

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

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, 1009 buffer-overrun alarms in commercial C programs of 636K lines and 183 among the 1009 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 lets us not hastily 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 detects buffer overruns¹ 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, 10, 8, 9].

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 ac-

curacy 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 raising false alarms.

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 indices. 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 values. We use interval domain $\hat{\mathbb{Z}}$ for abstract numeric values. $[a, b] \in \hat{\mathbb{Z}}$ 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. An abstract array (an abstract pointer to an array) is a triple which consists of its base location, its size interval, and an offset 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 $\langle l, [s, s], [0, 0] \rangle$.

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

*An extended version of this paper will be presented in SAS 2005. This work was supported by Brain Korea 21 of Korea Ministry of Education and Human Resource Development, and by National Security Research Institute of Korea Ministry of Information and Communication.

¹Buffer 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.

	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
Commercial Softwares	mptbase.c	6,158	0.79s	1	1	1
	aty128fb.c	2,466	0.32s	1	1	1
	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

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 uses 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.

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 nML² and tested to analyze GNU softwares, Linux kernel sources and commercial softwares.

²Korean dialect of ML programming language. <http://ropas.snu.ac.kr/n>

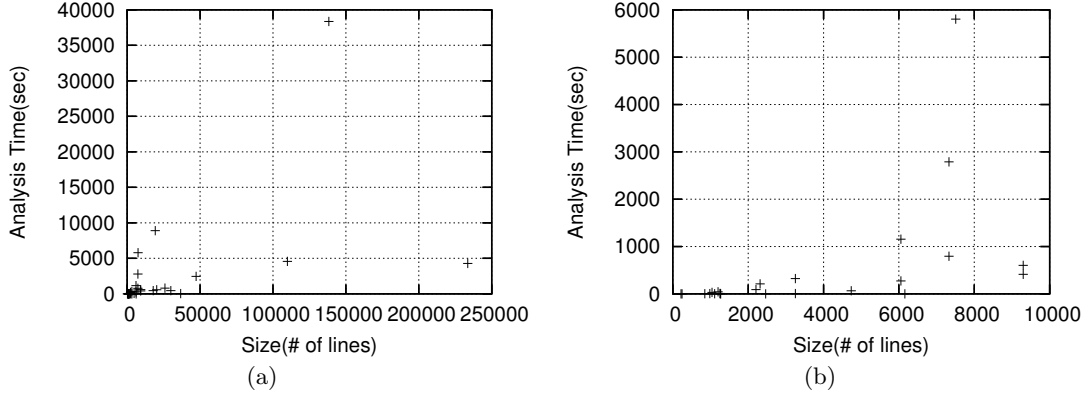


Figure 1: Airac's scalability

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

- Airac caught 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++;
    }
}
```

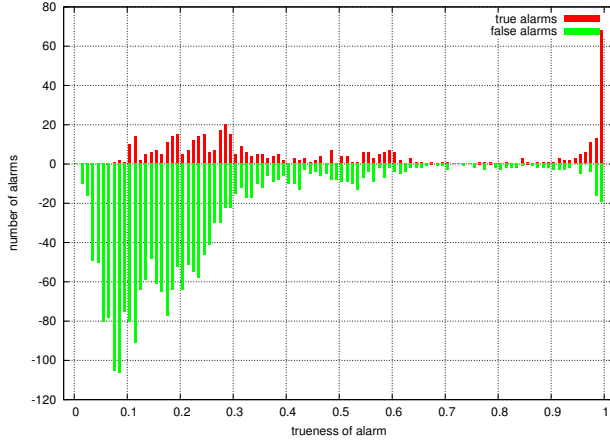
- Index variable is passed to a function by parameter and updated in the function:

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

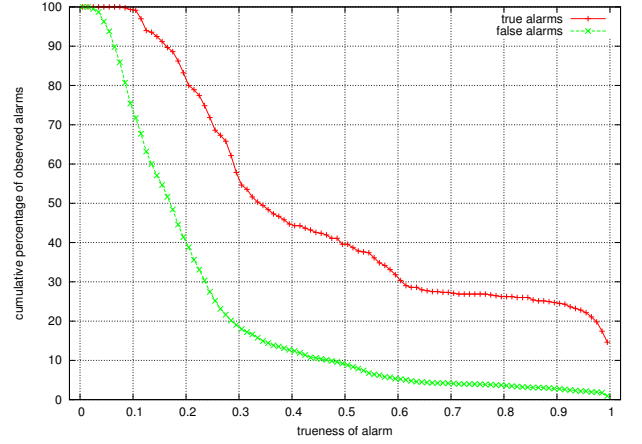
main() {
    simpleCal(2);
}
```

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 $P(A | B)$ 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



(a) Frequency of probability being true in true and false alarms. False alarms are counted in negative numbers. 74.83% of false alarms have probability less than 0.25.



(b) Cumulative percentage of observed alarms starting from probability 1 and down.

Figure 2: Experiment results

the probability of a new alarm with some symptoms being a true one.

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:

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

where p , θ_i and η_i have beta distributions as

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

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 $\psi_{j_1} < \psi_{j_2} < \dots < \psi_{j_N}$. We take the upper bound $\psi_{j_{(1-\alpha) \cdot N}}$ for $\hat{\psi}$. After obtaining the upper bound $\hat{\psi}$ of probability being true for each alarm, we have to decide whether we should report the alarm or not. To choose a reasonable threshold, user supplies two parameters defining the magnitude of risk: r_m for not reporting true alarms and r_f for reporting false alarms.

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

Given an alarm whose probability being true 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 probability to report the alarm can be chosen as:

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

If the probability of an alarm being true can be greater than or equal to such threshold, i.e. if the upper bound of $\hat{\psi}$ is greater than such threshold, then the alarm should be raised with $100(1 - 2\alpha)\%$ credibility. For example, user can supply $a_1 = 3, a_2 = 1$ if (s)he 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 probability of an alarm is greater than 0.25, 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. 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 $\hat{\psi}$.

Figure 2 was constructed from the data generated by repeating the experiment 15 times. For the histogram (a) on the left, dark bars indicate true alarms and lighter ones are false. 74.83% ($\approx 1504/2010$) of false alarms have probability 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 probability 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 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.

We can rank alarms by their probability to give effective results to 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 probable of true alarms are scattered over 0 through 1, we can see that most of the false alarms have small probability. Hence, sorting by probability and showing in decreasing order will effectively give true alarms first to the user. (b) of Figure 2 shows the cumulative percentage of observed alarms starting from probability 1 and down. Only 15.17% ($=305/2010$) of false alarms were mixed up until the user observes 50% of the true alarms, where the probability equals 0.3357.

5. Conclusion

Our Airac experience encourages us to conclude that in a realistic setting 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.

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 three times greater than that of false alarming, three quarters of false alarms could be sifted out. Moreover, if user inspects alarms having high probability first, only 15% of the false ones get mixed up while 50% of the trues are observed.

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. We thank Hakjoo Oh and Yikwon Hwang for their hardwork in collecting programs and classifying alarms used in our experiments.

6. 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. Csvg: 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*. Texts 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] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 83–93, New York, NY, USA, 2004. ACM Press.
- [10] 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.