to cause real overruns. Two graphs in Figure 1 show Airac's scalability behavior. X axis is the size (number of lines) of the input program to analyze and Y axis is the analysis time in seconds. (b) is a microscopic view of (a)'s lower left corner. Experiment was done in a Linux system with a Pentium4 3.2GHz CPU and 4GB of RAM.

We found some examples in real codes that Airac's accuracy and soundness shines:

- In GNU S/W tar-1.13 program rmt.c source, Airac detected the overrun point inside the `get_string` function to which a buffer pointer is passed:

```
static void
get_string (char *string)
{
  int counter;

  for (counter = 0;
       counter < STRING_SIZE;
       counter++) {
    .....
  }
  string[counter] = '\0';
      // counter == STRING_SIZE
}


int
```

[2]Korean dialect of ML programming language. `http://ropas.snu.ac.kr/n`

```
main (int argc, char *const *argv)
{
  char device_string[STRING_SIZE];
  ......
  get_string(device_string);
  ......
}
```

- Airac catched errors in the following simple cases, for which syntactic pattern matching or unsound analyzer are likely to fail to detect.
  - Function pointer is used for calculating an index value:

```
int incr(int i) { return i+1;}
int decr(int i) { return i-1;}

main() {
  int (*farr[]) (int) = {decr, decr, incr};
  int idx = rand()%3;
  int arr[10];
  int num = farr[idx](10);
  arr[num] = 10;         //index:[9, 11]
}
```

  - Index variable is increased in an infinite loop:

```
main() {
  int arr[10];
  int i = 0;
  while(1){
    *(arr + i) = 10;    //index:[0, +Inf]
    i++;
  }
}
```

  - Index variable is passed to a function by parameter and updated in the function:

```
simpleCal(int idx) {
  int arr[10];
  idx += 5;
  idx += 10;
  arr[idx];             //index:[17, 17]
}
main() {
  simpleCal(2);
}
```

4 Sifting-out False Alarms By Statistical Post Analysis

We use Bayesian approach [7] to compute the probability of alarms being true. Let $\oplus$ denote the event an alarm raised is true and $\ominus$ the event an alarm is false. S_i denotes a single symptom is observed in the raised alarm and $\vec{S}$ is a vector of such symptoms. $P(E)$ denotes the probability of an event E, and $P(A \mid B)$ is the conditional probability of A given B. Bayes' rule is used to predict the probability of a new event from prior knowledge. In our case, we accumulate the number of true and false alarms having each specific symptom from alarms already verified and classified to be true or false by humans. From this knowledge we compute the probability of a new alarm with some symptoms being a true one.

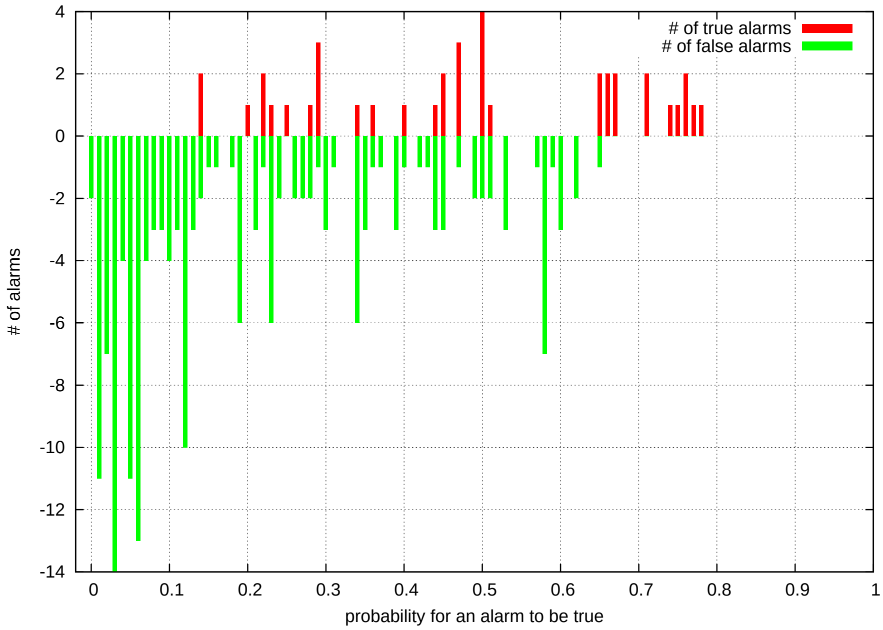


Figure 2: Probability of alarms being true. False alarms are counted in negative numbers. 52% of false alarms have probabilities under 0.14.

To compute the Bayesian probability, we need to define symptoms featuring alarms and gather them from already analyzed programs and classified alarms. We defined symptoms both syntactically and semantically. Syntactic symptoms describe the syntactic context before the alarmed expressions. The syntactic context consists of program constructs used before the alarmed expressions. Semantic symptoms are gathered during Airac's fixpoint computation phase. For such symptoms, we defined symptoms representing whether context pruning was applied, whether narrowing was applied, whether an interval value has infinity and so forth.

From the Bayes' theorem, probability $P(\oplus \mid \vec{S})$ of an alarm being true that has symptoms $\vec{S}$ can be computed as the following:

$$P(\oplus \mid \vec{S}) = \frac{P(\vec{S} \mid \oplus)P(\oplus)}{P(\vec{S})} = \frac{P(\vec{S} \mid \oplus)P(\oplus)}{P(\vec{S} \mid \oplus)P(\oplus) + P(\vec{S} \mid \ominus)P(\ominus)}.$$

By assuming each symptom in $\vec{S}$ occurs independently under each class, we have

$$P(\vec{S} \mid c) = \prod_{S_i \in \vec{S}} P(S_i \mid c) \text{ where } c \in \{\oplus, \ominus\}.$$

Here, $P(S_i \mid c)$ is estimated by Bayesian analysis from our empirical data. We assume prior distributions are uniform on $[0,1]$. Let p be the estimator of the probability $P(\oplus)$ of an alarm being true. $P(S_i \mid \oplus)$ and $P(S_i \mid \ominus)$ are estimated by θ_i and η_i respectively. Assuming that each S_i are independent in each class, the posterior distribution of $P(\oplus \mid \vec{S})$ taking our empirical data into account is established as following:

$$\hat{\psi}_j = \frac{(\prod_{S_i \in \vec{S}} \theta_i) \cdot p}{(\prod_{S_i \in \vec{S}} \theta_i) \cdot p + (\prod_{S_i \in \vec{S}} \eta_i) \cdot (1-p)}$$

where p, θ_i and η_i have beta distributions as

$$\begin{array}{rcl} p & \sim & Beta(N(\oplus)+1, n-N(\oplus)+1) \\ \theta_i & \sim & Beta(N(\oplus,\ S_i)+1, N(\oplus, \neg S_i)+1) \\ \eta_i & \sim & Beta(N(\ominus, S_i)+1, N(\ominus, \neg S_i)+1) \end{array}$$

and $N(E)$ is the number of events E counted from our empirical data.

Now the estimation of p, θ_i,η_i are done by Monte Carlo method. We randomly generate $p_i, \theta_{ij}, \eta_{ij}$ values N times from the beta distributions and compute N instances of ψ_j. We take the mean $\bar{\psi_j}$ for $\hat{\psi}$. Then the $100(1-2\alpha)\%$ credible set of $\hat{\psi}$ is $(\psi_{j_{\alpha \cdot N}}, \psi_{j_{(1-\alpha)\cdot N}})$ where $\psi_{j_1} < \psi_{j_2} < \cdots < \psi_{j_N}$. After obtaining the probability $\hat{\psi}$ for each alarm to be true, we have to decide whether we should report the alarm or not. To choose a reasonable threshold, the user supplies two parameters defining the magnitude of risk: a_1 for not reporting true alarms and a_2 for reporting false alarms.

	$\oplus$	$\ominus$
risk of reporting	0	a_2
risk of not reporting	a_1	0

Since the probability of an alarm being true error is $\hat{\psi}$, the expectation value of risk when we raise an alarm is $a_2 \cdot (1-\hat{\psi})$, and $a_1 \cdot \hat{\psi}$ when we don't raise. To minimize the risk, we must choose the smaller side. Hence, the threshold of probability for reporting can be chosen as:

$$a_1 \cdot \hat{\psi} > a_2 \cdot (1-\hat{\psi}) \iff \hat{\psi} > \frac{a_2}{a_1+a_2}.$$

For example, user can supply $a_1 = 9, a_2 = 1$ if he or she believes that not alarming for true errors have risk 9 times greater than raising false alarms. Then the threshold for the probability being true to report becomes $1/10 = 0.1$ and whenever the probability of an alarm is greater than 0.1, we should report it. For a sound analysis, to miss a true alarm is considered much riskier than to report a false alarm, so it is recommended to choose the two risk values $a_1 \gg a_2$ to keep more soundness.

We have done some experiments with our samples of programs and alarms. Samples were first divided into learning set and testing set. 50% of the alarms were randomly selected and their symptoms were counted based on their classes. With these precalculated numbers, $\hat{\psi}$ for each remaining alarm was computed by taking the mean of 2000 ψ_j's which was computed from p and each θ_i and η_i of its symptoms, all randomly generated. We can view alarms in the testing set as new alarms, since their symptoms didn't contribute to the numbers used for the calculation of $\hat{\psi}$.

The histogram in Figure 2 was constructed from the data of 3 runs of the experiment previously described. Dark bars indicate true alarms and white ones are false. Although probability of true alarms range from 0.14 to 0.78, 52% (=100 * 92/(92+82)) of false alarms have probability less than 0.14. If we had assumed the risk of missing true error is about 6 times greater than false alarming, then we could choose 0.143 as a threshold. Using this threshold, more than half of false alarms can be filtered, or deferred. We believe we will be able to distinguish true and false alarms even better than we do currently, if we extract better symptoms coupled with the weak points of Airac.

5 Conclusion

Our Airac experience encourages us to conclude that it is not inevitable to trade the soundness for a reduced number of false alarms. By striking a cost-accuracy balance of a sound analyzer, we can first achieve an analyzer that is itself useful with small false-alarm rate in most cases (as the experiment numbers showed for analyzing Linux kernels). Then, by a careful design of a Bayesian analysis of the analyzer's false-alarm behaviors, we can achieve a post-processing engine that sifts out false alarms from the analysis results. For the Bayesian analysis engine to be effective the analyzer designer must be able to pin-point the exact symptoms for false alarms. This ability comes from a deep understanding of the analyzer's weaknesses.

Though the Bayesian analysis phase still has the risk of sifting out true alarms, it can reduce the risk at the user's desire. Given the user-provided ratio of the risk of silencing true alarms to that of false alarming, a simple decision theory determines the threshold probability that an alarm with a lower probability is silenced as a false one. Because the underlying analyzer is sound, if the user is willing to, (s)he can receive a report that contain all the real alarms. For Airac, when the risk of missing true alarms is six times greater than that of false alarming, all the real alarms are reported with the half of false alarms sifted out.

Acknowledgements We thank Jaeyong Lee for the design of the statistical analysis. We thank Hakjoo Oh and Yikwon Hwang for helping our experiment by selecting sample programs and manually classifying alarms.

References

[1] bugtraq. `www.securityfocus.com`.

[2] CERT/CC advisories. `www.cert.org/advisories`.

[3] Bruno Blanchet, Patric Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antonie Mine, David Monnizux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, June 2003.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[5] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[6] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167. ACM Press, 2003.

[7] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Textx in Statistical Science. Chapman & Hall/CRC, second edition edition, 2004.

[8] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[9] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM Press, 2003.