

Argus: Online Statistical Bug Detection^{*}

Long Fei¹, Kyungwoo Lee¹, Fei Li², and Samuel P. Midkiff¹

¹ School of Electrical and Computer Engineering

² School of Industrial Engineering,
Purdue University,

West Lafayette, IN 47907, USA

{lfei, kwlee, li64, smidkiff}@purdue.edu

Abstract. Statistical debugging is a powerful technique for identifying bugs that do not violate programming rules or program invariants. Previously known statistical debugging techniques are offline bug isolation (or localization) techniques. In these techniques, the program dumps data during its execution, which is used by offline statistical analysis to discover differences in passing and failing executions. The differences identify potential bug sites. Offline techniques suffer from three limitations: (i) a large number of executions are needed to provide data, (ii) each execution must be labelled as passing or failing, and (iii) they are post-mortem techniques and therefore cannot raise an alert at runtime when a bug symptom occurs. In this paper, we present an *online statistical bug detection* tool called Argus. Argus constructs statistics at runtime using a *sliding window* over the program execution, is capable of detecting bugs in a single execution and can raise an alert at runtime when bug symptoms occur. Moreover, it eliminates the requirement for labelling all executions as passing or failing. We present experimental results using the Siemens bug benchmark showing that Argus is effective in detecting 102 out of 130 bugs. We introduce optimization techniques that greatly improve Argus' detection power and control the false alarm rate when a small number of executions are available. Argus generates more precise bug reports than the best known bug localization techniques.

1 Introduction

Program correctness and reliability are two of the greatest problems facing the developers, deployers and users of software. The financial implications are tremendous – a recent NIST report estimates that \$59.6 billion dollars a year, or 0.6% of the GDP, are lost every year because of software errors [17]. Single failures of software can disable businesses like Charles Schwab and eBay for hours or days, costing millions of dollars in revenues. Moreover, purely malicious and politically motivated sociopaths exploit software errors to disrupt civil society. Simultaneous with the increasing risks of bug-ridden code has been the rise in

^{*} This work was supported by the National Science Foundation by grants 0325603-CCR and 0313033-CCR, and by the Indiana Utility Regulatory Commission. The opinions expressed are those of the authors, and not the National Science Foundation.

size and complexity of software, which makes software debugging an increasingly challenging task. In order to relieve the programmers from laborious debugging, automated debugging techniques have been extensively studied. Among these, *bug detection* techniques focus on determining the existence/nonexistence of a bug; *bug isolation/localization* techniques focus on determining the cause of observed bug symptoms. In both cases, an effective bug report that distinguishes the potential bug site(s) (or their vicinities) will greatly expedite the debugging process. While the effectiveness of a bug isolation tool is typically measured by its precision in locating bug sites, the effectiveness of a bug detection tool is measured by three factors: the detection rate, the false alarm rate, and the precision with which it locates bug sites. The detection rate reveals the power of the tool in determining a bug has occurred; the false alarm rate measures the likelihood of raising an alarm when there is no bug; and the precision in locating bug sites measures the quality of a bug report generated by the tool.

Statistical debugging techniques have recently caught the attention of the debugging research community. These techniques isolate software bugs by comparing the data collected from a large number of passing and failing executions using statistical methods (e.g. logistic regression [13], conditional probability [14], and hypothesis testing [15]). Because these techniques identify differences in runtime information that is correlated with program behaviors, they have the potential to isolate software bugs that do not violate programming rules or program invariants – the bugs that are most difficult to isolate using traditional debugging techniques.

However, existing statistical debugging techniques share some drawbacks. First, they require a large number of passing and failing executions to facilitate offline statistical analysis. While the passing executions may be obtained from in-house regression testing before software release, a large number of failing executions are usually unavailable. This requirement has three consequences in reality. First, the debugging process relies on the Internet to collect data from a large number of executions, which restricts the applicability of these techniques in devices with limited connectivity and increases the complexity of the software. Second, collecting execution data from customers raises the question of sensitive information leaking, which restricts the applicability of these techniques in sensitive environments. Third, the software vendor cannot respond timely to a product failure until it accumulates enough failures. This contradicts the ethics of debugging in that the customers have to suffer many more failures (and the resulting damage) even though the software vendor is well aware of the existence of a bug after the first observed failure!

Second, existing statistical debugging techniques require labelling passing and failing executions. This requirement has two consequences. First, it is expensive to construct test cases and their corresponding expected results, and requires extensive domain knowledge. Second, it is infeasible to apply these techniques to applications where the correctness of an outcome can not be easily verified or the expected outcome is unknown *a priori*. Although a program crash is a typical symptom of failure, many failures are non-crashing.

Third, existing statistical debugging techniques are *postmortem* techniques, i.e. they are bug isolation/localization techniques, not bug detection techniques. They cannot raise an alert when bug symptoms occur. An alarm at the time of failure can give the user more time to respond to the failure and to reduce the damage caused by the failure. This is particularly important in mission-critical programs where an incorrect operation can initiate a chain of catastrophic consequences.

In this paper, we present an online statistical *bug detection* technique, called Argus. Argus detects software bugs at runtime using an *anomaly detection* approach. Argus overcomes the three drawbacks mentioned above by constructing runtime statistics on a *sliding window* over an execution rather than the whole execution, which makes it possible to accumulate multiple observations in a single execution. Argus runs in two modes: training and detection. In the training mode, Argus builds statistical models from one or multiple passing executions. In the detection mode, Argus can detect bugs in a single execution. Argus does not require labelling of passing and failing executions. Argus can sound an alarm at runtime when bug symptoms occur. Our experiments show that Argus is effective in detecting 102 out of 130 bugs in the Siemens bug benchmark. We also introduce optimization techniques to greatly improve Argus' bug detection power and achieve a low false alarm rate when a small number of detection executions are available. Argus generates a precise bug report that outperforms the best known bug localization techniques on the Siemens benchmark. Our experiments on the SPEC2000 benchmark show that Argus runtime overhead is comparable to that of many existing successful tools.

We make the following contributions:

- We propose the first online statistical bug detection technique that can detect bugs with a single execution. We present its design and implementation.
- We propose mathematically-rigorous formulas and their implementations to maximize the detection rate, to minimize the false alarm rate, and to minimize the number of executions required to meet a given detection rate and a given false alarm rate when multiple executions are available.
- We present the experimental results using the Siemens bug benchmark and SPEC2000 showing the effectiveness of our technique.

Supplemental discussions and code used in this paper can be found on the Argus web site (<https://engineering.purdue.edu/Apollo/research/argus>).

2 Related Work

Existing runtime bug detection techniques can be placed into two categories [25]: *programming-rule based* and *statistical-rule based*. Tools in the former category (e.g. Purify [10], SafeC [1], and runtime assertion) check runtime violations against programming language specifications, programming paradigms, or user-specified program-specific rules (e.g. dereferencing a NULL pointer or memory leak); tools in the latter category check runtime violations against program invariants (e.g. value invariants [7, 8], PC invariants [25]). Runtime statistical-rule

based checking differs from Argus in that the judgement in runtime statistical-rule based checking is based on whether a single sample violates an existing invariant, while the judgement of Argus is based on whether a runtime *statistic* computed from *an aggregation of samples* fits an existing statistical model.

Recently several statistical bug isolation/localization tools have been developed. Earlier work by Burnell et al. use a Bayesian belief network as a supplement to program slicing to find the root of a bug by analyzing a program dumpfile [2]. State of the art techniques collect runtime information from a large number of executions, and apply statistical techniques in offline data mining to discover the differences between passing and failing executions. Liblit et al. use logistic regression to select suspicious value predicates associated with a bug [13], and later use a technique based on probabilistic correlations of value predicates and program crash [14]. Liu et al. propose a technique based on hypothesis testing in [15].

Statistical methods have been used for other debugging problems. Dickinson et al. find program failures through clustering program execution profiles [6]. Podgurski et al. use logistic regression to select features and cluster failure reports on selected features [18]. Cluster results are shown to be useful in prioritizing software bugs.

Sekar et al. detect anomalous program behavior by tracing system calls using an FSA-based approach [22]. Dallmeier et al. propose a postmortem defect localization technique for Java based on the differences in frequencies of certain class method calling sequences showing up in passing and failing runs [5]. A sliding window is used to divide the calling sequences into subsequences for efficiency in comparison.

3 Argus Statistical Model

Argus detects bugs at runtime using an *anomaly detection* approach. Argus collects samples of runtime statistics characterizing program's runtime behavior. In the training run(s), samples of these runtime statistics are used to build statistical models. In the detection run, samples of these runtime statistics are tested against the learned statistical models. If the samples significantly deviate from the learned statistical models, an alarm is raised.

3.1 Extended Finite State Automaton

We use an extended finite-state-automaton (ext-FSA) to characterize the program's runtime control-flow behavior. In our ext-FSA, each state is a *runtime event*, and each transition is a transition from one runtime event to the next runtime event. Runtime events are defined to be the set of `{program start, program exit, procedure entrance, procedure return, loop start, loop exit, compound statement entrance, compound statement exit}`. Each transition is augmented with the distribution of transition frequency (Figure 1(a)). We assume that anomalous transition patterns are indicators of buggy behaviors.

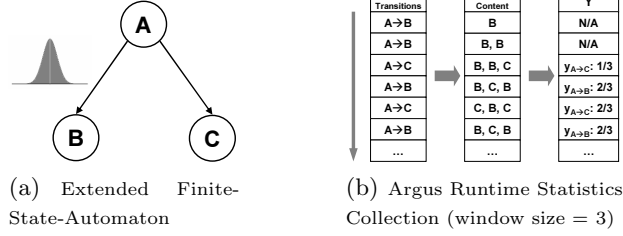


Fig. 1. Argus Statistical Model

3.2 Modelling State Transition Statistics

Using the states in Figure 1(a) for example, we define random variable $X_{A \rightarrow B}$ to be: $X_{A \rightarrow B} = 1$ if transition $A \rightarrow B$ takes place; and $X_{A \rightarrow B} = 0$ if transition $A \rightarrow C$ takes place, where C is a state other than B . The random variable $X_{A \rightarrow B}$ has an unknown distribution $\Theta_{X, A \rightarrow B}$. Let $\{X_{A \rightarrow B, i}\}$ be an *independent and identically distributed (i.i.d)* sample from the distribution. According to Central Limit Theorem [3], the following statistic

$$Y_{A \rightarrow B} = (\sum_{i=1}^n X_{A \rightarrow B, i})/n \quad (1)$$

conforms to a normal distribution $N(\mu_{Y_{A \rightarrow B}}, \sigma_{Y_{A \rightarrow B}}^2)$ as $n \rightarrow \infty$, with mean $\mu_{Y_{A \rightarrow B}}$ and variance $\sigma_{Y_{A \rightarrow B}}^2$. Intuitively, $X_{A \rightarrow B}$'s distribution is dictated by the conditional probability: $Pr\{transition A \rightarrow B | state A\}$; and $Y_{A \rightarrow B}$ is the *frequency* of transition $A \rightarrow B$ being taken in the past n accesses to A , which is an estimator of the conditional probability: $Pr\{transition A \rightarrow B | state A\}$ based on past n accesses to A .

To determine if an observation $y_{A \rightarrow B}$ comes from an existing model $\Theta_{Y, A \rightarrow B}$ (in our case $N(\mu_{Y_{A \rightarrow B}}, \sigma_{Y_{A \rightarrow B}}^2)$), we adopt a *hypothesis testing* [3] approach. Our *null hypothesis* H_0 is: $y_{A \rightarrow B}$ belongs to $N(\mu_{Y_{A \rightarrow B}}, \sigma_{Y_{A \rightarrow B}}^2)$. Let the *probability density function (pdf)* of $Y_{A \rightarrow B}$ be $f_{Y_{A \rightarrow B}}(y|\theta_{Y_{A \rightarrow B}})$, where $\theta_{Y_{A \rightarrow B}}$ denotes the parameters of the normal distribution. The *likelihood function* of observation $y_{A \rightarrow B}$ coming from $N(\mu_{Y_{A \rightarrow B}}, \sigma_{Y_{A \rightarrow B}}^2)$ is

$$L(\theta_{A \rightarrow B} | y_{A \rightarrow B}) = f_{Y_{A \rightarrow B}}(y_{A \rightarrow B} | \theta_{Y_{A \rightarrow B}}) \quad (2)$$

We reject the null hypothesis if the likelihood is smaller than a threshold, which indicates that $y_{A \rightarrow B}$ is significantly different from the existing model.

We make several assumptions to make this method practical for bug detection. First, we assume that $Y_{A \rightarrow B}$ computed from a window containing K consecutive $X_{A \rightarrow B}$'s (when $n = K$ in Equation 1) approximately conforms to a normal distribution, where K is a system parameter (window size). Second, we assume $Y_{A \rightarrow B}$'s we compute from different windows are *i.i.d.* samples from $Y_{A \rightarrow B}$'s underlying distribution. Our experiments show that these assumptions are reasonable approximations of the theoretical properties.

In order to perform runtime bug detection, Argus computes runtime statistics using Equation 1 on a *sliding window* of size K rather than the full execution

trace. For each state A , let S_A be the set of states that are possible targets of transitions from A . At runtime, if we observe transition $A \rightarrow B$, where $B \in S_A$, we compute $y_{A \rightarrow B}$ by applying Equation 1 on the last K transitions from A . Figure 1(b) shows how Argus computes y 's at runtime. In this example, K is set to 3. A window size of 10 works well in our experiments.

We augment each transition $A \rightarrow B$ in our ext-FSA with a distribution of the corresponding statistic $Y_{A \rightarrow B}$. Mean and variance of the distribution are estimated from samples of the statistic $\{y_{A \rightarrow B, i}\}$ collected in training executions:

$$\mu_{Y_{A \rightarrow B}} = (\sum_{i=1}^N y_{A \rightarrow B, i}) / N \quad (3)$$

and

$$\sigma_{Y_{A \rightarrow B}}^2 = \frac{1}{N-1} \sum_{i=1}^N (y_{A \rightarrow B, i} - \mu_{Y_{A \rightarrow B}})^2 \quad (4)$$

where N is the number of samples. During a detection run, each $y_{A \rightarrow B}$ is tested against the null hypothesis. For simplicity, we test the normalized statistic:

$$Z_{A \rightarrow B} = (Y_{A \rightarrow B} - \mu_{Y_{A \rightarrow B}}) / \sigma_{Y_{A \rightarrow B}} \quad (5)$$

against the null hypothesis " H_0 : $z_{A \rightarrow B}$ belongs to $N(0, 1)$ "¹. The corresponding pdf of this normalized statistic is $f(z) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$. For computational efficiency, we measure the *log likelihood* and ignore the constant coefficient $\frac{1}{\sqrt{2\pi}}$:

$$\log L(\theta|z) = -z^2/2 \quad (6)$$

If the log likelihood falls below a threshold t , an alarm is sounded to indicate the possible existence of a bug.

For each state A , frequencies of transitions to different states in S_A are usually *inversely correlated*. For example, in Figure 1(a), large $Y_{A \rightarrow B}$ leads to small $Y_{A \rightarrow C}$, and vice versa. Therefore, Argus tests the log likelihood of $y_{A \rightarrow B}$ only when $y_{A \rightarrow B} > \mu_{Y_{A \rightarrow B}}$, where $\mu_{Y_{A \rightarrow B}}$ is the mean of the learned distribution. The underlying strategy is that anomalously small $y_{A \rightarrow B}$ values will be reflected by an anomalously large $y_{A \rightarrow C}$ and thus Argus only needs to sound one alarm.

3.3 Alarm Threshold

Given a program P , Argus can be considered as a *binary function* of an execution e and the log likelihood threshold t . $Argus_P(e, t)$ yields 1 if Argus raises alarm on e , and 0 otherwise.

Given a set of failing executions $\{e_i\}_{i=1..m}$, and a log likelihood threshold t , we define the *detection rate* of Argus to be:

$$D(t) = (\sum_{i=1}^m Argus_P(e_i, t)) / m \quad (7)$$

¹ For simplicity and computational efficiency, here we use a normal distribution to approximate Z 's distribution (t-distribution).

Intuitively, $D(t)$ is the fraction of buggy executions that Argus raises an alarm. Accordingly the *false alarm rate* given a set of passing executions $\{e'_i\}_{i=1\dots n}$ and t is defined to be the fraction of normal executions that Argus raises alarms:

$$F(t) = (\sum_{i=1}^n \text{Argus}_P(e'_i, t)) / n \quad (8)$$

Intuitively, the larger the threshold t is, the larger $D(t)$ and $F(t)$ are. Therefore, the detection power of Argus is largely constrained by the number of false alarms the user is willing to tolerate. Let f be the false alarm limit, we have $t = F^{-1}(f)$, and the corresponding detection rate is a function of f :

$$D'(f) = D(F^{-1}(f)) \quad (9)$$

In Argus, t is a tunable system parameter. F can be found by measuring the false alarm rates with different t 's using the passing executions from the test cases before software release. This allows the user to tune Argus based on the false alarm rate he/she is willing to tolerate.

3.4 Optimizing the Effectiveness over Multiple Executions

Although Argus is capable of detecting bugs using a single detection execution, Argus' power is *not* confined to a single execution. Argus can benefit from multiple detection executions to improve the overall detection rate and to reduce the overall false alarm rate. Now the question is: if multiple detection executions are available, how can a user *maximize the overall detection rate*, *minimize the overall false alarm rate*, and *minimize the number of detection runs needed*?

Let $\{f_i\}_{i=1\dots m}$ be the set of different false alarm rates (sorted in ascending order), with f_m being the upper bound on the false alarm rate the user is willing to tolerate. The corresponding detection rates are $\{d_i\}_{i=1\dots m}$. Let R denote the total number of detection executions, and r_i denote the number of detection executions under false alarm rate f_i , $\sum_{i=1}^m r_i = R$. We have the *overall miss rate*:

$$M(\{d_i\}, \{r_i\}) = \prod_{i=1}^m (1 - d_i)^{r_i} \quad (10)$$

which is the joint probability of the bug going undetected in all the detection executions. Correspondingly, the *overall detection rate* is:

$$D(\{d_i\}, \{r_i\}) = 1 - M(\{d_i\}, \{r_i\}) = 1 - \prod_{i=1}^m (1 - d_i)^{r_i} \quad (11)$$

Although actual $\{d_i\}$ are not known *a priori*, they can be estimated by running Argus against bugs found and fixed in the testing phase.

Problem 1: Given $\{f_i\}$, $\{d_i\}$, R , and overall false alarm rate limit f , how are values for $\{r_i\}$ chosen to maximize the overall detection rate?

Rather than solving this problem, we attack the dual problem instead: how do we minimize the overall miss rate. This problem can be described as the following *linear programming* problem:

$$\begin{aligned} & \text{minimize: } \sum_{i=1}^m r_i \log(1 - d_i) \\ & \text{subject to: } \sum_{i=1}^m r_i = R, 0 \leq r_i \leq R \\ & \sum_{i=1}^m f_i \times r_i \leq f \times R \end{aligned} \quad (12)$$

Here we take the \log of $M(\{d_i\}, \{r_i\})$ to make it linear. The constraints are: the total number of executions under different false alarm limits is R , and the total number of false alarms under different false alarm limits must be smaller than the total number of false alarms allowed. The optimal detection rate is a function of $\{f_i\}$, $\{d_i\}$, f , and R under the constraints:

$$D_{opt}(\{f_i\}, \{d_i\}, f, R) = \underset{\{r_i\}}{\operatorname{argmax}}(1 - \exp(\sum_{i=1}^m r_i \log(1 - d_i))) \quad (13)$$

which uses the result from Equation 12.

Problem 2: Given $\{f_i\}$, $\{d_i\}$, R , and overall detection rate requirement d , how do we choose values for $\{r_i\}$ to minimize the overall false alarm rate?

The corresponding linear programming problem is:

$$\begin{aligned} & \text{minimize: } \sum_{i=1}^m f_i \times r_i \\ & \text{subject to: } \sum_{i=1}^m r_i = R, 0 \leq r_i \leq R \\ & \sum_{i=1}^m r_i \log(1 - d_i) \leq \log(1 - d) \end{aligned} \quad (14)$$

Here we minimize the total number of false alarms under the constraints the total number of executions is R , and the overall miss rate must be smaller than $(1 - d)$. The optimal false alarm rate is a function of $\{f_i\}$, $\{d_i\}$, R , and d under the constraints:

$$F_{opt}(\{f_i\}, \{d_i\}, R, d) = \underset{\{r_i\}}{\operatorname{argmin}}\left(\frac{1}{R} \sum_{i=1}^m f_i \times r_i\right) \quad (15)$$

which uses the result from Equation 14.

Problem 3: Given $\{f_i\}$, $\{d_i\}$, the overall false alarm rate limit f and the overall detection rate requirement d , how many detection runs are sufficient?

The corresponding linear programming problem is:

$$\begin{aligned} & \text{minimize: } \sum_{i=1}^m r_i \\ & \text{subject to: } 0 \leq r_i, i = 1 \dots m \\ & \sum_{i=1}^m f_i \times r_i \leq f \times \sum_{i=1}^m r_i \\ & \sum_{i=1}^m r_i \log(1 - d_i) \leq \log(1 - d) \end{aligned} \quad (16)$$

where the constraints are: total number of false alarms must be smaller than the number of false alarms allowed, and the overall miss rate must be smaller than $(1 - d)$. The optimal number of executions needed is a function of $\{f_i\}$, $\{d_i\}$, f , and d under the constraints:

$$R_{opt}(\{f_i\}, \{d_i\}, f, d) = \underset{\{r_i\}}{\operatorname{argmin}}(\sum_{i=1}^m r_i) \quad (17)$$

which uses the result from Equation 16.

Equations 12, 14, and 16 are easily solvable using linear programming tools. Sample code written in GAMS (<http://www.gams.com>) can be found on the Argus web site.

4 Argus Design and Implementation

Argus is implemented as a runtime library for C. We use the `Cetus C` compiler [12] to instrument the program source with calls to the Argus runtime library. `Cetus` instruments the following program points: program start, program exit, procedure entrance, procedure return, loop entrance, loop exit, compound statement entrance, and compound statement exit. Argus monitors the transitions of these events, and computes the runtime statistic y for each transition using Equation 1. Argus intercepts crashing signals (`SIGSEGV`, `SIGABRT`, and `SIGTERM`), and monitors event transitions even at a program crash.

If Argus is running in training mode, it also estimates the distribution parameters for each transition. Because Equations 3 and 4 require storing every y (sample of the runtime statistic Y) computed using Equation 1 at runtime, they are not scalable. Instead, we estimate the distribution parameters *recursively* using the following equations [24]:

$$\mu_t = \frac{t-1}{t}\mu_{t-1} + \frac{1}{t}y_t \quad (18)$$

$$\sigma_t^2 = \frac{t-1}{t}\sigma_{t-1}^2 + \frac{1}{t-1}(y_t - \mu_t)^2 \quad (19)$$

The estimated distribution parameters are dumped into a transition distribution profile at the end of each training execution. The profile is then loaded at the initialization phase of a detection execution.

If Argus is running in detection mode, it performs likelihood test for each y computed at runtime using Equations 5 and 6. A bug report is generated at the end of each detection run. The bug report contains the top k most suspicious transitions (transitions with lowest log likelihood values), where k is a configurable parameter.

5 Experiments

In this section, we evaluate the effectiveness of Argus at detecting bugs, its runtime overhead, and the profile size. To test Argus' bug detection power, we apply Argus to the Siemens bug benchmark [11]. To measure the cost of applying Argus to real world applications, we use the SPEC2000 benchmark to measure Argus' runtime overhead and profile size.

The Siemens benchmark was originally used by Siemens Corporation Research to study test adequacy criteria [11]. The siemens benchmark contains seven small programs: `print_tokens`, `print_tokens2`, `replace`, `schedule`, `schedule2`, `tcas`, and `tot_info`. Each program has multiple versions. Each version is injected with one bug. Altogether there are 130 faulty versions, simulating a wide variety of realistic bugs. A number of previous works in the field of bug isolation have reported their results using this benchmark, including [4, 9, 15, 19, 20, 21].

5.1 Bug Detection Power

We measure the bug detection power of Argus on the Siemens benchmark. Our goal is to reveal the average detection rate and the effectiveness of Argus with regard to given false alarm limits. We consider Argus *effective* in detecting bugs in a faulty program if the alarm raised by Argus is more likely to be a true alarm rather than a false alarm. We define the *effective function* on a given faulty program P as (using ternary expression semantics):

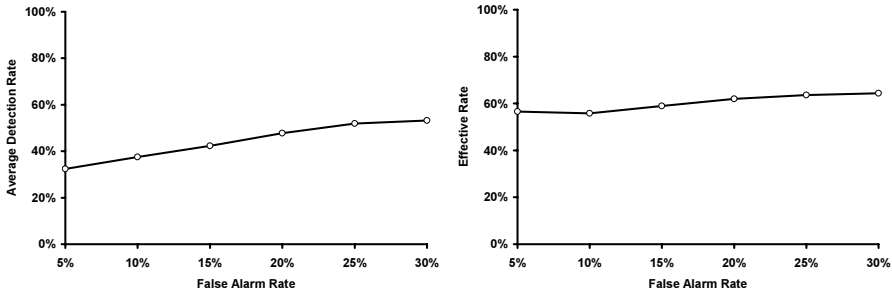
$$E_P(f) = (D'(f) > f ? 1 : 0) \quad (20)$$

where $D'()$ is defined in Equation 9. That is, given a false alarm rate f , if the corresponding detection rate is larger than f , we consider Argus effective. Given a set of faulty programs $\{P_i\}_{i=1\dots m}$, and a false alarm rate f , the *effective rate* is defined as:

$$ER_{\{P_i\}}(f) = \frac{1}{m} \sum_{i=1}^m E_{P_i}(f) \quad (21)$$

Intuitively, the effective rate measures the overall effectiveness of Argus on a set of faulty programs. Specifically, our goal is to establish Equation 9 and Equation 21. We use 10 as our sliding window size.

Each program in the Siemens benchmark has a bug-free version. We partition the test cases randomly with a 9 : 1 ratio as follows. We apply Argus in training mode on the bug-free version with 90% of the test cases to learn a statistical model for each transition. We then apply Argus in detection mode on the bug-free version with the remaining 10% of the test cases. The smallest log likelihood value observed in each execution is recorded and sorted in ascending order. Among the sorted log likelihood values, the one with $x\%$ of the values smaller than it is recorded as the log likelihood threshold corresponding to false alarm limit of $x\%$. Intuitively, if we use this as the threshold, Argus will sound an alarm on $x\%$ of the normal executions. We discover the log likelihood thresholds corresponding to false alarm rates $\{5\%, 10\%, 15\%, 20\%, 25\%, 30\%\}$ (assuming 30% is the upper limit of user tolerance) in this way. Each application has its own set of thresholds.



(a) Average Detection Rate on All Faulty Versions (b) Effective Rate on All Faulty Versions

Fig. 2. Argus Bug Detection Power

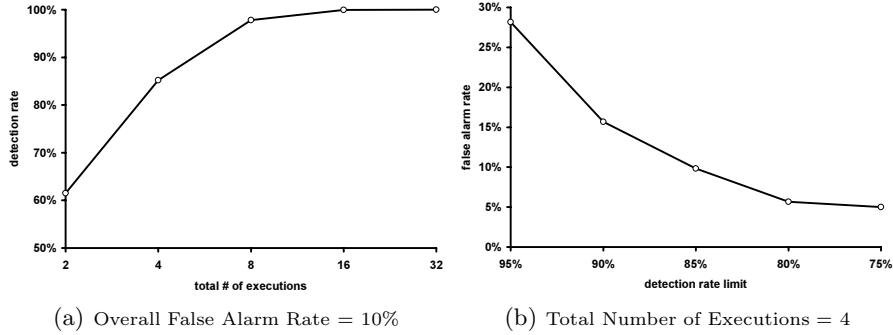


Fig. 3. Argus Overall False Alarm Rate, Overall Detection Rate, and Total Number of Execution

We note that Argus does not rely on bug-free version to discover thresholds. Thresholds discovered in this way are equivalent to thresholds discovered on test cases where no bug sites are covered, which corresponds to the whole passing test suite before software release in real world. In our experiment, an alternative is discovering the thresholds for each faulty version using passing cases on that particular version. We consider it less proper because the passing cases in each faulty version may not be a general representative of normal cases (while the whole test suite is assumed to be general).

We then measure the detection rates under these thresholds by running each of the faulty versions of each application through *all* the test cases. For each faulty version, Argus (in training mode) learns a transition distribution profile from the passing executions. We then use the set of thresholds corresponding to that application discovered above, and measure on how many of the failing executions Argus (in detection mode) raises alarms under the false alarm limits $\{5\%, 10\%, 15\%, 20\%, 25\%, 30\%\}$. The detection rate under a false alarm limit is then computed using Equation 7. The average detection rate over the 130 faulty versions with regard to different false alarm levels are shown in Figure 2(a).

Effectiveness (Equation 20) is measured by comparing the detection rate on each faulty version with the corresponding false alarm limit. The effective rate (Equation 21) is then measured by averaging the results of Equation 20 on all the 130 faulty versions. The average effective rate is shown in Figure 2(b).

Argus sounds an alarm on 118 out of 130 faulty versions. On average, Argus has 32.89% chance to detect the bug in each faulty version using a single detection run when the false alarm tolerance is 5%. This average detection rate increases almost linearly to 52.27% when the false alarm tolerance is 25%. It is further increased to 53.54% when the false alarm tolerance is 30%. When the false alarm tolerance is $\{5\%, 10\%, 15\%, 20\%, 25\%, 30\%\}$, Argus is *effective* (as defined in Equation 20) in detecting bugs in $\{74, 73, 77, 81, 83, 84\}$ of the faulty versions. Altogether, 102 out of 130 bugs can be *effectively* detected by Argus under at least one of the false alarm rate configurations.

Using the $\{f_i\}$ and average $\{d_i\}$ values discovered above, we can maximize the detection rate, minimize the false alarm rate, and assess how many executions are sufficient when multiple executions are available. Equations 13,15 and 17 can be written in short forms as $D_{opt}(f, R)$, $F_{opt}(R, d)$, and $R_{opt}(f, d)$, respectively. These three functions can be plotted with 3D charts. For simplicity and readability reasons, we project them onto 2D charts by fixing one variable at a reasonable value and then plot the relationship between the remaining two (Figure 3). Interested readers can find other values of interest using the code provided on Argus web site. Due to space constraints, we present only two of the projections. The remaining projections can be found on Argus web site. From Figure 3, we can see that when the false alarm tolerance is 10%, 8 executions are sufficient to guarantee a 98% detection rate; if we have only 4 runs, we can let the overall detection rate slip to 80% to bring the overall false alarm rate down to 6%. These results show that the detection power of Argus increases rapidly with a small number of executions, and the false alarm rate can be kept to a low level. This in turn implies that, in general, using Argus can greatly reduce the number of test cases needed, leading to a shorter debugging cycle.

5.2 Bug Report Quality

To measure the quality of the bug report generated by Argus, we adopt a paradigm which was proposed by Renieris et al. [20] and was later adopted by Cleve et al. in CT [4] and Liu et al. in SOBER [15]. This paradigm estimates the human effort to locate the bug based on the bug report. The measure was based on the size of the subgraph of the program dependence graph (PDG) the human must explore (starting from the statements pointed out by the bug report) vs. the size of the full PDG. Because Argus detects anomalous *transitions* rather than *statements*, we use a variant of the paradigm:

1. The FSA is a connected graph G , where each state is a vertex and each transition is an edge.
2. If $A \rightarrow B$ is a transition, and if the buggy statement is executed after event A (with no other events in between) yet before event B (with no other events in between), we consider $A \rightarrow B$ a *defect edge*. Note that one buggy statement may cover multiple defect edges depending on runtime control flow. We use V_{defect} to denote the set of vertices covered by defect edge(s).
3. Given a bug report, the set of vertices covered by the top k suspicious transitions reported by Argus is denoted V_{blamed} .
4. A programmer performs a breadth-first search from V_{blamed} until a vertex in V_{defect} is reached. The set of vertices covered by the breadth-first search is denoted $V_{examined}$.
5. The T-score is defined as:

$$T = |V_{examined}|/|V| \times 100\% \quad (22)$$

where $|V|$ is the size of graph G .

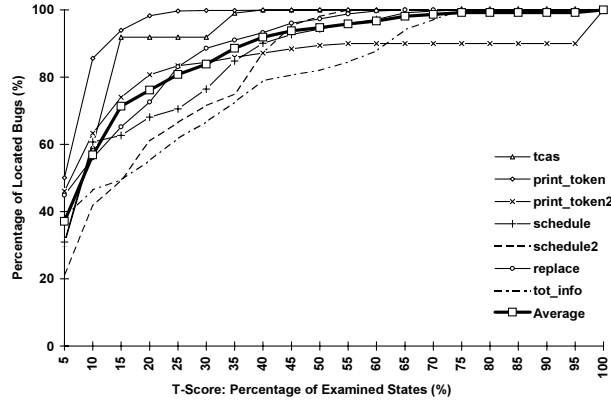


Fig. 4. Argus Bugs Located (cumulative) vs. Average T-score, $k = 5$

The T-score *estimates* the human effort to reach the buggy transition from the top k suspicious transitions reported by Argus. A high T-score corresponds to more human effort, and in the worst case a score of 100% indicates the whole program must be examined. We measure the T-score of Argus bug report for *each buggy execution* in each faulty version.

Although Argus' major design goal is runtime bug detection, the bug report it generates is precise enough that it can be used for efficient bug localization as well. Figure 4 shows the number of bugs that can be localized vs. the average T-score (percentage of states examined in ext-FSA, as defined in Equation 22). Since SOBER yields best performance when $k = 5$ (where k is the number of the most suspicious statements to start the breadth-first search from), we use $k = 5$ to make the results more comparable, where k is the number of the most suspicious transitions to start breadth-first search from. On average, when 10% of the states are examined, Argus can help the user localize 56.82% of the bugs. As presented in our previous work [15], SOBER is able to localize 52.31% of the bugs in Siemens benchmark when 10% of the statements are examined; while other known techniques like *Scalable Statistical Bug Isolation* (Liblit et al. [14]) and CT (Cleve et al. [4]) are able to localize 40.00% and 26.36% of the bugs respectively in the same benchmark. If we assume that the effort in examining $x\%$ of the total states is comparable to $x\%$ of the total statements, then Argus outperforms the best known bug localization techniques. On the Argus web site, we present discussion showing that if Argus and a statement-based scheme (e.g. SOBER) have the same T-score value, the user needs to examine fewer statements using Argus. The high precision of the Argus bug report and Argus' capability in generating the bug report using a single execution make it a powerful bug localization tool even if we do not need its runtime bug detection capability.

5.3 Runtime Overhead and Profile Size

Because Siemens benchmark programs are small (138 – 516 LOC), they are not suitable testbeds to measure Argus' runtime overhead in large, more realistic ap-

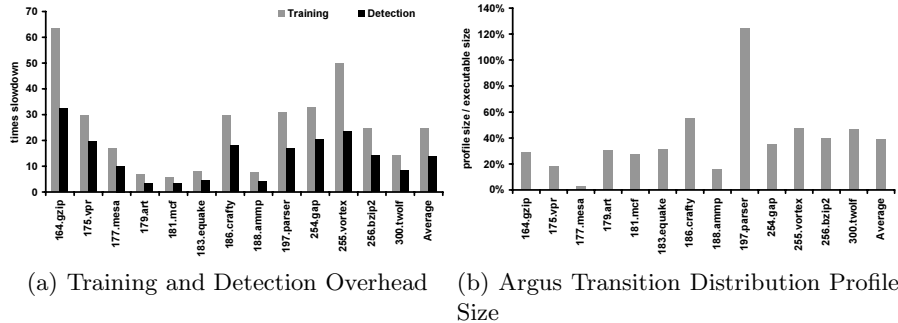


Fig. 5. Argus runtime overhead and transition distribution profile size on SPEC2000

plications. We use the standard SPEC2000 benchmarks to study Argus runtime overhead and the transition distribution profile size. We use all SPEC C programs² except 176.gcc, which the Cetus compiler cannot properly compile. All the experiments are run on a DELL Precision 350 workstation (3.06GHz Pentium IV, 1.5G memory) running RedHat Linux 9.0 with the Intel C/C++ compiler `icc 8.1`. All of the programs are compiled with the recommended optimization flag `icc -O2`

Figure 5 shows Argus runtime overhead in training and detection modes and the ratio of the transition distribution profile size to the original executable size. On average, Argus suffers 24.6X overhead in training mode and 13.8X overhead in detection mode. The corresponding standard deviations are 16.9 and 8.7. The detection overhead of Argus is comparable with runtime detection tools like DIDUCE [8], `rtcc` [23] and `SafeC` [1], and is considerably lower than tools like `Purify` [10] and runtime type checking [16]. Argus' detection scheme is compatible with random sampling [13]. It can use random sampling to achieve low runtime overhead at the cost of more detection executions. Therefore, Argus is also a good technique to be used in production runs. The transition distribution profile size is 38.6% of the original executable size on average, with a standard deviation of 28.3%.

6 Conclusions and Future Work

Statistical debugging is a powerful technique for identifying bugs that do not violate programming rules or program invariants. In this paper, we present an online statistical bug detection technique called Argus. Argus is capable of detecting bugs in a single execution and can raise an alert at runtime when bug symptoms occur. Argus eliminates the requirement for labeling passing and failing executions. Argus generates more precise bug reports than the best known bug localization techniques.

² 164.gzip, 175.vpr, 177.mesa, 179.art, 181.mcf, 183.equake, 186.crafty, 188.ammp, 197.parser, 254.gap, 255.vortex, 256.bzip2, and 300.twolf

The authors are investigating making Argus capable of detecting bugs reflected in program behaviors other than runtime control flow. Second, we want to develop implementations for other languages (like Java), and to reduce the runtime overhead.

References

- [1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.
- [2] L. Burnell and E. Horvitz. Structure and chance: melding logic and probability for software debugging. *Communications of the ACM*, 38(3):31–ff., 1995.
- [3] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury Press, second edition, 2001.
- [4] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, 2005.
- [5] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005.
- [6] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 339–348, 2001.
- [7] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, 2000.
- [8] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, 2002.
- [9] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. *Journal of Software Testing, Verifications, and Reliability*, 10(3):171–194, 2000.
- [10] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference*, 1992.
- [11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, 1994.
- [12] T. A. Johnson, S.-I. Lee, L. Fei, A. Basumallik, G. Upadhyaya, R. Eigenmann, and S. P. Midkiff. Experiences in using cetus for source-to-source transformations. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2004.
- [13] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 141–154, 2003.
- [14] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 conference on Programming Language Design and Implementation*, 2005.

- [15] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proceedings of The fifth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 05)*, 2005.
- [16] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via run-time type checking. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 217–232, 2001.
- [17] Software errors cost U.S. economy \$59.5 billion annually, 2002. NIST News Release 2002-10.
- [18] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, pages 465–475, 2003.
- [19] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, pages 287–296, 2003.
- [20] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, 2003.
- [21] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, 1998.
- [22] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 144, 2001.
- [23] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software Practice and Experience*, 22(4):305–316, 1992.
- [24] K. Teknomo. Recursive simple statistics tutorial. online document. <http://people.revoledu.com/kardi/tutorial/RecursiveStatistic/>.
- [25] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AcMon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'04)*, 2004.