AutoSense: A Framework for Automated Sensitivity Analysis of Program Data

Bernard Nongpoh*, Rajarshi Ray*, Saikat Dutta†, Ansuman Banerjee‡
*Department of Computer Science & Engineering, National Institute of Technology Meghalaya, Shillong, India.
†Department of Computer Science & Engineering, Jadavpur University, Kolkata, India.
‡Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Kolkata, India.

***Abstract*—In recent times, approximate computing is being increasingly adopted across the computing stack, from algorithms to computing hardware, to gain energy and performance efficiency by trading accuracy within acceptable limits. Approximation aware programming languages have been proposed where programmers can annotate data with type qualifiers (e.g. precise and approx) to denote its reliability. However, programmers need to judiciously annotate so that the accuracy loss remains within the desired limits. This can be non-trivial for large applications where error resilient and non-resilient program data may not be easily identifiable. Mis-annotation of even one data as error resilient/insensitive may result in an unacceptable output. In this paper, we present *AutoSense*, a framework to automatically classify resilient (insensitive) program data versus the sensitive ones with probabilistic reliability guarantee. AutoSense implements a combination of dynamic and static analysis methods for data sensitivity analysis. The dynamic analysis is based on statistical hypothesis testing, while the static analysis is based on classical data flow analysis. Experimental results compare our automated data classification with reported manual annotations on popular benchmarks used in approximate computing literature. AutoSense achieves promising reliability results compared to manual annotations and earlier methods, as evident from the experimental results.**

***Index Terms*—Approximate Computing, Sensitivity Analysis, Hypothesis Testing, Sequential Probability Ratio Test.**

I. INTRODUCTION

Performance efficiency and at the same time, low energy consumption is becoming a foremost concern for applications running in hand-held mobile systems to large data center applications. *Approximate computing* [17], [27] is a new computing paradigm that attempts to achieve both but at the expense of accuracy loss within acceptable limits. Approximate computing is also believed to be one of the alternate sources of efficiency given that trend in processor technology suggest an approaching saturation in the transistor counts and deceleration of Moore's law [14], [16]. Approximate computing leverages on the inherent tolerance of applications in several domains to inaccuracies in computation and data. An inaccuracy tolerant application is one in which parts of its computation or data or both can be inaccurate without disturbing the output beyond an acceptable limit. In the approximate computing paradigm, an application output is usually a continuous interval of values around the exact value, also known as the Quality of Service (QoS) band. An inaccuracy tolerant application produces output within this QoS band in the presence of inaccuracies in its computation or data or both. The inaccuracy tolerance in the applications can arise due to factors like inability of humans to perceive noise within limits, inherent nature of approximation in the output etc. Multimedia applications, search engines, gaming applications and data analytics are example domains where applications can be inherently tolerant to inaccuracy.

A standing challenge in approximate computing across the different computing stacks is to identify the error resilient components, that can in turn be realized and mapped onto the approximate implementations. At the application level, identifying program data which are error resilient against critical ones is a daunting challenge, since incorrect identification of critical data as error resilient can be catastrophic in terms of the output quality of the application. This has inspired a number of recent research articles in this direction [4], [5], [6], [22], [24]. However, existing work either propose programming language frameworks for manual annotations of approximable data [24], identification of critical code segments [5], [6] or propose automated analysis of data resiliency without any quantitative reliability guarantee [22].

In this work, we present AutoSense, a collection of systematic methods for program data classification with quantitative confidence guarantee. The contributions of this work are:

(a) We present a statistical method to classify program data as error resilient or critical based on dynamic analysis. Our method comes with a probabilistic guarantee derived from statistical tests.
(b) While the method above is purely dynamic, we additionally present a hybrid analysis framework that combines the power of static and dynamic analysis together to derive resiliency annotations more efficiently.
(c) We present experimental results on popular benchmarks in approximate computing to illustrate our method.

AutoSense can guide application designers to use programming language constructs like the one reported in [25] to effectively annotate resilient data components to gain energy efficiency without having any prior knowledge of the application domain. Moreover, *AutoSense* can be tuned with a probabilistic confidence parameter and an acceptable QoS band, depending on designer requirements, to perform partitioning accordingly.

AutoSense has been evaluated on the Scimark benchmark [18] and three applications from the AxBench benchmark [8]. In all the evaluated applications, manual classification of error resilient data has been reported in earlier work [25]. We show that introducing approximations in all the manually classified error resilient data results in failed QoS requirement for 62.5%

of the applications, for a large number of executions. On the other hand, 25% of the applications failed the same QoS requirement when approximation is introduced in the AutoSense derived error resilient data, for the same number of executions. However, in 12.5% out of the 25% of the applications that failed the QoS requirement, the percentage of executions that failed the required QoS is less than the maximum allowed threshold given by AutoSense, in its classification.

The organization of the paper is as follows. In Section II, we illustrate the notion of sensitivity of program data with an example and provide a formal definition of sensitivity of program data. We present statistical methods used in AutoSense in Section III. A static-dynamic combined sensitivity analysis framework is presented in Section IV. Section V presents the implementation of AutoSense. The experimental results are discussed in Section VI. Related work in the realm of automated data and code resiliency analysis and programming languages that allow approximation annotations are presented in Section VII. We conclude in Section VIII.

II. Problem Statement and Motivation

Sensitivity analysis has been applied to mathematical models of systems to understand a relation between the uncertainty in the system's output and the uncertainty in the input to the system. Questions like which are the system inputs that play a critical role in determining the variance of the system output can be answered with sensitivity analysis techniques [23]. Our focus here is to analyze the sensitivity of a program's output to its internal data rather than its input.

Let $\mathcal{I}$ and $\mathcal{R}$ denote the set of program input and output data respectively. Let $\mathcal{D}$ be the program data neither in $\mathcal{I}$ nor in $\mathcal{R}$. The objective of sensitivity analysis is to partition the set $\mathcal{D}$ into the set of sensitive data, $\mathcal{SD}$ and the set of insensitive data $\overline{\mathcal{SD}} = \mathcal{D} - \mathcal{SD}$. The notion of sensitivity of program data with respect to a user defined Quality of Service (QoS) is formally defined as follows. Let E be the set of all possible executions of a program $\mathcal{P}$. Given an execution $e \in E$ and a program data $v \in \mathcal{D}$, let (v_e, ℓ) denote the value of v at program point ℓ in $\mathcal{P}$ during the execution e. We term this value as *exact* value of v at location ℓ of $\mathcal{P}$ with respect to the execution e. Evidently, there could be multiple locations and therefore multiple (v_e, ℓ) values for the execution e. Let the set of program locations where v occurs in an execution e be denoted as ℓ_v^e. Let $(v_{approx}, \ell) \neq (v_e, \ell)$ denote any value of v at location ℓ. We term this as a candidate *approximate* value of v at location ℓ in $\mathcal{P}$ with respect to the execution e. The definition of sensitivity of v is now defined as:

Definition 1: Given an acceptable QoS band for a program $\mathcal{P}$ and a sensitivity threshold probability θ, a program data $v \in \mathcal{D}$ is called sensitive if and only if $\forall e \in E$, the probability that the program output remains in the acceptable QoS band when every instance (v_e, l) in e is replaced with some (v_{approx}, ℓ), is less than θ. Formally, the definition is:

$$\mathcal{SD} = \big\{ v \in \mathcal{D} \mid \forall e \in E, \forall \ell \in \ell_v^e, (v_e, \ell) \rightarrow (v_{approx}, \ell) \implies Pr(\mathcal{R} \in QoS\ ) < \theta \big\} \quad (1)$$

where $(v_e, \ell) \rightarrow (v_{approx}, \ell)$ denotes the substitution of (v_{approx}, ℓ) in place of (v_e, ℓ). The set of *insensitive* data is defined as $\overline{\mathcal{SD}} = \mathcal{D} - \mathcal{SD}$.

To illustrate the notion of sensitivity of program data, we present a simple example program of a binary search procedure to search for the presence of a key element in an input array, in Listing 1. The program has a data set $\mathcal{D} = \{lo, hi, size, mid\}$, input data set $\mathcal{I} = \{a\}$ and the output data set $\mathcal{R} = \{ret\}$, where ret denotes the return value of the procedure. We consider a strict QoS here, which states that the output is acceptable only when it produces the accurate output, i.e., the procedure should return a $true$ when the search key is present in the input array and $false$ otherwise.

```
1 : bool binsearch(int lo, int hi)
2 : {
3 :     unsigned int size = hi-lo + 1;
4 :     unsigned int mid = (lo+hi)/2;
5 :     if(lo>hi) return false;
6 :
7 :     if (size >= 1){
8 :         if(a[mid] == key)
9 :           return true;
10:         else if(a[mid]>key)
11:           return binsearch(lo, mid-1);
12:         else
13:           return binsearch(mid+1, hi);
14:     }
15:     return false;
16:}
```

Listing 1. Binary Search

It may be observed from the example that the data $size$ does not affect much the output, the return value of the program. For any inexact value that $size$ may take other than 0, the binary search procedure will return an acceptable output. Therefore, we may deduce that $size$ is likely to be not sensitive to the output. Notice that the data $size$ is not *dead*[1] with respect to the program output as there exists a valuation in the data range of $size$, the value 0, such that when $size$ takes 0 as an inexact value, the program output may not be giving acceptable QoS. On the other hand, the data lo, hi and mid are likely to be sensitive to the program output since they are used in computing the indices of the input array a within which to search for the key. Observe that in line 4, mid depends on the data lo and hi and in line 8, mid is used as an array index. Therefore, an inexact value in any of lo, hi or mid may result in the array index mid in line 8 to be outside the allocated memory for array a. This may cause memory error or unacceptable output. It is to be observed that for a multi-line code with a complex control and data flow, it is hard to classify sensitive and insensitive data by manual inspection.

The notion of acceptable QoS depends on the application under consideration. As an example, the QoS of an image rendering application can be measured by the PSNR (Peak

[1] A dead variable is one whose value has no effect on the program output

Signal to Noise Ratio) of the image. The output image of the application is deemed acceptable, given that the PSNR is less than a predefined acceptable threshold. An acceptable QoS band is user defined and it specifies the degree of precision desirable or in other words, how much approximation is acceptable to a user. In the experiments section, we define the QoS metrics considered for the benchmark applications.

III. Dynamic Sensitivity Analysis

In this section, we present our dynamic sensitivity analysis method implemented inside AutoSense for automatic classification of sensitive and insensitive program data. AutoSense provides probabilistic guarantees on the classification, i.e., it identifies application data which are insensitive to the QoS with a probability at least θ, where θ is user specified. Since AutoSense is based on probabilistic methods, the classification may have errors. However, the probability of making an error can be bounded in the framework. To test for sensitivity of a data, our idea is to deliberately inject inexact value in the data and execute the program in the presence of inexactness. Such an execution of the program is considered as a random experiment. The outcome of the program is observed to check if it lies within the acceptable QoS band.

As an example, consider an application that renders images. We inject an inexact value in one of the application data and observe the noise in the final rendered image in the presence of inexactness to judge if it is to be rejected or accepted. In this way, every fault injected execution of an application can be interpreted as a Bernoulli trial with two possible outcomes, a *success* and a *failure*. The outcome is a *success* if the application output remains within the acceptable QoS band and is a *failure* otherwise.

Broadly, our framework consists of (1) conducting Bernoulli trials by executing fault injected applications and (2) performing statistical analysis of the outcomes of the Bernoulli trials to identify sensitivity of application data. We discuss the fault injection mechanism and the statistical methods used in our work in the following sections.

A. Fault Injection Model

An important step in our method is fault injection, whereby we inject faults in an application execution. The purpose is to emulate an approximate computation and observe its effect on the output quality. Our fault injection model is similar to the one proposed in [22]. There are standard tools for fault injection in a program. [13] is a LLVM [10] based fault injection tool. We however, implemented our own fault injector for better control, as discussed in Section V. In the fault model, we inject faults in data write operations during an execution of the program. A program data to be tested for sensitivity is selected and faults are injected at every assignment to this data during an execution. This allows us to capture the effect of faults in the chosen data under test only and therefore, the sensitivity analysis of a data is not influenced by the sensitivity of other application data. No fault is injected when a data is read. Since any program data is expected to be initialized before use, the proposed fault model presents an inexact value at every use of the data under test during execution. If the data under test has no assignments during an execution, then the fault model injects no faults, resulting in an exact computation.

B. Problem Formulation

Our work follows in the lines of [30], [31] which proposed probabilistic model checking using acceptance sampling. Acceptance sampling is used in the context of quality control of production systems where a sample of products are examined to either accept or reject the production system [9]. We use hypothesis testing [29] for acceptance sampling in our framework based on finitely many samplings. In our context, a sample is an execution of the application with an input and an injected error into an application data. Such an execution of the application is interpreted as a Bernoulli trial with a *success* or *failure* outcome depending on the QoS of the output. The requirement is that an experiment should complete in finite time to produce an output. Notice that we might have an infinite run of an application after an inexact value is injected into a program data. This could be possible if the injected inexact values induce an unbounded loop in the program. In our framework, we observe an execution for a reasonable time bound and deem the trial as *failed* if it does not terminate within the time bound. Observe that the statistical experiment of observing outcomes of n samples (trials) models a Binomial distribution, with the probability of observing exactly x number of *success* given by:

$$f(x) = \binom{n}{x} p^x (1-p)^x \tag{2}$$

where p is the probability of observing a *success* outcome of a trial. The mean of the distribution is $\mu = np$ and the variance is $\sigma^2 = np(1-p)$. For data sensitivity analysis using acceptance sampling, we need to choose the number of trials n and the expected number of *success* outcomes x out of the n trials to deem the data under test as tolerable to approximation. To have a probabilistic confidence on the acceptance sampling, we may decide on a minimum threshold on the probability of observing at least x *success* outcomes, given by:

$$\begin{aligned} f(\geq x) &= 1 - f(< x) \\ &= 1 - \sum_{k=0}^{x-1} \binom{n}{k} p^k (1-p)^{n-k} \end{aligned} \tag{3}$$

Note that the probability given by Eq. 3 depends on n, x and p. If we consider the value of p as 0.5, we need to choose an n and x such that the probability of Eq. 3 is larger than the threshold. For any program data, we can then perform n fault injection trials and if the number of successful outcomes is larger than or equal to the computed x, we can classify the data as *insensitive*. However, this method does not provide any probability guarantee on the sensitivity classification since the considered minimum probability of observing at least x *success* outcomes depends on the number of samples n and the probability of a *successful* outcome p. Moreover, there is no bound on the probability of making an error, i.e., accepting a *sensitive* data erroneously as *insensitive* or vice-versa.

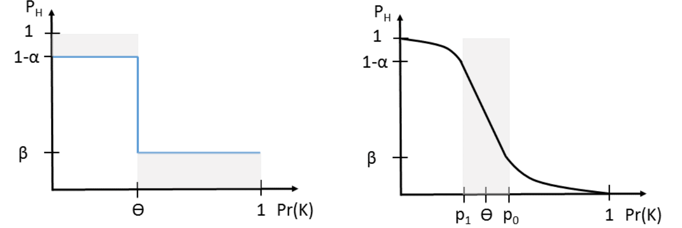
C. Acceptance Sampling using Hypothesis Testing

To address the discussed issues of acceptance sampling using the probability density function of Binomial distribution, we propose to perform acceptance sampling using hypothesis testing. We assign an hypothesis and a contrary hypothesis for every program data. For every $v \in \mathcal{D}$, we propose a hypothesis that $\forall e \in E, \forall \ell \in \ell_v^e, (v_e, \ell) \rightarrow (v_{approx}, \ell) \implies \mathcal{R} \in QoS$, where E, ℓ_v^e, (v_e, ℓ) and (v_{approx}, ℓ) are as defined in Definition 1. Let us denote such an hypothesis by K. In our analysis, we test the following null and contrary hypothesis:

$$\begin{aligned} H &: Pr(K) < \theta \\ H' &: Pr(K) \geq \theta \end{aligned} \tag{4}$$

where $Pr(K)$ is the probability that the hypothesis K is true. Note that the truth of the hypothesis H implies that the program data v is *sensitive* by Definition 1. Similarly, the truth of the contrary hypothesis H' implies that the data v is *insensitive*. A successful Bernoulli trial is an evidence of hypothesis whereas a failed trial is an evidence of the contrary hypothesis. The test of hypothesis can be performed using procedures for hypothesis testing [12], [30]. In this way, sensitivity analysis of application data is formulated as an instance of the classical *hypothesis testing* problem.

Any statistical procedure for hypothesis testing has a probability of accepting a false hypothesis. However, it is possible to have the probability of making an error reasonably low. The probability of accepting the contrary hypothesis H' when H holds is denoted as α and is called a Type I error or false negative [11]. Similarly, the probability of accepting H when H' holds is denoted by β and is called a Type II error or false positive [11]. We expect that the testing procedure has both α and β to be low (generally less than 0.5). The parameters α and β define the strength of the acceptance sampling test. The probability of accepting the hypothesis H (P_H) with acceptance sampling test of strength $\langle \alpha, \beta \rangle$ is shown in Figure 1(a). When $Pr(K) < \theta$, the null hypothesis is accepted with a probability of at least $1 - \alpha$, shown as the grey region to the left of θ in the figure. When $Pr(K) \geq \theta$, the null hypothesis is accepted with a probability of at most β, shown as the grey region to the right of θ. Observe that in such an acceptance sampling test, the probability of accepting H when $Pr(K) = \theta$ should be at most β and for $Pr(K) = \theta - \epsilon$, where ϵ is infinitesimally small, the probability of accepting H should be at least $1 - \alpha$, that is largely different from β. Such an acceptance testing would either demand nearly exhaustive sampling of the sample space, which is infeasible for large sized sample space or will have $\alpha = 1 - \beta$, meaning that keeping the probability of making Type I error low shall make the probability of Type II error large and vice-versa. To overcome this problem, the use of *indifference region* has been proposed in the literature [30]. Two probabilities p_0, p_1 close to θ are used such that $p_0 > p_1$ and new hypotheses $H_0 : Pr(K) \leq p_1$ and $H_1 : Pr(K) \geq p_0$ are tested instead. The null hypothesis H is accepted if H_0 is accepted and the contrary hypothesis H' is accepted if H_1 is accepted. If the probability $Pr(K)$ lies in the interval $[p_1, p_0]$, the test is indifferent to both the null and the contrary hypotheses and there is no bound on the probability of accepting a false hypothesis. This is why the probability region defined by the interval $[p_1, p_0]$ is called the indifference region. Figure 1(b) shows the typical characteristics of a realistic acceptance sampling test using indifference region [30], shown as the gray shaded area. Indifference region allows a smooth transition from accept to reject decision and as it goes narrow, we approach the ideal behavior. We now discuss the hypothesis testing procedure used in our framework.



(a) Probability of Accepting the Null Hypothesis H with a Hypothetical Acceptance Sampling Test of Strength $\langle \alpha, \beta \rangle$

(b) Probability of Accepting the Hypothesis $H_0 : Pr(K) \leq p_1$ with a Typical Acceptance Sampling Test of Strength $\langle \alpha, \beta \rangle$ using Indifference Region.

Fig. 1. Indifference Region in Acceptance Testing

D. Sequential Probability Ratio Test

We use sequential probability ratio test (SPRT) out of the many algorithms for hypothesis testing [28], [30]. The principle behind SPRT is to decide whether additional experiments need to be performed to accept or reject a hypothesis on the basis of the previously observed outcomes. It requires provably an optimal number of trials to test an hypothesis when $Pr(K) = p_0$ or p_1 [28]. After conducting k Bernoulli trials with outcomes $x_1, \ldots, x_k$, the procedure computes the ratio of two probabilities as shown below:

$$\frac{p_{1k}}{p_{0k}} = \prod_{i=1}^{k} \frac{Pr[X_i = x_i | p = p_1]}{Pr[X_i = x_i | p = p_0]} = \frac{p_1^{b_k}(1-p_1)^{k-b_k}}{p_0^{b_k}(1-p_0)^{k-b_k}} \tag{5}$$

where X_i is a random variable associated with the ith Bernoulli trial and x_i is the outcome of the trial. p_{1k} and p_{0k} denote the probabilities of observing the sequence $x_1, \ldots, x_k$ given that $Pr[X_i = 1] = p_1$ and $Pr[X_i = 1] = p_0$ respectively. $b_k = \sum_{i=1}^{k} x_i$ is the number of successful trials. The algorithm terminates by accepting the hypothesis H_0 if:

$$\frac{p_{1k}}{p_{0k}} \leq B \tag{6}$$

and it accepts the hypothesis H_1 if:

$$\frac{p_{1k}}{p_{0k}} \geq A \tag{7}$$

In practice, we choose $A = \frac{1-\beta}{\alpha}$ and $B = \frac{\beta}{1-\alpha}$ for hypothesis testing with strength $\langle \alpha, \beta \rangle$ to closely match the strength [28]. The algorithm is guaranteed to terminate, either accepting H_0 or H_1. In our framework, the test strength $\langle \alpha, \beta \rangle$ and the indifference region ($[p_1, p_0]$) are user defined parameters.

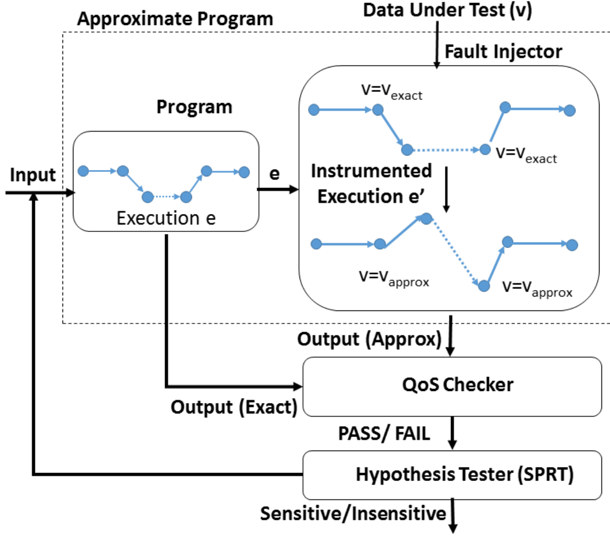


Fig. 2. Framework of Dynamic Sensitivity Analysis with Hypothesis Testing

We first discuss about the running time of the method used. The running time of SPRT depends on two parameters, (1) the number of trials to decide the acceptance of the hypothesis and (2) the time taken to complete a Bernoulli trial. It is shown that the number of trials depends on the distance of the actual probability $Pr(K)$ to the indifference region [28]. The number of trials tends to increase as $Pr(K)$ gets closer to the indifference region and gets maximum when it is equal to the center of the indifference region. The number of samples decreases as $Pr(K)$ moves away from the indifference region. The time taken to complete a Bernoulli trial depends on the application. Compute intensive applications are expected to result in reduced efficiency compared to efficient applications.

A schematic of the dynamic analysis method with the fault model is shown in Figure 2. In Algorithm 1, program data are tested for sensitivity with a hypothesis tester and partitioned as either *sensitive* or *insensitive*. The hypothesis testing procedure is shown in Algorithm 2. The null hypothesis is assigned on the data to be tested in line 3. The exact output of the program is computed on an input in line 4. Bernoulli trials are performed in a loop until the null hypothesis is accepted or rejected in line 10, by executing the program with the same input as in line 4 in the presence of faults in the data x, to get an approximate output. The approximate output is compared with the exact one to either accept or reject the output as per the acceptable QoS requirement in line 11. The SPRT test updates the ratio of p_{1k} to p_{0k} and compares it with A and B to decide on the acceptability of the hypothesis in lines 16 to 21.

TABLE I
DYNAMIC SENSITIVITY ANALYSIS ON BINARY SEARCH

	Confidence Measure θ							
Data	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
lo	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$
hi	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$
size	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$
mid	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$

Algorithm 1 Dynamic Sensitivity Analysis

1: **function** PARTITIONDATA($\mathcal{P}$, $\mathcal{D}$)
2: $\quad\overline{\mathcal{SD}} \leftarrow \emptyset$, $\mathcal{SD} \leftarrow \emptyset$
3: $\quad$Initialize θ, indiff. region $[p_1, p_0]$, test strength α, β
4: $\quad$**for all** $x \in \mathcal{D}$ **do**
5: $\quad\quad res \leftarrow$ TestSensitivity($\mathcal{P}$, x, θ, p_0, p_1, α, β)
6: $\quad\quad$**if** res **then**
7: $\quad\quad\quad\mathcal{SD} \leftarrow \mathcal{SD} \cup \{x\}$
8: $\quad\quad$**else**
9: $\quad\quad\quad\overline{\mathcal{SD}} \leftarrow \overline{\mathcal{SD}} \cup \{x\}$
10: $\quad\quad$**end if**
11: $\quad$**end for**
12: **end function**

Algorithm 2 Testing Sensitivity with Hypothesis Testing

1: **function** TESTSENSITIVITY($\mathcal{P}$, x, θ, p_0, p_1, α, β)
2: $\quad$Assign hypothesis K for data x
3: $\quad$Assign null hypothesis $H_0 : Pr(K) \leq p_1$
4: $\quad r_e \leftarrow$ execution of $\mathcal{P}$ on an input ip
5: $\quad A = \frac{1-\beta}{\alpha}$, $B = \frac{\beta}{1-\alpha}$
6: $\quad k = 0$ $\quad\triangleright$ number of Bernoulli Trials
7: $\quad b_k = 0$ $\quad\triangleright$ number of successful Bernouilli Trials
8: $\quad p_{1k} = p_1^{b_k}(1-p_1)^{k-b_k}$, $p_{0k} = p_0^{b_k}(1-p_0)^{k-b_k}$
9: $\quad$**while** H_0 not accepted/rejected **do**
10: $\quad\quad r_a \leftarrow$ execution of $\mathcal{P}$ on input ip with faults in x
11: $\quad\quad$**if** QoS(r_a, r_e) acceptable **then**
12: $\quad\quad\quad k = k+1$, $b_k = b_k + 1$ $\quad\triangleright$ Successful trial
13: $\quad\quad$**else**
14: $\quad\quad\quad k = k+1$
15: $\quad\quad$**end if**
16: $\quad\quad p_{1k} = p_1^{b_k}(1-p_1)^{k-b_k}$, $p_{0k} = p_0^{b_k}(1-p_0)^{k-b_k}$
17: $\quad\quad$**if** $\frac{p_{1k}}{p_{0k}} \leq B$ **then**
18: $\quad\quad\quad$Accept H_0 and return true
19: $\quad\quad$**end if**
20: $\quad\quad$**if** $\frac{p_{1k}}{p_{0k}} \geq A$ **then**
21: $\quad\quad\quad$Reject H_0 and return false
22: $\quad\quad$**end if**
23: $\quad$**end while**
24: **end function**

The results of running the dynamic sensitivity analysis on the motivational example of binary search in Listing 1 is shown in Table I. The confidence θ of the analysis is the probability θ of Equation 4. A program data is classified as $\mathcal{SD}$ in the table corresponding to a θ if the analysis accepts the hypothesis H of Eqn 4. Observe that the data *lo* and *hi* are classified *insensitive* with a confidence of θ from 0.3 to 0.5 but not for 0.6 and beyond. The data mid is classified as sensitive for all confidence θ from 0.3 to 1.0, since it is an array index data. The data $size$ is marked as insensitive with probability 0.9 and we do expect it to be highly insensitive. Observe that the data $size$ is marked as $insensitive$ with a confidence $\theta = 1$ by the analysis, however, *size* is not a dead program data as discussed previously. This is an instance when the algorithm accepts the contrary hypothesis, $H' : Pr(K) = 1$ when the null hypothesis, $H : Pr(K) < 1$ is true and therefore exhibits

a Type I error. Therefore, we see that the analysis may infer sensitivity of program data wrongly due to the presence of Type I and Type II errors, however, with bounded probability.

Limitations of Dynamic Analysis

The dynamic analysis step requires a number of program executions to complete hypothesis testing. There can be applications where the executions are sensitive to the input. In such applications, the executions used for testing should be selected with an input sampling strategy in order to satisfy some coverage criteria such as all branch or all statement coverage. This might require the use of automated test case generators to intelligently sample the inputs for testing. However, for applications for which automated test case generators cannot generate inputs, a corpus of inputs for the dynamic analysis engine is to be created manually and this can be challenging. Another limitation of dynamic analysis is that there can be code and data covered by complex conditionals which are triggered in very few specific executions. Such data are likely to be marked as insensitive by the analysis but they can be critical. For example, if there is data used in triggering an error handler inside a complex conditional branch, then such a critical data gets classified as insensitive. Perhaps global sensitivity analysis (GSA) techniques might be useful in such applications instead of regionalised sensitivity analysis (RSA) and this remains a subject of future research.

Sensitivity Analysis and Statistical Methods

Statistical methods are good tools for sensitivity analysis in mathematical and computational models. For example, the two-sample Kolmogorov-Smirnov test (K-S test) is a statistical method for testing the hypothesis that the given two cumulative distribution functions (CDF) are identical. The maximum distance between the CDFs (D-statistic) is computed and the test accepts the hypothesis if this distance is less than a critical threshold. Such a statistical procedure for hypothesis testing can be used for sensitivity analysis of a model. A standard way of sensitivity analysis using K-S test is to run Monte-Carlo simulations of the model on different values of the input factors as statistical experiments and partition the simulation output as either an *acceptable* or *unacceptable* event, based on some criteria. The criteria of acceptance of a model behavior or simulation output could be any metric showing the deviance of the observed behavior from the expected behavior. The CDF of the acceptable and unacceptable events are passed to the K-S test. A rejected hypothesis indicates a considerable distance between the two CDFs, indicating sensitivity of the model to the input factors. Similarly, there are other sensitivity analysis techniques based on statistical methods such as variance and correlation based analysis, Bayesian uncertainty estimation etc. There are sensitivity analysis techniques not using statistical methods too. A simple such analysis is by computing the partial derivative of a model output with respect to an input factor. These remain to be explored in future.

IV. A Static-Dynamic Combined Sensitivity Analysis

For programs having a large number of data, the dynamic analysis can be slow since hypothesis for each data is tested. Moreover, compute and data intensive programs may take a long time to terminate, making each trial during the hypothesis testing expensive and slowing down the overall analysis. To address these performance issues, we propose static analysis to aid the dynamic analysis step.

Data flow analysis is a common static analysis technique. The idea behind data flow analysis is to abstract the program structure as a control flow graph and define data flow equations specifying the value of the analysis of interest at the entry and exit points of the program statements. Figure 4 shows an example of the control flow graph of a simple average program. The entry point of a program statement $stmt$ means the point of the program just before $stmt$ and the exit point means the program point immediately following $stmt$. The equations are solved for a *may* or *must* analysis solution at the program points. A *may* analysis solution at a program point is satisfied by at least one execution path. The *must* analysis solution at any program point is satisfied by every execution path. There is an underlying framework in solving the data flow equations, called the *Monotone framework* [15]. Any data flow analysis which is formulated as a monotone framework can be solved with a generic maximum fixed point algorithm (MFP) [15]. A monotone framework consists of two components, a complete lattice L over the property space satisfying the ascending chain condition and a set $\mathcal{F} : L \rightarrow L$ of monotone functions closed under composition and containing the identity function. Monotonicity and the ascending chain property of the lattice guarantee termination of the MFP algorithm on lattices of finite height. The property space is the domain of values of interest to any program analysis, e.g., intervals for interval analysis, set of variables for live variable analysis etc. The set $\mathcal{F}$ contains the transfer functions of the analysis. A transfer function $f_\ell \in \mathcal{F}$ specifies how the program statement at a program point ℓ can transform the value of the analysis.

We present a static analysis of program data sensitivity as an instance of the monotone framework. An instance of a monotone framework consists of the following:

- A complete lattice, L, of the framework
- The space of functions, $\mathcal{F}$, of the framework.
- A finite control flow graph of the program P, $flow(P)$.
- A set of initial labels, E, containing the labels of statements of program P which are its entry points.
- An initial value of the analysis, $init \in L$, for labels $\in E$.
- A mapping, $f : Labels \rightarrow \mathcal{F}$, which maps each program statement label to a transfer function in $\mathcal{F}$.

where a label of a statement in a program is a unique natural number assigned to the statement. $Labels$ is the set of all such labels of a program. The elements of the complete lattice L of our analysis are mappings $\sigma : \mathcal{D} \rightarrow \{\bot, S, I, \top\}$. $\sigma(x) = \bot$ denotes that no information is known about the data x whereas $\sigma(x) = \top$ denotes that x may be *sensitive* or *insensitive*. $\sigma(x) = S$ and $\sigma(x) = I$ denote x to be *sensitive* and

insensitive respectively. Borrowing from the work on language based information flow methods to ensure security properties in [2], we define a *data sensitivity lattice* over the range of σ, i.e., $\{\perp, S, I, \top\}$ as shown in Figure 3. The partial order on σ is defined in Eq. 8.

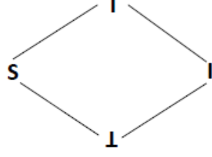


Fig. 3. Data Sensitivity Lattice

$$\begin{aligned} &\forall \sigma : \perp \sqsubseteq \sigma \\ &\forall \sigma_1, \sigma_2 : \sigma_1 \sqsubseteq \sigma_2 \textit{ iff } \forall x, \sigma_1(x) \sqsubseteq_D \sigma_2(x). \end{aligned} \tag{8}$$

where $\perp \in \sigma$ maps every $x \in \mathcal{D}$ to $\perp$, $\sqsubseteq$ denotes the partial order relation on σ and $\sqsubseteq_D$ denotes the partial order relation of the *data sensitivity lattice*. The *join* operation over σ is defined in Eq. 9.

$$(\sigma_1 \sqcup \sigma_2)(x) = \sigma_1(x) \sqcup \sigma_2(x) \tag{9}$$

The transfer functions in our analysis are based on the assumption that approximate or *insensitive* data does not flow into *sensitive* data. A similar approach is adopted in the EnerJ programming model [24] where programmers have the provision to annotate datatypes with @approx or @precise keywords but the type-checker does not allow assignment of approximate data into precise ones. Considering such a flow restriction on approximate programs, we define the transfer functions of our assignment statements. The transfer function for all other types of statements are considered to be an identity function. Considering a general assignment statement block $[x := a]$, a being any expression, we define the transfer functions of our analysis as:

$$\begin{aligned} [x = a] : f(\sigma) &= \begin{cases} \sigma(x \to I) & \textit{if } \forall v \in FV(a), \sigma(v) = I \\ \sigma(x \to S) & \textit{if } \forall v \in FV(a), \sigma(v) = S \\ \sigma(x \to \top) & \textit{if } \exists u,\, v \in FV(a) \\ & \quad \textit{s.t. } \sigma(u) = S,\, \sigma(v) = I \\ \sigma & \textit{if } FV(a) = \emptyset \end{cases} \\ [\cdots] : f(\sigma) &= \sigma \end{aligned} \tag{10}$$

where $[\cdots]$ denotes any program statement which is not an assignment statement and $FV(a)$ is the set of all free variables of the expression a. Essentially, we classify the data in the lhs of the assignment statement as *insensitive* if the data in the rhs of the assignment statement are classified as *insensitive*. In this case, since *insensitive* data is flowing in the lhs data x, we classify it to be $insensitive$ as well. Similarly, we classify the data in the lhs as *sensitive* if the data in the rhs of the assignment statement are already classified to be *sensitive*. This expresses the condition that *sensitive* data should flow into *sensitive* data only. When there are both *sensitive* and *insensitive* data in the rhs, we are *inconclusive* about the sensitivity information of the lhs data x, and we assign it to the $\top$ element of the *data sensitivity lattice*. In the last case, when there is no free variable in the rhs (i.e, a is an expression with constants), we keep the sensitivity mapping of the argument. We now present the definition of our static sensitivity analysis:

Definition 2: Sensitivity Analysis ($\mathcal{SA}$) is an instance of a monotone framework consisting of:

(i) The complete lattice $L = (\sigma, \sqsubseteq, \sqcup)$ such that $\sigma : \mathcal{D} \to \{\perp, S, I, \top\}$ and $\sqsubseteq$, $\sqcup$ are as defined in Eq. 8 and Eq. 9 respectively.
(ii) The set of monotone functions $\mathcal{F} = \{f : \sigma \to \sigma\}$
(iii) A finite flow graph of the program, $flow(P)$.
(iv) A set of initial labels of the program, E.
(v) An initial value of the analysis, $\sigma_{init} \in L$, for each label in E.
(vi) A mapping, $f : Labels \to \mathcal{F}$, which maps the labels ℓ of the assignment statements of P to the functions in $\mathcal{F}$ as defined in Eq. 10 and maps all other statement labels to the identity function.

σ_{init} is a given initial data sensitivity mapping. If no sensitivity information is known initially, then σ_{init} is $\perp$. Note that with $\sigma_{init} = \perp$, our proposed analysis is expected to produce a solution of $\sigma = \sigma_{init} = \perp$, since it propagates the initial known sensitivity information by the data flow relations. Therefore, the effectiveness of the proposed method depends largely on σ_{init}. In this context, Section IV-A discusses our approach of using the previously proposed dynamic analysis methods to initialize σ_{init}. While solving our static analysis, we do not update the sensitivity mapping of program data that is already mapped to *sensitive*(S) or *insensitive*(I) by σ_{init}.

It is easy to see that L satisfies the ascending chain condition since the number of elements in L is finite. $\mathcal{F}$ contains the identity function and is closed under composition. An instance of the analysis can be solved using the maximum fixed point algorithm (MFP) [15] using Eq. 11.

$$\begin{aligned} &\mathcal{SA}_{entry}(\ell) = \begin{cases} \bigsqcup \mathcal{SA}_{exit}(\ell') \mid (\ell', \ell) \in flow(P) & \textit{if } \ell \notin E \\ \perp & \textit{if } \ell \in E \end{cases} \\ &\mathcal{SA}_{exit}(\ell) = f_\ell(\mathcal{SA}_{entry}(\ell)), \textit{ where } f_\ell = f(\ell). \end{aligned} \tag{11}$$

$\mathcal{SA}_{entry}(\ell)$ and $\mathcal{SA}_{exit}(\ell)$ denote the solution of the analysis at the entry and exit points of the statement labeled ℓ. After obtaining a solution of the analysis, we get the set of sensitive data $\mathcal{D}$ using Eq. 12.

$$\mathcal{SD} = \bigsqcup \{\mathcal{SA}_{exit}(\ell) \mid \ell \in final\} \tag{12}$$

where $final$ is the set of labels of the exit points of a program. Eq. 12 states that the data sensitivity is the join of the sensitivity derived at all the exit points of the program.

A. Combining Dynamic Analysis

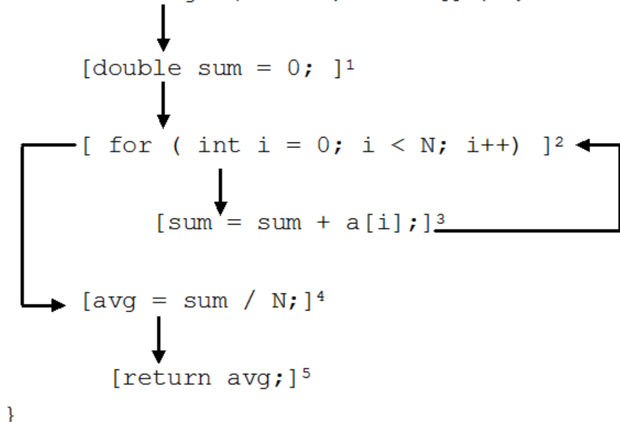
As discussed, the effectiveness of the static analysis depends on the initial known sensitivity information in σ_{init}. We

propose to obtain σ_{init} by running our dynamic analysis presented in Section III on some of the data from $\mathcal{D}$. In AutoSense, we perform dynamic analysis on the following types of data, (1) global data, (2) method parameters and (3) method local data whose expression has constant(s) or function call(s). AutoSense applies static analysis method-wise. We perform the sensitivity of method parameters and global data dynamically as method local data are likely to be data dependent on its method parameters and global data. Local variables which are initialized to constants are tested dynamically because the static analysis would not be able to identify the sensitivity of such variables. Method variables for which the assignment expression has a function call are also tested dynamically because their sensitivity depends on the sensitivity of return value of the functions which might not be known at the time of solving the analysis statically.

B. Illustration of Static-Dynamic Combined Analysis

The MFP algorithm is an iterative procedure that processes the edges (ℓ, ℓ') of the flow graph $flow(P)$, until no further edges are left for processing. The edges to be processed are collected in a data structure called $worklist$. Initially, all the edges of $flow(P)$ are inserted in the $worklist$. Another data-structure, $Analysis$, stores the current analysis solution at the entry of a statement labeled at i at $Analysis(i)$. Initially, $Analysis(i)$ contains the initial value of the analysis. In an iteration, an edge (ℓ, ℓ') is removed from the $worklist$ and the transfer function $f_\ell = f(\ell)$ is applied on $Analysis(\ell)$. If $f_\ell(Analysis(\ell))$ is not $\sqsubseteq$ $Analysis(\ell')$ then $Analysis(\ell')$ is updated to $f_\ell(Analysis(\ell)) \bigsqcup Analysis(\ell')$ and all the edges (ℓ', ℓ'') in $flow(P)$ are inserted into the $worklist$. The reason for inserting the edges (ℓ', ℓ'') into the $worklist$ is that since $Analysis(\ell')$ is updated, the $Analysis(\ell'')$ needs to be recomputed. The iterations continue until the $worklist$ is empty meaning that the fixed point solution has been reached for the analysis at every program statement.

Fig. 4. Control Flow Graph of an Average Routine



We illustrate the proposed static-dynamic hybrid sensitivity analysis over an averaging routine for N numbers. The control flow graph of the program is shown in Figure 4. The superscripts of the program statements denote the labels associated with the statements for ease of illustration. The sensitivity of the program data N, a and sum are derived using dynamic analysis as per the initialization rules discussed in Section IV-A. Let the derived sensitivity of N, a and sum be $insensitive$ (I) by the dynamic analysis. The initial value of the analysis, σ_{init}, therefore assigns N, a and sum to I and the remaining data avg to $\perp$. The worklist initially contains all the edges $W = \{(1, 2), (2, 3), (2, 4), (3, 2), (4, 5)\}$. The MFP iterations and the analysis value at every entry point is shown in Table II. The table entries show the value of the analysis at the entry points of the program statements, i.e, the mapping σ from the program data to the elements of the *data sensitivity lattice*. In the first iteration, the edge $(1, 2)$ is removed from the $worklist$ and the transfer function corresponding to the assignment statement labeled by 1 ($sum = 0$) is applied. Since the rhs is a constant, there is no free variable and the sensitivity mapping remains unchanged at $Analysis(2)$. In the second iteration, the edge $(2, 3)$ is removed from the $worklist$ and the transfer function corresponding to the for statement, the identity function, is applied on $Analysis(2)$ resulting in no change. Similarly, edge $(2, 4)$ is removed from the $worklist$ in the third iteration and results in no change of values in $Analysis$. In the fourth iteration, the edge (3, 2) is removed and also results in no change of values in $Analysis$ since the lhs of the assignment statement labeled by 3 is sum which is already mapped as *insensitive* (I) by σ_{init} and we do not update mappings by σ_{init}. In the fifth iteration, the last remaining edge in the $worklist$, $(4, 5)$ is removed and the transfer function corresponding to the assignment statement labeled by 4 is applied on $Analysis(4)$. The free variables of the rhs are N and sum which are both mapped as *insensitive*(I) in $Analysis(4)$. Therefore, the transfer function $f(4)$ maps the lhs data avg as *insensitive*(I). The updated mapping σ is not $\sqsubseteq$ to the old $Analysis(5)$ since $\sigma(avg) = I$ is not $\sqsubseteq$ to the sensitivity of avg in $Analysis(5)$ which is $\perp$. Thus, $Analysis(5)$ is updated to $Analysis(5) \sqcup \sigma = \sigma$. This modified mapping is shown in $Analysis(5)$ in iteration 6 of the table. At this stage, the $worklist$ is empty and the algorithm terminates. Overall, we see that the analysis value remains the same as σ_{init} in all the iterations except the last.

V. IMPLEMENTATION

In our implementation, all the fault injection experiments on an application are conducted with a fixed input to the application. Our fault model is implemented with program binary instrumentation. We consider Java applications and perform bytecode instrumentation given the bytecode for a classfile and the application data where faults are to be injected using the Byte Code Engineering Library (BCEL) from Apache Commons [7]. In runtime, a Java Agent [1] intercepts the execution and replaces the Java bytecode for the class under inspection and injects a faulty value for the variable under test. We intend to emulate faults in memory considering random memory bit flip errors. Under such memory faults, a faulty value could be any random value in the range of the data. Therefore, to reflect memory bit flip errors in the experiments, the faulty value is generated from a uniform distribution in the

TABLE II
MFP ITERATIONS FOR $\mathcal{SA}$ OF AN AVERAGING ROUTINE

		Analysis value at entry points				
Iters	Worklist	1	2	3	4	5
1	{(1,2),. . ., (4,5)}	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$
2	{(2,3),. . ., (4,5)}	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$
3	{(2,4),. . ., (4,5)}	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$
4	{(3,2),(4,5)}	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$
5	{(4,5)}	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$
6	Empty	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to \bot$	$N \to I$, $a \to S$, $sum \to I$, $avg \to I$

interval of the range of the data under test. We consider the following kinds of program data in our implementation and instrument each as follows:

Fields: A Java class can have final/non-final, static/non-static data members. During instrumentation of a particular field variable, we set its initial value using BCEL API to a random value obtained from a uniform distribution and remove all instructions which change its value.

Method Parameters: We set the values for the method parameters at the beginning of the method and prevent all overwrites to its values in the successive instructions.

Method Local Variables: Each local variable in a method has a different scope of existence. Two or more variables with the same name can exist in a method. Hence they must be instrumented based on their scope. We track each write instruction for the variables within their scope and replace them with a faulty value.

Method Return Value: Each method can have different points of exit i.e. they can have multiple return statements. We instrument all those return statements, for methods having non-void return type, to return the faulty value.

The present bytecode instrumentation routine works only for data of elementary data types and array types. The instrumentation of class *constructor* members and objects has not been implemented in AutoSense in the current version. The dynamic analysis using SPRT is implemented keeping the probability of making Type I and Type II errors, α and β respectively, fixed to 0.01. The width of the indifference region $[p_1, p_0]$ is fixed to 2δ where $\delta = 0.01$. The static analysis is solved by implementing the Maximum Fixed Point (MFP) algorithm. The implementation derives the sensitivity of array type data as the join ($\sqcup$) of the sensitivity of its members.

VI. RESULTS

We evaluated AutoSense on applications which are known to be tolerant to approximations. AutoSense provides two types of tunable parameters. The first, is the analysis confidence probability θ used in the hypothesis of Eqn. 4. The second, is the QoS degradation tolerance threshold, γ, which provides a measure of how much approximation could be tolerated in the application's output. The value of γ depends on the QoS metric used in AutoSense which can be any error metric like the mean square error, normalized error, absolute error, root mean square error etc. The threshold γ specifies the maximum tolerable error.

QoS Metric

The QoS metric used in the evaluation of AutoSense is selected based on the application. For most of the applications that produce single or multiple numerical results, we considered *normalized mean error* as the QoS metric. The normalized mean error is given by the mean of the normalized error between all the output elements. A normalized error is given by the absolute difference between the actual and the approximate result but the difference is bounded by 1, i.e., if the absolute difference is greater than 1 then the normalized error is taken as 1. The QoS degradation tolerance threshold γ, considering normalized mean error as the metric, can be any specified value between 0 and 1. Setting $\gamma = 0$ specifies that no error is to be tolerated in the application output. In this case, AutoSense is going to identify program data which are highly insensitive (for e.g., dead variables). On the contrary, setting $\gamma = 1$ specifies that any error is acceptable and therefore all data can be identified as *insensitive*. However, AutoSense reports all loop counters and array indices as *sensitive* by default irrespective of any specified value of the parameters.

Array indices are considered as *sensitive* since an inaccurate value in the index may cause out of bound memory access. Although, loop counters can be insensitive in principle, they are conservatively assumed as *sensitive* by AutoSense. For example, an approximate value in a counter of an empty loop will be insensitive. However, it is unlikely to have programs with empty loops. In programs with non-empty loops, approximate loop counters can potentially introduce large error depending on the loop instructions. PSNR (Peak Signal to Noise Ratio) is the considered QoS metric for applications that render images. Some applications are evaluated without any error tolerance in the QoS. For example, in a barcode decoder application, an output is accepted only when the decoded barcode is correct. The value of γ with a strict QoS metric is marked as *exact* in the experiments table.

Applications for Evaluation

Applications in the Scimark 2.0 benchmark [18] are evaluated to test for performance and accuracy of analysis on floating point intensive numerical computations. The applications in the benchmark include kernels for Fast Fourier Transform (FFT), LU decomposition of a matrix (LU), Sparse Matrix Multiply (SMM), Successive Over Relaxation method to solve system of equations (SOR) and Monte Carlo method (MC). In addition to the Scimark benchmark, experimental results on a simple Raytracer [2], a 3D image renderer, Jmeint - a triangle intersection detector in 3D and ZXing, a barcode decoder [3] application for mobile devices running on Android OS are also shown to illustrate scalability of AutoSense. The benchmarks have been chosen same as in [25] to enable a comparison of their manual annotations.

A. Evaluation of Dynamic Sensitivity Analysis

The evaluation of the dynamic analysis by varying the parameters θ and γ is shown in Table III and Table IV. Table III shows the increase in the number of $insensitive$ data identifications by relaxing the QoS requirement and keeping the confidence measure fixed at $\theta = 0.5$. Similarly, Table IV shows the increase in identifications by relaxing the confidence measure and keeping the QoS degradation threshold fixed at $\gamma = 0.5$. Since there is a probability of accepting a false hypothesis in our analysis (though very small), the number of identified $insensitive$ data may vary over runs of AutoSense. Therefore, the reported results are taken as the median over three runs. We have verified that there is negligible change in the reported percentage of $insensitive$ data derivations over multiple runs of our framework. Observe that for the benchmarks LU, MC and SMM, the percentage of *insensitive* program data does not change by varying the parameters. This implies that the insensitive data of these applications are highly $insensitive$ and the other data are highly $sensitive$.

Table V shows the number of trials that the dynamic analysis performed for different confidence over some program data

TABLE III
THE PERCENT INSENSITIVE DATA REPORTED BY AUTOSENSE ON VARYING γ AND FIXED $\theta = 0.5$

QoS	Percent Data Derived Insensitive				
γ	FFT	SOR	MC	SMM	LU
0.010	0	0	33	15	11
0.025	0	0	33	15	11
0.050	0	0	33	15	11
0.075	0	0	33	15	11
0.1	0	0	33	15	11
0.2	0	13	33	15	11
0.3	0	27	33	15	11
0.4	7	27	33	15	11
0.5	7	27	33	15	11

TABLE IV
THE PERCENT INSENSITIVE DATA REPORTED BY AUTOSENSE ON VARYING θ AND FIXED QOS $\gamma = 0.5$ (SCIMARK2), PSNR=10.5 (RAYTRACER) AND EXACT (JMEINT)

$Conf.$	Percent Data Derived Insensitive						
θ	FFT	SOR	MC	SMM	LU	Raytracer	Jmeint
0.3	10	33	33	15	11	48	24
0.4	10	33	33	15	11	48	24
0.5	7	27	33	15	11	44	24
0.6	7	20	33	15	11	44	24
0.7	7	20	33	15	11	44	24
0.8	7	20	33	15	11	44	24
0.9	7	20	33	15	11	44	24

in the benchmark applications. The experiment uses the QoS tolerance parameter γ fixed at 0.5. A plot of the data is shown in Figure 5. For the hypothesis K defined in Eqn. 4 on a data under test, let the $Pr(K) = p$. It is known that the number of trials in SPRT depends on the distance of the parameter θ from p [30]. The number of trials tends to increase as θ gets closer to p and it decreases as θ gets farther away from p. Therefore, the plot in Figure 5 can give us an idea of p for the data under test. For example, the number of trials for the considered data in FFT and SOR is maximum when $\theta = 0.5$. We can therefore deduce that $Pr(K) \approx 0.5$. Similarly, we can deduce that $Pr(K) \approx 0.8$ for the considered data in the application Jmeint. In the applications MC, SMM, LU and Ray-tracer, the number of trials decreases on increasing θ from 0.3 to 0.9, which implies that the $Pr(K) \leq 0.3$ for the considered data in these applications. In the application Zxing, the number of trials increases on increasing θ from 0.3 to 0.9, which implies that $Pr(K) \geq 0.9$ for the considered data.

Note that in SOR, the dynamic analysis classified the data as *insensitive* for $\theta = 0.5$ and as *sensitive* for $\theta = 0.6$. This means that the hypothesis $H : Pr(K) < 0.5$ has failed (the contrary hypothesis $H' : Pr(K) \geq 0.5$ has passed) and the hypothesis $H : Pr(K) < 0.6$ has passed. This observation indicates that $0.5 \leq Pr(K) < 0.6$. Similarly in FFT, we get that $0.6 \leq Pr(K) < 0.7$. However, due to Type I and Type II errors in SPRT, $Pr(K)$ may not always be in the interval. In general, the confidence that $\theta_1 \leq Pr(K) < \theta_2$ is given by the probability $(1 - \alpha)(1 - \beta)$, when the contrary hypothesis $H' : Pr(K) \geq \theta_1$ and the null hypothesis $H : Pr(K) < \theta_2$ passes the test. Recall that α and β are the probability of making a Type I and Type II error in the test respectively. Since

[2] https://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=5590&lngWId=2

[3] https://github.com/zxing/zxing

the confidence of $Pr(K) \geq \theta_1$ and $Pr(K) < \theta_2$ is $(1-\alpha)$ and $(1-\beta)$ respectively, the confidence of $\theta_1 \leq Pr(K) < \theta_2$ is $(1-\alpha)(1-\beta)$.

Comparison with Earlier Work

Our proposed dynamic analysis is most similar to ASAC [22], which also proposes an automated sensitivity analysis of program data using a statistical method. We implemented the ASAC algorithm with a fault injection model that is presented in Section III-A. In order to obtain an interval to select a random instrumented value for fault injection on a data, [22] proposes performing a static range analysis. In our implementation, we do not perform a range analysis since our goal is to emulate memory bit flip errors which can result in any erroneous data in the datatype range and therefore, selecting fault values from an obtained range by static analysis is not justified. Table VI reports the percent data identified as *insensitive* by ASAC as compared to AutoSense for the Scimark 2.0 benchmark. The QoS metric used for the applications is *normalized mean error*, with QoS tolerance $\gamma = 0.5$. AutoSense derivations are with a confidence of $\theta = 0.5$. The algorithm in ASAC has two tunable parameters to qualitatively improve the confidence of the derivation. One of the parameters is denoted as the *discretisation constant* k and the other is denoted as the number of samples m. The derivation confidence increases with larger values of k and m. We choose $k = 30$ and $m = 100$ for our experiments which can be considered giving derivations with medium to high confidence. We observe that the performance of AutoSense is better than ASAC mostly because it requires fewer executions. We also observe that if there is a highly sensitive data in the set of data tested by ASAC, all the data are derived as sensitive. This is because, the experiments in ASAC are performed by introducing perturbations simultaneously in all the data and high sensitivity of a single data in the set results in all outcomes failing the QoS test. This is a limitation of ASAC. For example, 0% data is derived as *insensitive* in the application MC because we include a data $under_curve$ in the set of data to be tested by ASAC which is highly sensitive. Similar phenomenon is also seen in the application FFT.

Reliability Evaluation

To evaluate the reliability of our analysis, we compare our sensitivity classification with the manual annotations in the applications reported in [25] in which approximable data are annotated with the $@approx$ keyword. The authors of [25] mention that an approximate value in their manual annotations guarantees that the application does not crash and keeps a balance between reliability and energy saving. We observe the output of 1000 executions of the applications in the benchmark by injecting errors in all the manually annotated data and report the percentage of outputs that failed the QoS degradation threshold. Similarly, we perform 1000 executions of the benchmark applications by injecting errors in the AutoSense derived *insensitive* data with confidence $\theta = 0.3$ and $\theta = 0.5$ respectively and report the percentage of outputs failing the same QoS degradation threshold. The QoS threshold is fixed to $\gamma = 0.5$ for the QoS metric of normalized error, PSNR ≥ 10.5 for the QoS metric of PSNR. For two applications, there is no tolerance to QoS degradation. A comparison of the percentage of output failures is reported in Table VII. Column 5 (#Manual Annot.) shows the number of data in the application that is annotated as *approximable* in [25], while Column 6 (%Fail Manual) shows the percentage of output failures with inexactness in the manually annotated approximable data. Column 7 (%Data AS Tested) shows the percentage of data that is tested by AutoSense with hypothesis testing. The eighth and eleventh columns (%I) show the percentage of data in the application that is classified as *insensitive* by AutoSense with confidence $\theta = 0.3$ and $\theta = 0.5$ respectively. We see that 100% of the executions failed the QoS requirement for the applications MC, Ray-tracer, LU and Zxing when there is any inexact value in the manually annotated data. On the other hand, 100% of the executions passed the same QoS requirement when there is an inexact value in the AutoSense derived *insensitive* data with confidence $\theta = 0.5$ in MC, Ray-tracer and LU. However, 10% of the executions failed the QoS in Zxing. Considering that AutoSense performed the sensitivity classification with the confidence probability of $\theta = 0.5$, the number of executions that may fail the QoS should be less than 50% of the executions with inexact value in the *insensitive* data. In both the applications Zxing and Jmeint, the percentage of executions that failed with inexactness in AutoSense derived data is less than 50% (10% and 29% respectively). We also see that SOR, FFT, Jmeint and Zxing failed in 49%, 77%, 29% and 10% of the executions with inexact value in *insensitive* data derived with confidence $\theta = 0.3$. This shows the reduced reliability of the derivations with reduced confidence parameter value of θ. Note that a confidence $\theta = 0.3$ allows at most 70% failure of executions with inexactness in the derived *insensitive* data. The observed failure percentage is less than 70% except in FFT (77%). We believe that this is because of the classification of some *sensitive* data as *insensitive* due to Type I or Type II error. The percentage of *insensitive* data in a program gives a measure of its error resiliency. For example, we can deduce that the application *Ray-tracer* is the most error resilient of the applications tested with our analysis since it has 44% of *insensitive* data with confidence $\theta = 0.5$.

The reliability of AutoSense derivations is further illustrated on the application Ray-tracer, a 3D image renderer. Figure 6(b) shows the rendered image when inaccuracies are injected in all the data identified as *insensitive* by AutoSense with a confidence of $\theta = 0.5$. Figure 6(c) shows the rendered image when inaccuracy is injected in a manually annotated approximable data which AutoSense classified as *sensitive*. The perceived noise in Figure 6(c) is much more in comparison to Figure 6(b). This shows that inaccuracy in even one sensitive data may cause unacceptable program output.

B. *Evaluation of Static-Dynamic Combined Analysis*

We show the performance gain with our proposed static-dynamic combined sensitivity analysis as discussed in Section IV. Figure 7 shows the performance improvement on an average over 10 runs. The largest benchmark is approximately 15

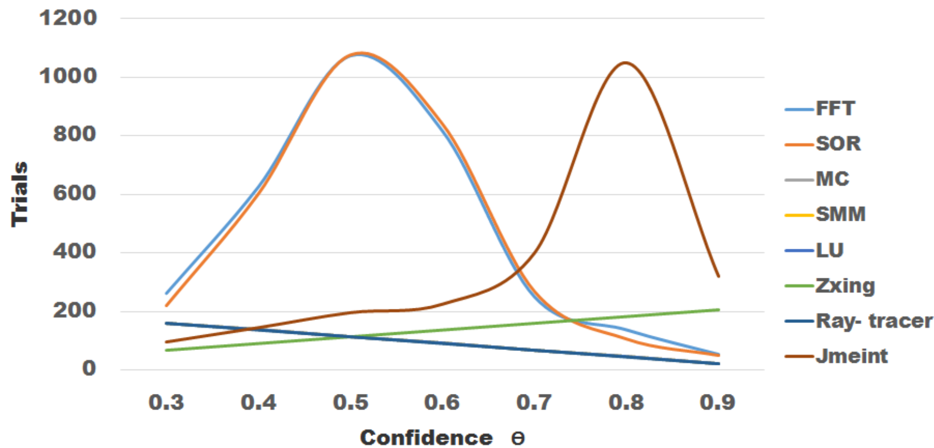
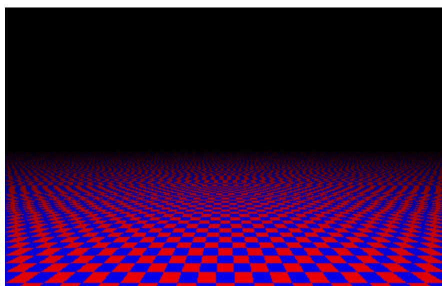


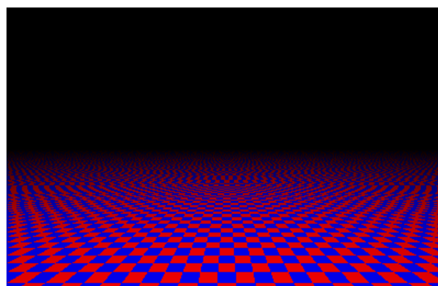
Fig. 5. Number of Trials vs. Confidence θ in SPRT

TABLE V
NUMBER OF TRIALS IN DYNAMIC ANALYSIS OF A SINGLE DATA BY VARYING CONFIDENCE

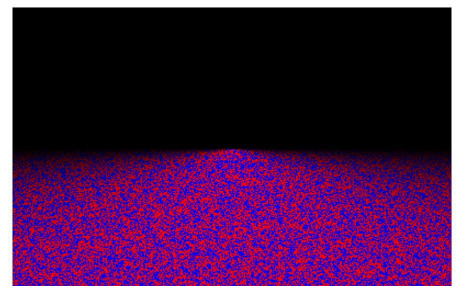
Application	Data	Number of Trials in SPRT						
		$\theta = 0.3$	$\theta = 0.4$	$\theta = 0.5$	$\theta = 0.6$	$\theta = 0.7$	$\theta = 0.8$	$\theta = 0.9$
FFT	n	261 (I)	626 (I)	1075 (I)	817 (I)	249 (S)	136 (S)	52 (S)
SOR	N	219 (I)	601 (I)	1075 (I)	839 (S)	267 (S)	105 (S)	49 (S)
MC	x	161 (S)	138 (S)	115 (S)	92 (S)	69 (S)	46 (S)	23 (S)
SMM	row	161 (S)	138 (S)	115 (S)	92 (S)	69 (S)	46 (S)	23 (S)
LU	Aii	161 (S)	138 (S)	115 (S)	92 (S)	69 (S)	46 (S)	23 (S)
Zxing	row	69 (I)	92 (I)	115 (I)	138 (I)	161 (I)	184 (I)	207 (I)
Ray-tracer	xe	161 (S)	138 (S)	115 (S)	92 (S)	69 (S)	46 (S)	23 (S)
Jmeint	xx	97 (I)	146 (I)	197 (I)	226 (I)	401 (I)	1050 (I)	321 (S)



(a) Original Image



(b) Rendered image with AutoSense guided approximation



(c) Rendered image with approximation in a manually annotated data

Fig. 6. Illustrating QoS reliability in Raytracer rendered image with AutoSense guided approximation

TABLE VI
PERFORMANCE COMPARISON OF ASAC AND AUTOSENSE

Application	LOC	%Data Tested	ASAC %I	ASAC Time (Sec)	AutoSense %I	AutoSense Time (Sec)
MC	22	100	0	44	33	1
SMM	29	38	22	76	40	18
SOR	36	60	33	92	27	27
FFT	119	62	0	163	7	22
LU	165	35	9	155	11	41

KLOC taking nearly 80 seconds with dynamic analysis alone. The analysis of Raytracer is most expensive with 110 seconds since it involves an expensive image rendering algorithm. The dynamic analysis in combination with the static sensitivity analysis of the Raytracer program takes 39 secs, though it fails to identify many $insensitive$ data. Table VIII shows that the gain in performance with the combined analysis is at the cost of precision and recall in some cases. The precision and recall are compared against the sensitivity derivations using the dynamic analysis with hypothesis testing. In the table, TP, FP and TN stands for *true positive*, *false positive* and *false negative* respectively. A *true positive* means that the classified *insensitive* data by $\mathcal{SA}$ is also classified as *insensitive* by the dynamic analysis. A *false positive* means that the classified *insensitive* data by $\mathcal{SA}$ is classified *sensitive* by the dynamic analysis and a *false negative* means that the classified *sensitive*

TABLE VII
PERCENT OUTPUT FAILING QOS WITH CONFIDENCE $\theta = 0.3$ AND $\theta = 0.5$

							$\theta = 0.3$			$\theta = 0.5$		
Application	QoS Metric	QoS (γ)	LOC	#Manual Annot.	%Fail Manual	%Data AS Tested	%I	Time (Sec)	%Fail	%I	Time (Sec)	%Fail
MC	Normalized Mean Error	0.5	22	2	100	100	33	1	0	33	1	0
SMM	Normalized Mean Error	0.5	29	4	0	54	15	18	0	15	18	0
SOR	Normalized Mean Error	0.5	36	3	0	60	33	27	49	27	27	0
RayTracer	PSNR	≥ 10.5	111	11	100	55	48	221	0	44	613	0
FFT	Normalized Mean Error	0.5	119	24	0	62	10	22	77	7	22	0
LU	Normalized Mean Error	0.5	165	22	100	36	11	41	0	11	41	0
Jmeint	Pass if correct, Fail otherwise	Exact	2694	109	28	50	24	947	29	24	1150	29
ZXing	Pass if correct, Fail otherwise	Exact	15785	115	100	80	35	1188	10	35	1435	10

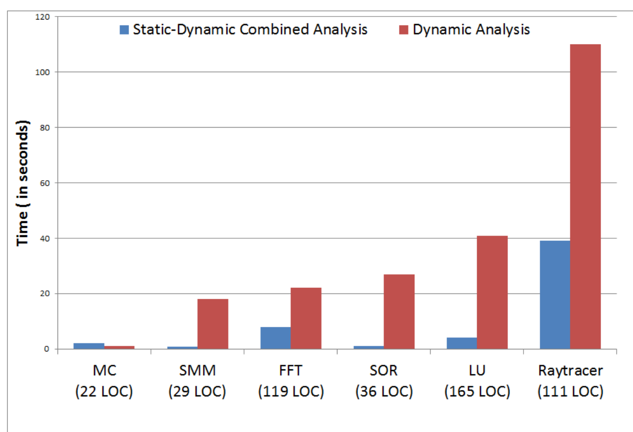


Fig. 7. Performance of Static-Dynamic Combined Analysis in Comparison to Dynamic Analysis

data by the combined analysis is classified as *insensitive* by the dynamic analysis. A precision is computed as $\frac{TP}{(TP+FP)} \times 100$ and the recall is computed as $\frac{TP}{(TP+FN)} \times 100$. Observe that the static-dynamic combined analysis displays 100% precision and 100%, 50% and 100% recall for SOR, MC and SMM respectively and completes much faster than the dynamic analysis alone. Note that the poor precision is not because of high *false positives* but because of low *true positives*. In essence, the combined analysis is efficient but misses to identify many *insensitive* data of a program. The precision of the combined analysis is expected to increase when the combination uses more of dynamic analysis and the performance is expected to increase when the combination uses more of static analysis.

TABLE VIII
PRECISION AND RECALL OF THE STATIC-DYNAMIC COMBINED ANALYSIS IN COMPARISON TO DYNAMIC ANALYSIS

Application	TP	FP	FN	Precision (%)	Recall (%)
FFT	0	0	3	0	0
SOR	3	0	0	100	100
MC	1	0	1	100	50
SMM	2	0	0	100	100
LU	0	0	9	0	0
Raytracer	0	1	2	0	0

VII. RELATED WORK

In this section, we discuss about relevant research in approximate computing related to partitioning program data and code as critical and resilient to the QoS. Loop perforation schemes have been proposed in [26] where critical loops and tunable loops are isolated. The tunable loops are perforated at runtime to gain energy efficiency but keeping the output acceptably accurate. The tunable loops are identified by profiling the application's accuracy on a set of representative inputs. Identifying critical input and program code regions automatically has been proposed as a framework, *Snap* using program executions [5]. A program is executed with a set of representative inputs and the baseline executions and traces are recorded. The program is then executed with fuzzed inputs and the behavior differences are compared with the baseline executions. The input and code regions are classified as either critical or non-critical using data mining techniques. *Snap* however, does not classify the program internal data. Automated partitioning of code segments as resilient and critical is also reported in a framework *ARC* [6]. A program is profiled and the inner loops in a program which contribute to more than a specified fraction of the total execution time (like 1%) are identified. These program hotspots are called the *kernels*. Errors are then injected in the data by binary code instrumentation which are modified inside a *kernel* and used outside it. If these instrumented errors cause programs to show unacceptable behavior, then the kernel is classified as critical and error resilient otherwise. This work also proposes methods to quantify the resiliency of the identified resilient kernels. Partitioning a computation into tasks and identifying the non critical ones such that discarding those does not distort the output beyond acceptable bounds has been proposed in [19]. The program executions are sampled randomly at various task failure rates to obtain a quantitative probabilistic model of the QoS failures versus task failures. Based on this model, resilient tasks are identified such that discarding those during a computation does not affect the accuracy of the output beyond a bound. Such task partitioning also allows computation in a distributed hardware platform with the critical tasks mapped to the reliable hardware components. Moreover, purposefully discarding non critical tasks also result in energy and time efficient computations with output distortions within acceptable bounds. Early phase termination at barrier synchronization

points in parallel computations has been proposed in [20], [21] where unfinished tasks are terminated early at the barrier synchronization points to prevent barrier idling. The effect of terminating tasks early to the computation result has been characterized using probabilistic distortion models. Patterns in computations which are tolerant to early phase terminations are also reported. A programming language extension to java, EnerJ that provides approximate data types has been proposed in [24]. In EnerJ, a programmer can annotate data types to be either precise or approximate. Such annotations allow direct mapping of approximate data to approximate memory storage devices by the compilation unit for energy efficiency. It also provides a type checker to control data flow from approximate to precise components of a program. The reliability of the approximate program depends on the annotations made by the programmer.The Green framework [3] generates approximate programs using a QoS model created by calibration. The program behavior under training input data set is monitored against approximations to create a QoS model. This model is used to guide the generation of an approximate program so that statistical bounds on the QoS requirements are met. Carbin et al. [4] present a programming language *Rely* that allows developers to reason about the quantitative reliability of an application, namely the probability that it produces the correct result when executed on unreliable hardware. They also present a *static quantitative reliability analysis* that verifies quantitative requirements on the reliability of an application. The analysis takes a *Rely* program with reliability specification and a hardware specification that characterizes the reliability of the underlying hardware components and verifies that the program satisfies its reliability specification when executed on unreliable hardware platform. Our proposed work is most similar to [22] which also proposes an automatic sensitivity analysis of program data using statistical methods. Sensitivity analysis is performed by systematically testing the program output in the presence of injected distortions in the data. The accuracy of a pre-determined number of fault injected program executions are tested against the exact result. Based on the observations, the sensitivity is deduced using a statistical test called the K-S test. The sensitivity results reported, however, have no reliability guarantee. In addition, the number of program executions in the presence of perturbations required for performing the statistical test is chosen in an adhoc manner. The number of samples chosen influences the reliability and efficiency of the results. The proposed work in this paper not only provides probabilistic guarantee on the sensitivity classification but also requires an optimal number of program executions for a desired confidence of the analysis. Moreover, a dynamic-static hybrid sensitivity analysis is also proposed with an improvement in the efficiency.

VIII. Conclusion

Identifying insensitive error resilient data of an application is non-trivial, especially when the application is large and has substantial data and control dependencies. Manual annotation of data may not be reliable and may result in unacceptable output even when one data is mis-annotated as insensitive / resilient in approximation aware programming languages like EnerJ. We illustrated that a systematic study of the effect of inaccuracy in program data with statistical methods like hypothesis testing can lead to automatic classification of insensitive and sensitive data. Moreover, probabilistic reliability guarantee is provided on the classification. A combination of static and dynamic sensitivity analysis is also proposed that is efficient, especially for applications running compute expensive algorithms, but results in many false negatives compared to the dynamic analysis. However, 100 % precision with the combined analysis could be achieved for some of the benchmarks. The proposed methods are implemented as a tool - AutoSense. Initial results look promising and we believe, will open up many interesting research directions.

References

[1] Java instrumentation library. *URL http://docs.oracle.com/javase/7/docs/api/java/lang/in-strument/package-summary.html*.

[2] A. Askarov and A. Myers. A semantic framework for declassification and endorsement. ESOP'10, pages 64–84, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Not.*, 45(6):198–209, June 2010.

[4] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *ACM SIGPLAN Notices*, volume 48, pages 33–52. ACM, 2013.

[5] M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *Proceedings of ISSTA'10*, pages 37–48. ACM, 2010.

[6] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 113:1–113:9. ACM, 2013.

[7] A. Commons. Bcel: Byte code engineering library. *URL https://commons.apache.org/proper/commons-bcel/index.html*.

[8] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO 2012, Canada, December 1-5, 2012*, pages 449–460. IEEE Computer Society, 2012.

[9] E. Kreyszig. *Advanced Engineering Mathematics*. John Wiley & Sons, INC., USA, 2006.

[10] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO'04)*, Palo Alto, California, Mar 2004.

[11] A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *RV 2010*, pages 122–135, 2010.

[12] E. L. Lehmann and J. P. Romano. *Testing Statistical Hypotheses*. Springer-Verlag New York, 2005.

[13] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman. LLFI: an intermediate code-level fault injection tool for hardware faults. In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*, pages 11–16. IEEE, 2015.

[14] G. E. Moore. Readings in computer architecture. chapter Cramming More Components Onto Integrated Circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[15] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[16] K. Palem and A. Lingamneni. What to do about the end of moore's law, probably! In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 924–929, New York, NY, USA, 2012. ACM.

[17] K. V. Palem. What exactly is inexact computation good for? In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 1. ACM, 2014.

[18] R. Pozo and B. Miller. Scimark 2.0.

[19] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of ICS'06*, ICS '06, pages 324–334, New York, NY, USA, 2006. ACM.

[20] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. *SIGPLAN Not.*, 42(10):369–386, Oct. 2007.

[21] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 369–386, New York, NY, USA, 2007. ACM.

[22] P. Roy, R. Ray, C. Wang, and W. F. Wong. Asac: Automatic sensitivity analysis for approximate computing. *SIGPLAN Not.*, 49(5):95–104, June 2014.

[23] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. Halsted Press, New York, NY, USA, 2004.

[24] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164–174, June 2011.

[25] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[26] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of FSE'11*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[27] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan. Approximate computing and the quest for computing efficiency. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 120:1–120:6. ACM, 2015.

[28] A. Wald. Sequential tests of statistical hypotheses. *Ann. Math. Statist.*, 16(2):117–186, 06 1945.

[29] E. W. Weisstein. Hypothesis testing. from mathworld–a wolfram web resource. *URL http://mathworld.wolfram.com/HypothesisTesting.html*.

[30] H. L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania., 2005.

[31] H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *In Proc. of CAV'02, volume 2404 of LNCS*, pages 223–235. Springer, 2002.

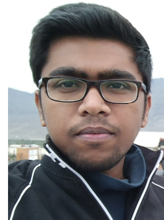
Acknowledgement

This work was supported in part by National Institute of Technology Meghalaya and Visvesvaraya Ph.D. Scheme, GoI.

Author Biography

Bernard Nongpoh is a research scholar at the department of Computer Science and Engineering, National Institute of Technology Meghalaya. He holds a masters degree in Information Technology from Tezpur University Assam and a bachelor's degree in Information Technology from North-Eastern Hill University Shillong. His research interests are in the area of approximate computing and its applications in system design.

Rajarshi Ray is an Assistant Professor at the department of Computer Science and Engineering, National Institute of Technology Meghalaya. He received his B.Sc. in Computer Science from University of Pune in 2005 and M.Sc. in Computer Science from the Chennai Mathematical Institute in 2007. He received his Ph.D. in Computer Science from Verimag, University of Grenoble in 2012. His research interests are in formal methods in systems verification, cyber-physical systems and approximate computing.



Saikat Dutta is a Software Developer in the Bing and Cortana Team at Microsoft India Development Centre, Hyderabad. He earned his bachelor's degree from Jadavpur University, Kolkata in 2015. His research interests include software verification, software testing and energy aware computing.

Ansuman Banerjee is an Associate Professor at the Advanced Computing and Microelectronics Unit, Indian Statistical Institute Kolkata. He received his B.E. from Jadavpur University, and M.S. and Ph.D. degrees from the Indian Institute of Technology Kharagpur, all in Computer Science. His research interests are in formal methods and their applications to systems design and verification.