Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis [⋆]

Yungbum Jung, Jaehwang Kim, Jaeho Shin and Kwangkeun Yi

Programming Research Laboratory
School of Computer Science and Engineering
Seoul National University
{dreameye,jaehwang,netj,kwang}@ropas.snu.ac.kr

Abstract. We present our experience of combining, in a realistic setting, a static analyzer with a statistical analysis. This combination is in order to reduce the inevitable false alarms from a domain-unaware static analyzer. Our analyzer named Airac(Array Index Range Analyzer for C) collects all the true buffer-overrun points in ANSI C programs. The soundness is maintained, and the analysis' cost-accuracy improvement is achieved by techniques that static analysis community has long accumulated. For still inevitable false alarms (e.g. Airac raised 970 buffer-overrun alarms in commercial C programs of 5.3 million lines and 737 among the 970 alarms were false), which are always apt for particular C programs, we use a statistical post analysis. The statistical analysis, given the analysis results (alarms), sifts out probable false alarms and prioritizes true alarms. It estimates the probability of each alarm being true. The probabilities are used in two ways: 1) only the alarms that have true-alarm probabilities higher than a threshold are reported to the user; 2) the alarms are sorted by the probability before reporting, so that the user can check highly probable errors first. In our experiments with Linux kernel sources, if we set the risk of missing true error is about 3 times greater than false alarming, 74.83% of false alarms could be filtered; only 15.17% of false alarms were mixed up until the user observes 50% of the true 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

[⋆] This work was supported by Brain Korea 21 Project of Korea Ministry of Education and Human Resources, by IT Leading R&D Support Project of Korea Ministry of Information and Communication, by Korea Research Foundation grant KRF-2003-041-D00528, and by National Security Research Institute.

[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.

their C softwares, they challenged us with three goals: they hoped the analyzer 1) to be sound, detecting all possible buffer overruns; 2) to have a reasonable cost-accuracy balance; and 3) not to assume a particular set of programming style about the input C programs because they handle a wide spectrum of C softwares to be embedded in various electronic devices. Building a realistic C buffer-overrun analyzer that satisfies all the three requirements was a hard challenge. In the literature, we have seen impressive static analyzers yet their application targets seem to allow them to drop one of the three requirements [7, 4, 13, 9]. The major challenge is how to reduce the number of inevitable false alarms from a realistic, sound static analyzer that cannot assume a particular style for the input C programs.

In respond to the challenge, we decided to try the following path: design a sound static analysis whose accuracy is stretched to a point where the analysis cost remains acceptable, then use a statistical post analysis in order to sift out alarms that are probable to be false. The analyzer named Airac(Array Index Range Analyzer for C) collects all the true buffer-overrun points in ANSI C programs. The soundness is maintained, and the analysis' cost-accuracy balance is stroke with techniques that static analysis community has long accumulated. Now for still inevitable false alarms, which are always apt for particular C programs, we use a statistical post analysis. The statistical analysis, given the analysis results (alarms), sifts out some alarms that are probable to be false. It estimates the probability of each alarm being true. The probabilities are used in two ways: 1) 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. 2) By sorting the alarms to be reported in descending order, it allows the user to examine highly probable alarms first.

Airac targets the full set of ANSI C constructs as indexing expressions: from simple arithmetics to arbitrary expressions involving function calls, pointer arithmetics, and aliases. Airac handles buffers that are dynamically allocated consecutive memory cells of dynamic lengths as well as static arrays. "*buffer overrun*" happens when an index value denotes an address outside the target buffer area. Striking a cost-accuracy balance of Airac is done by the following techniques: for accuracy improvement we use narrowing after widening, flow-sensitivity, polyvariance, context pruning (an instance of trace partitioning[11]) and static loop unrolling. For cost reduction, we used stack obviation (removal of the stack from our abstract state), selective memory join (point-wise join for abstract memory is applied only to the changed entries), and wait-at-join (a work-list iteration does not continue to pass a join point until all threads arrive). For commercial C programs of 5.3 million LOC, Airac raises 970 buffer-overrun alarms, among which 233 alarms are true. For some parts of the Linux kernel of 18,760 LOC, Airac raises 26 alarms, among which 16 are true.

The statistical method aiming to sift out false alarms is designed by the Bayesian data analysis framework[8], implemented by the Monte Carlo method[12], and parameterized by a simple decision theory[3]. We define a conditional probability formula for an alarm to be true given the set of symptoms observed for the alarm. This probability formula has parameter probabilities, whose distributions are determined by Bayesian analysis from the "training set" (or "past experience knowledge"). The parameter probabilities are obtained by the Monte Carlo method. The training set, which consists of alarms and their conditional probabilities of having symptoms given that they are either true or false, is obtained by running our analyzer for a set of Linux kernel, and textbook C programs and manually classifying the alarms into either true or false. Having computed the probability of each alarm being true, we report only the alarms that have the true-alarm probabilities higher than a threshold. The threshold is determined by the user-provided ratio of the risk of silencing true alarms to that of raising false alarms. The ratio, for each alarm, determines the expected risks of silencing it or alarming it. The action with a smaller risk is chosen. This statistical engine's effectiveness is promising. If the user set the risk of missing true error is 3 times greater than false alarming, then 74.83% of false alarms could be sifted out. Meanwhile, by ranking the alarms by higher probabilities and examining from the top, the user encounters only 15.17% of false alarms until he or she reaches 50% of the true ones.

2 Airac, the Analyzer

Airac is an abstract interpreter. To find out all possible buffer overruns in programs, Airac considers all states which may occur during programs executions. It computes a sound approximation of dynamic program states occurring at each program point and reports possible buffer overruns by examining the approximate states.

A concrete array block is abstracted as a triple that consists of abstract base address, offset, and size. Abstract base address is one for each memory allocation site in C programs. Abstract offset and size are integer intervals. For example, for the following C code:

```
int p[5];
int *q = p + 3;
*(q+3) = 1;
```

The pointer p's abstract value is $\langle l, [0,0], [5,5] \rangle$ where name l is the abstract base address for the declared array. $[0,0]$ and $[5,5]$ are respectively the current offset and size as intervals. After the pointer arithmetic, q is initialized as $\langle l, [3,3], [5,5] \rangle$; then the value of q+3 is $\langle l, [6,6], [5,5] \rangle$ whose offset exceeds its size, where our analyzer raises a buffer overrun alarm.

2.1 Semantics and Its Abstraction

C program's collecting semantics is defined as the set of transition sequences of machine states. A machine state is a tuple of a program point, data stack, environment, memory, and control stack (dump). A C program's semantics is the least fixed point of the following function that transforms the machine transition traces:

$$\begin{aligned}&\mathcal{F} : 2^{\mathit{Machine}^{\omega}} \rightarrow 2^{\mathit{Machine}^{\omega}}\\&\mathcal{F}(X) = \{m_0\} \cup \{t \hookrightarrow m_{n+1} \mid t \overset{\text{let}}{=} \cdots \hookrightarrow m_n \in X,\ m_n \hookrightarrow m_{n+1}\}\end{aligned}$$

where $\mathit{Machine} = \mathit{Edge} \times \mathit{State}$, the program points $\mathit{Edge} = \mathit{Lab} \times \mathit{Lab}$ are the set of edges between two program labels, and m_0 is the initial machine state. Labels are uniquely assigned to all the expressions and commands of the input C program.

We approximate the collecting semantics by $T \in \mathit{Edge} \rightarrow \widehat{\mathit{State}}$ that maps each program point to an abstract state

$$\widehat{\mathit{State}} = \widehat{\mathit{Stack}} \times \widehat{\mathit{Mem}} \times \widehat{\mathit{Dump}}$$

The abstract state at each program point approximates all the states occurring at the point in all the concrete transition sequences. The map is defined as the least fixed point of the following function:

$$\begin{aligned}&\widehat{\mathcal{F}} : (\mathit{Edge} \rightarrow \widehat{\mathit{State}}) \rightarrow (\mathit{Edge} \rightarrow \widehat{\mathit{State}})\\&\widehat{\mathcal{F}}(T) = \lambda\langle l, l'\rangle.s \ \text{ where } \langle l,\ \bigsqcup\{s' \mid p \in \mathit{pred}(l), T\langle p, l\rangle = s'\}\rangle \hookrightarrow^{\#} \langle l', s\rangle\end{aligned}$$

where $\mathit{pred}(l)$ is the set of predecessors of label l in the transition sequences and $\hookrightarrow^{\#}$ is an abstraction of the concrete transition relation $\hookrightarrow$.

2.2 Fixpoint Algorithm

The fixpoint algorithm is a chaotic working set algorithm. The working set consists of labels of expressions whose abstract state has to be re-computed. When a computed machine state for $T\langle l, l'\rangle$ is changed, we add l' to the working set, to re-compute the states of the edges from l'. The working set is a stack, hence each abstract transition step follows the program's execution flow in a depth-first order of the flow graph. When the next program points to evaluate are multiple (as when we compute conditional expressions), those two points are grouped together and pushed as a single unit to the working set stack. This grouping adds a flavor of breadth-first traversal of the flow graph. The working set algorithm selectively applies the widening and narrowing operations at the heads of flow cycles.

2.3 Accuracy Improvement

We use some techniques to improve the analysis accuracy: 1) we use widening and narrowing for interval domain[5]; 2) we use destructive assignment to achieve flow sensitive analysis except for within cyclic call chains; 3) we use context pruning to confine interval values; 4) we use function-inlining for polyvariant analysis; 5) we use static loop unrolling. Though each technique is independent of others, using all the techniques in combination results in a synergy for improving the analysis accuracy.

Table 1. Experiment result of cost reduction techniques

Version	Time[a](s)	Speed-Up	Time[b](s)	Speed-Up
none	18317.55	0%	16253.18	0%
selective join	16055.58	12.35%	14286.72	12.1%
wait-at-join	19317.67	-5.45%	13153.43	19.98%
stack obviation	3717.06	79.71%	3247.79	81.02%
all	3461.57	81.11%	2320.58	85.73%

[a] the sum of analysis time for 43 Linux kernel programs.

[b] same as *a* except one program that *wait-at-join* has bad influence upon work list algorithm.

2.4 Cost Reduction

We used three techniques for cost reduction of Airac. They are *stack obviation*, *selective join* and *wait-at-join*. From experiment results on parts of the Linux kernel, we could observe that *stack obviation* is a very powerful technique for cost reduction. The *wait-at-join* technique works well for most programs with some exceptions.

Stack Obviation Comparing($\sqsubseteq$) and joining($\sqcup$) abstract states, which take most of the analysis time, involves applying the operations to the abstract stack component. If the abstract stack component is always reflected in other components of the machine states then we can skip applying the operations to the stack component.

Before analysis begins, Airac transforms the input program to have all stack variations of each transition be reflected on the memory. For example, conditional expression

```
(x > 0) ? 1 : 2
```

is transformed to

```
{ var tmp;  if (x > 0) tmp = 1; else tmp = 2;  tmp; }
```

Note that the original expression's branches are to push different values to the stack component. Hence estimating the final value must join the stack components from the branches. On the other hand, for the transformed expression, we don't have to consider joining the stack components because the state difference of the two branches are to be reflected by the memory component because of the assignments to the temporary variable `tmp`. This transformation costs only one more location in the abstract memory, while it avoids scanning the abstract stack component. This technique reduced our analysis time by 79.71%.

Selective Memory Join Airac keeps track of information that indicates changed entries in abstract memory. Join operation is applied only to those changed values. Comparing with pre-state, Airac reduces size of the information by removing unchanged entries. This technique, which was also mentioned in [4], reduced our analysis time by 12.35%.

Wait-at-Join For program points where many control-flows join, Airac delays the computation for current point until all computations for the incoming edges are done. By this, Airac can reduce redundant computations after the junction point. However, it is very costly to decide whether all threads have reached current point or not. So Airac chooses a simple strategy which waits until the working stack becomes empty. This technique is very effective for C programs that have a many junction points, e.g., large switch statements. This technique reduced our analysis time by 19.98% for most programs.

2.5 Airac's Cost, Accuracy and Scalability

We implemented Airac using nML[2] and analyzed various softwares from toy C programs to serious ones such as GNU softwares, Linux kernel sources and commercial softwares. All these commercial softwares are embedded softwares[3]. Airac found some fatal bugs in these softwares which were under development. Table 2.5 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 that 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 on a Linux system with a single Pentium4 3.2GHz CPU and 4GB of RAM.

Airac is scalable enough to analyze real world softwares. Airac can analyze programs of up to about 10,000 lines at once. GNU softwares such as grep, gzip and sed were analyzed as a whole. And these analyses took less than an hour to finish.

3 Statistical Taming of False Alarms

Reducing the number of false alarms is the key issue in sound analyses. Sound analyzers that cannot assume a particular style for the input programs can often have a high false-alarm rate. Controlling the abstraction level of the analysis will work but not very effectively. It is clear that by using less abstract domains we can distinguish more concrete values, but practically, relying solely on this approach will soon hit an unacceptable cost. Furthermore, if the analyzer must handle unlimited set of input programs, there can always be some programs that

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

[3] Their real names cannot be disclosed due to the contract.

Table 2. Analysis speed and accuracy of Airac

	Software	#Lines	Time	#Airac Alarms		#Real
			(sec)	#Buffers	#Accesses	bugs
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

fool the analyzer. User annotation in source codes can be a powerful method [7] yet is always less desirable than being fully automatic. And, that the analyzer blindly repair its accuracy based on the annotations makes the approach vulnerable to annotation bugs. Heuristics can be applied to classify alarms into true and false ones[10]. However, unless heuristics have a strong basis, we can hardly be confident with their classifications. Heuristics can be used even during the analysis itself and may tempt us to give up the soundness and claim that such sacrifice is inevitable in order to increase the analysis precision. But, if possible with a comparable price, it is always better to know all possible bugs than knowing only some of them.

Without giving up its soundness, Airac handles the inevitable false alarms by using statistical post analysis built on top of a firm theoretical basis. The statistical analysis, 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. Though the statistical analysis phase still has the risk of sifting out true alarms, it can reduce the risk at the

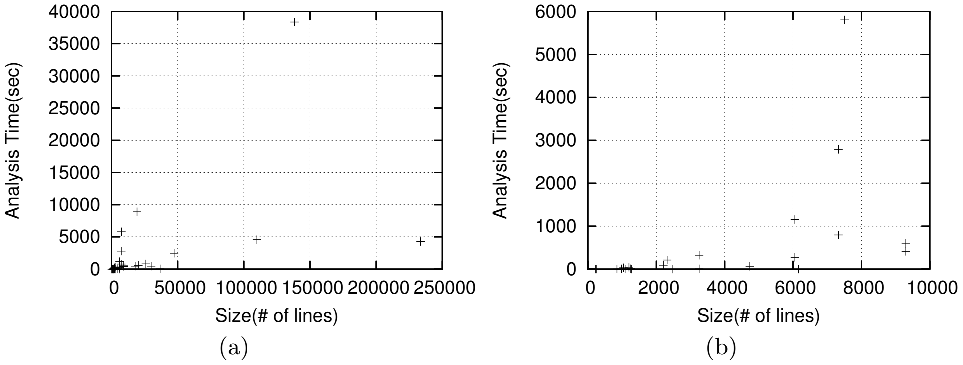

Fig. 1. Airac's scalability

user's desire. Because the underlying analyzer is sound, if the user is willing to, (s)he can receive a report that contain all the real alarms.

3.1 Bayesian Analysis

We use Bayesian statistics[8] to compute the probability of an alarm being true. Let $\oplus$ denote the event an alarm raised is true, and let $\ominus$ denote it is false. S_i denotes that a single symptom is observed in the raised alarm and $\boldsymbol{S}$ is a vector of such symptoms. The set of symptoms that we used for Airac will be presented in 3.4. $P(E)$ denotes the probability of an event E, and $P(A \mid B)$ is the conditional probability of A given B. We call the probability $P(\oplus \mid \boldsymbol{S})$ of an alarm being true given its symptoms as *the trueness of the alarm.*

Bayes' theorem is used to predict the probability of a new event from prior knowledge. To set up such knowledge base we classify alarms into true and false manually and count occurrences of each symptom in true and false alarms respectively. From this knowledge we are able to compute the trueness of new alarms using their symptoms. Using Bayes' theorem, the trueness $P(\oplus \mid \boldsymbol{S})$ can be computed as the following:

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

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

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

Here, $P(S_i \mid \oplus)$ is estimated by $\hat{\psi}$ using Bayesian analysis of our empirical data. We assume prior distributions are uniform on $[0, 1]$. Let p be the estimator of the ratio $P(\oplus)$ of true alarms to all raised alarms. Each $P(S_i \mid \oplus)$ and $P(S_i \mid$

$\ominus$) is estimated by θ_i and η_i respectively. Assuming that each S_i are independent in each class, the posterior distribution of $P(\oplus \mid \boldsymbol{S})$ taking our empirical data into account is established as following:

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

where p, θ_i and η_i have beta distributions as

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

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. Then the $100(1-2\alpha)\%$ credible set of $\hat{\psi}$ is $(\psi_{j_{\alpha \cdot N}}, \psi_{j_{(1-\alpha)\cdot N}})$ where $\psi_{j_1} < \psi_{j_2} < \cdots < \psi_{j_N}$. We take the upper bound $\psi_{j_{(1-\alpha)\cdot N}}$ for $\hat{\psi}$, since the maximal probability being true is our concern as seen later.

3.2 Sifting Out False Alarms

We can use the estimated trueness for sifting out false alarms systematically. To decide whether we should sift out an alarm or not, we need a threshold to compare with the estimated $\hat{\psi}$ with $100(1-2\alpha)\%$ credibility. To choose a reasonable threshold, the user supplies two parameters defining the magnitude of risk: r_m for missing true alarms and r_f for reporting false alarms. Only their ratio, not their absolute values matter.

	$\oplus$	$\ominus$
risk of reporting	0	r_f
risk of not reporting	r_m	0

Given an alarm whose trueness is ψ, the expectation of risk when we raise an alarm is $r_f \cdot (1-\psi)$, and $r_m \cdot \psi$ when we don't. To minimize the risk, we must choose the smaller side. Hence, the threshold of trueness to report the alarm can be chosen as:

$$r_m \cdot \psi \geq r_f \cdot (1-\psi) \quad \iff \quad \psi \geq \frac{r_f}{r_m + r_f}.$$

If the trueness of an alarm can be greater than or equal to such threshold, i.e. if the upper bound of trueness $\hat{\psi}$ is greater than such threshold, then the alarm should be raised with $100(1-2\alpha)\%$ credibility. For example, the user can supply $r_m = 3, r_f = 1$ if he or she believes that not alarming for true errors have risk 3 times greater than raising false alarms. Then the threshold for the probability being true to report becomes $1/4 = 0.25$ and whenever the estimated trueness of an alarm is greater than 0.25, we should report it.

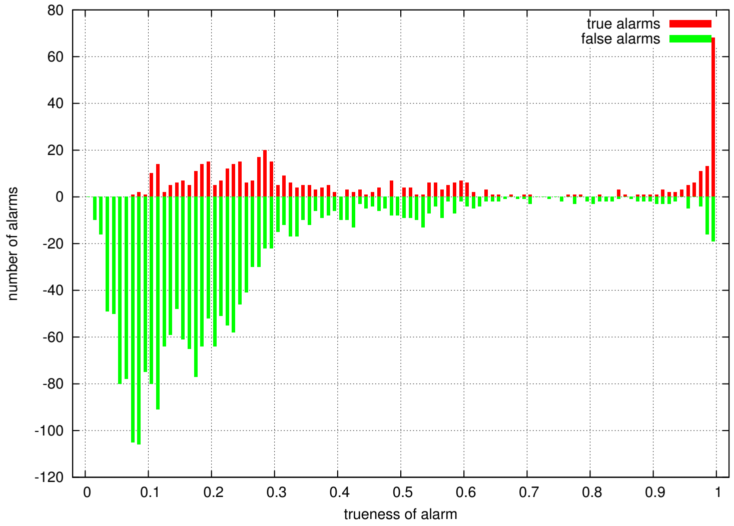

Fig. 2. Frequency of trueness in true and false alarms. False alarms are counted in negative numbers. 74.83% of false alarms have trueness less than 0.25.

We have done some experiments with our samples of programs and alarms. Some parts of the Linux kernel and programs that demonstrate classical algorithms were used for the experiment. For a single experiment, samples were first divided into learning set and testing set. 50% of the alarms were randomly selected as learning set, and the others for testing set. Each symptom in the learning set were counted according to whether the alarm was true or false. With these pre-calculated numbers, $\hat{\psi}$ for each alarm in the testing set was estimated using the 90% credible set constructed by Monte Carlo method. Using Equation (1), we computed 2000 ψ_j's from 2000 p's and θ_i's and η_i's, all randomly generated from their distributions. We can view alarms in the testing set as alarms from new programs, since their symptoms didn't contribute to the numbers used for the estimation of $\hat{\psi}$.

The histogram in Figure 2 was constructed from the data generated by repeating the experiment 15 times. Dark bars indicate true alarms and lighter ones are false. 74.83% ($\approx$ 1504/2010) of false alarms have trueness less than 0.25, so that they can be sifted out. For users who consider the risk of missing true error is 3 times greater than false alarming, almost three quarters of false alarms could be sifted out, or preferably just deferred.

For a sound analysis, it is considered much riskier to miss a true alarm than to report a false one, so it is recommended to choose the two risk values $r_m \gg r_f$ to keep more soundness. For the experiment result Figure 2 presents, 31.40% ($\approx$ 146/465) of true alarms had trueness less than or equal to 0.25, and were also sifted out with false alarms. Although we do not miss any true alarm by lowering the threshold down to 0.07 ($r_m/r_f \approx 13$) for this case, it does not guarantee any

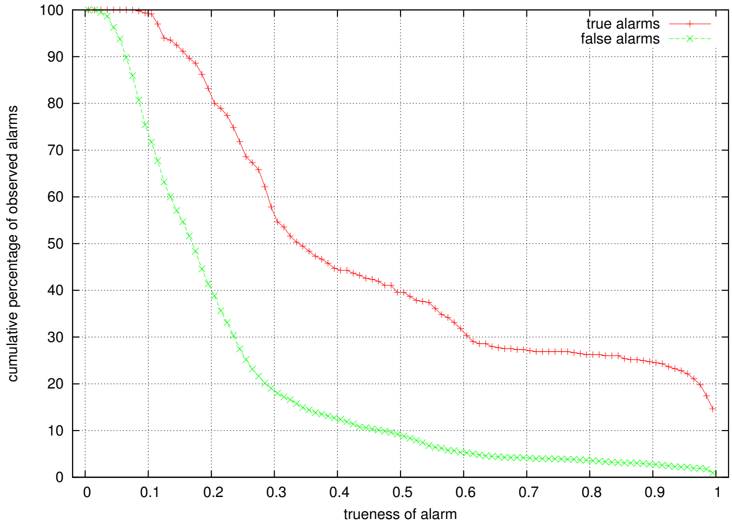

Fig. 3. Cumulative percentage of observed alarms starting from trueness 1 and down. Only 15.17% of false alarms get mixed up until 50% of the trues are observed.

kind of soundness in general. However, to obtain a sound analysis result, one can always set $r_f = 0$, i.e. allowing none of the alarms to be sifted out.

3.3 Ranking False Alarms

We can rank alarms by their trueness to give effective results to the user. This ranking can be used both with and without the previous sifting-out technique. By ordering alarms, we let the user handle more probable errors first. Although the trueness of true alarms are scattered over 0 through 1, we can see that most of the false alarms have small trueness. Hence, sorting by trueness and showing in decreasing order will effectively give true alarms first to the user. Figure 3 shows the cumulative percentage of observed alarms starting from trueness 1 and down using the same data in Figure 2. When the user inspects alarms having high trueness first, only 15.17% (=305/2010) of false alarms gets mixed up until 50% of the true alarms are observed, where the trueness equals 0.3357.

3.4 Symptoms

We use both syntactic and semantic symptoms that may influence the analysis accuracy. The symptoms can be classified into three types: 1) syntactic context of the alarmed expressions; 2) general factors that influence the analysis accuracy; and 3) properties of the analysis results(estimated array indices).

Syntactic Context Syntactic symptoms describe the syntactic context around the alarmed expressions. For a given alarmed expression we gather the following symptoms from the function body that contains the alarmed expression:

AfterLoop
AfterReturn
InNestedBranchBody$N$
InBranchCond
InNestedFunParam
AfterBranch
InNestedLoopBody$N$
InLoopCond
InFunParam
InRightOfAnd

AfterLoop and AfterBranch are respectively turned on when loops and branches appear before the alarmed expressions. These symptoms are for false alarms; loop and branch can decrease analysis accuracy due to the join operations at their flow-join points. AfterReturn is on when a return statement precedes the alarmed expression. This symptom is for false alarms; while the return statements in the middle of function body are often for exiting on erroneous cases static analysis can blindly propagate such erroneous cases to independent yet later array access expressions. InNestedLoopBody$N$ and InNestedBranchBody$N$ are for true alarms because programmer are easy to mistake inside nested loops and branches. Since we found that simple nested structures were common in both true and false alarms we refined symptoms by their nesting depth $N = 1, 2, 3$, or >3. InLoopCond and InBranchCond are on when alarms are inside the condition expressions, and InFuncParam and InNestedFuncParam are on when alarms are inside function's actual argument expressions. These four symptoms are for true alarms because it is likely that expressions in those contexts are more carefully checked by programmers than expressions in other contexts. InRightOfAnd is for alarms in the right hand side of the logical-and && operator. This symptom is for false alarms because C's short-circuit semantics can skip executing the && operator's rhs expressions.

General Accuracy Factors Symptoms that can be detected only during the analysis can be useful indicators. Following symptoms are collected during the fixed point iterations:

Join$N$
Narrowed
Pruned
PassedVal
InStructure

The number of program points where the join operation occurs affects the analysis accuracy. This situation is captured by symptom Join$N$. N is the number of such program points before an alarmed expression. N ranges over $\{1, \ldots, 10, > 10\}$. The context pruning and the narrowing operations too are influencing factors for the analysis accuracy. Pruned and Narrowed are on when those operations are successful. PassedValue is on when array index values are passed as arguments, and InStructure is on when the target arrays are pointed to from some data structures (e.g. record fields). These two symptoms are for true alarms because such complicated use of the target arrays and indices are likely to be confused.

Properties of Estimated Array Indices The analysis results themselves are used as symptoms too:

TopIndex HalfInfiniteIndex FiniteIndex

If an estimated array index is the whole integer interval it is likely to be a false alarm (TopIndex). HalfInfiniteIndex is on when an index interval is half-infinite like $[1, \infty]$. Conversely, array indices with exact boundaries(FiniteIndex) strongly suggest true alarms.

4 Related Work

Reducing false alarms has always been a critical problem in static analysis. Existing tools have addressed the false alarm problem by 1) giving up the soundness of analysis (e.g. SPLINT[14], ARCHER[13]); 2) depending on user annotations (e.g. CSSV[7], SPLINT[14]); 3) limiting the target programs (e.g. ASTRÉE[4, 6]); 4) heuristically classifying the alarms into either true ones or false ones (e.g. Z-ranking[10]).

Airac differs from existing tools in that it uses Bayesian statistical analysis to classify the alarms by their probabilities being true. Our Bayesian approach can be orthogonally used with the user annotation approach. As of the analysis itself, it is sound, does not rely on user annotations, covers the full set of ANSI C constructs, and scale up to several 10K LOC.

CSSV[7] and SPLINT[14] rely on user annotations to reduce the false alarms. With imprecise or null user annotation, these tools have rapidly increasing false-alarm rate. ARCHER[13] is not sound, having a low detection-rate for bugs. ASTRÉE[4, 6] is a static program analyzer aiming at verifying the absence of run time errors in a limited number of avionics controller programs in C. This analyzer reports zero or very few false alarm. It excludes several C features (e.g. union types, dynamic memory allocation, and unbounded recursive function calls).

The one most directly related to our Bayesian approach is Z-ranking[10]. It ranks alarms by heuristics. It first partitions successes (e.g. safe buffer accesses) and failures (e.g. buffer overrun alarms) into groups. In each group, using a three heuristics, it computes "z-score" for each alarm being true. Alarms in decreasing order of z-scores are presented to the user. This approach has two drawbacks. The heuristics are only about the relative numbers of successes and failures in each group and they do not mention about any systematic method on how to partition the alarms. Thus if the partitioning happen to group alarms about which the heuristics fail to reflect the reality, Z-ranking can be ineffective. In comparison, our statistical approach is more robust. Our method has no arbitrary parameter like the "partitioning" in Z-ranking; it's competence does not rely on a particular factor of the method because the set of symptoms, which correspond to our method's heuristics, are extensive covering both the analyzer's internal behaviors and the input programs' syntactic characteristics; and lastly, thanks to the Bayesian framework's learning capability, our method's competence will improve as the analysis results are accumulated.

5 Conclusion and Discussion

We present that combining, in a realistic setting, a domain-unaware static analyzer with a Bayesian analysis can be a viable approach to handle false alarms. Our analyzer Airac, which collects all the true buffer-overrun points in ANSI C programs, is sound and its cost-accuracy improvement is achieved by techniques that static analysis community has long accumulated. For still inevitable false alarms we design a Bayesian post analysis. The statistical analysis, given the analysis results (alarms), estimates the probability of each alarm being true. The probabilities are used to sift out probable false alarms and prioritize true alarms. Only the alarms that have trueness higher than a threshold are reported to the user, and the alarms are sorted by the probability before reporting, so that the user can check highly probable errors first. In our experiments with Linux kernel sources and some textbook programs, if the user set the risk of missing true error is about 3 times greater than false alarming, 74.83% of false alarms could be filtered; and only 15.17% of false alarms were mixed up until the user observes 50% of the true alarms.

The Bayesian analysis' competence heavily depends on how we define symptoms. Since the inference framework is known to work well, better symptoms and feasible size of pre-classified alarms is the key of this approach. We think promising symptoms are tightly coupled with analysis' weakness and/or its preciseness, and some fair insight into the analysis is required to define them. However, since general symptoms, such as syntactic ones, are tend to reflect the programming style, and such patterns are well practiced within organizations, we believe local construction and use of the knowledge base of such simple symptoms will still be effective. Furthermore, we see this approach easily adaptable to possibly any kind of static analysis.

Another approach to handling false alarms is to equip the analyzer with all possible techniques for accuracy improvement and let the user choose a right combination of the techniques for her/his programs to analyze. The library of techniques must be extensive enough to specialize the analyzer for as wide spectrum of the input programs as possible. This approach lets the user decide how to control false alarms, while our Bayesian approach lets the analysis designer decide by choosing the symptoms based on the knowledge about the weakness and strength of his/her analyzer. We see no reason we cannot combine the two approaches.

Acknowledgements We thank Jaeyong Lee for helping us design our Bayesian analysis, Hakjoo Oh and Yikwon Hwang for collecting and identifying false alarm cases, and anonymous referees for helpful comments.

References

1. bugtraq. `www.securityfocus.com`.
2. CERT/CC advisories. `www.cert.org/advisories`.

3. James O. Berger. *Statistical Decision Theory and Bayesian Analysis, 2nd Edition.* Springer, 1985.
4. 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.
5. Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295. Springer-Verlag, 1992.
6. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, 2005.
7. 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.
8. Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis.* Text in Statistical Science. Chapman & Hall/CRC, second edition edition, 2004.
9. David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
10. Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In Radhia Cousot, editor, *SAS '03: Proceedings of the 10th Annual International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 295–315. Springer, 2003.
11. Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.
12. N. Metropolis and S. Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341, September 1949.
13. 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.
14. Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106. ACM Press, 2004.