© 2024 Keyur Parag Joshi

COMPOSITIONAL ANALYSIS OF THE EFFECTS OF UNCERTAINTY ON COMPUTATIONS

BY

KEYUR PARAG JOSHI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Doctoral Committee:

Associate Professor Saša Misailović, Chair
Professor Sarita Adve
Professor Sayan Mitra
Associate Professor Antonio Filieri, Imperial College London

Abstract

Modern computations must regularly interact with imprecise sensors, deal with hardware failures, and operate on incomplete or inaccurate input data. Developers may also resort to intentionally adding approximate algorithms and machine learning models to such computations in order to make them tractable.

Uncertainty analyses provide developers with the means to ensure that uncertainty introduced into a computation in this manner does not lead to unwanted or dangerous consequences. However, developers regularly modify modern computations throughout their lifetime to fix bugs and add features. An uncertainty analysis can become prohibitively expensive if it must be run from scratch every time a developer modifies the computation. Compositional analyses of uncertainty, which analyze different components of a computation in isolation and then analyze the overall computation, would have a clear advantage in this scenario; when a computation is modified, it would not be necessary to re-analyze the unmodified components. While researchers have developed compositional analyses for testing a variety of other properties, there is less work on developing compositional and precise analyses of uncertainty.

In this dissertation, I present my work which shows that composable uncertainty analyses can have precision close to that of monolithic, non-composable uncertainty analyses. First, I describe a statistical analysis of the accuracy of approximate randomized algorithm implementations and computations running on unreliable hardware. Second, I describe a composable analysis of uncertainty in autonomous vehicle systems. Third, I describe an analysis that calculates how recovery mechanisms can increase the reliability of critical subcomputations running in an unreliable environment. Lastly, I describe a composable analysis that determines how soft errors affect computations and selects sets of vulnerable instructions to protect. The availability of composable analyses of uncertainty will encourage developers to regularly test the effects of proposed changes on the uncertainty characteristics of modern computations, possibly as part of regression testing suites.

Acknowledgments

First of all, I would like to thank my extended family, especially my mother, father, and sister, for their unending love, support, and constructive criticism. They have taught me valuable life skills, and have always ensured that I had the best opportunities for education.

I would like to thank my advisor Sasa Misailovic for his valuable guidance, feedback, and funding. Sasa was always available to provide advice, whether in person or online. During the first few years of my PhD, he closely mentored me to ensure that I got off to a good start, while in the later years, he offered me freedom to conduct research as I saw fit.

I would like to thank my friends at UIUC, my friends in Naperville, and my friends from IIT Hyderabad who have regularly provided companionship and help during my PhD. I would especially like to thank my labmates Ashitabh, Jacob, Rem, Saikat, Shubham, Vimuth, Yifan, Zixin, and others for being great friends, coworkers, research partners, snack providers, and much more. I would also like to thank Owolabi, who provided useful advice on being a PhD student, Chiao, who was a co-contributor to GAS (Chapter 3), and Rahul, Abdulrahman, and Tommaso, who were co-contributors to FastFlip (Chapter 5).

I would like to thank Ramakrishna Upadrasta at IIT Hyderabad, who introduced me to research during my undergraduate studies and motivated me to go for a postgraduate degree. I would also like to thank Albert Cohen (formerly) at INRIA for giving me my first opportunity to participate in research in another country.

I would like to thank the faculty at UIUC, especially Sarita Adve, Robin Kravets, Darko Marinov, Sayan Mitra, Gagandeep Singh, and Tianyin Xu, for their support during my time at UIUC. I would also like to thank the other professors and teachers who have taught me since my childhood for all contributing towards getting me to where I am today, as well as the staff and administrators who made the work of these educators possible.

I would like to thank the people at MIT and the Scratch Foundation responsible for designing and maintaining the Scratch programming environment, which introduced me to programming over fifteen years ago. Scratch allowed me to experience the joys (and frustrations) of programming at an early stage, planting a seed that would eventually lead me to choose a career in computer science.

I would like to thank the members of the community who reviewed my conference and journal submissions and provided harsh but often necessary criticism. This critical feedback helped me to polish and improve the works presented in this dissertation from rough drafts into their current state. I would also like to thank the members of my final exam committee, who provided similar feedback on this dissertation.

The research presented in this dissertation was funded through various grants, in particular NSF Awards 1629431, 1703637, 1846354, 1956374, 2008883, 2028861 and the DARPA Domain-Specific Systems On Chip (DSSOC) program, part of the Electronics Resurgence Initiative (ERI), under Contract HR0011-18-C-0122.

Table of Contents

Chapter 1: Introduction

The presence of uncertainty is a first-class concern for modern computations[1]. Some forms of uncertainty are inherently unavoidable. For example, modern computations must regularly deal with noise from imprecise sensors [1, 2, 3], hardware failures [4, 5, 6, 7, 8, 9], and missing or unreliable input data [10, 11, 12]. Developers may also choose to intentionally introduce uncertainty into a computation. For example, approximate algorithms are often necessary when processing large volumes of data [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26], imprecise machine learning models are increasingly becoming a part of computations in order to autonomously perform tasks that would otherwise require human effort [27, 28, 29, 30], and in differential privacy [31, 32], developers reduce the exposure of sensitive information by intentionally adding noise to a computation's data.

The tolerance of computations to uncertainty varies from domain to domain. In domains such as audiovisual media processing, the output does not need to be fully precise to be acceptable to a user, and this fact is often exploited to save resources [33, 34, 35, 36, 37]. Some computations are naturally *self-stabilizing*, in the sense that they can gradually reduce uncertainty over time. For example, iterative methods for finding roots of equations start from an initial guess and refine it to arrive at the solution [38, 39]. In domains such as robotic surgery, automatic medical diagnosis, and autonomous vehicles, developers must carefully control uncertainty to prevent loss of life and damage to property. For example, there are growing calls to regulate the behavior of potentially imprecise systems in autonomous vehicles [40, 41, 42, 43].

Uncertainty analyses. Developers use *uncertainty analyses* to ensure that the error in the output of a computation is within acceptable levels. These uncertainty analyses consider how uncertainty can be introduced into the computation, and how it propagates through the computation up to its output [36, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60]. Some uncertainty analyses go further by adding approximations to existing computations that introduce an acceptable level of uncertainty in exchange for performance benefits [37, 61, 62, 63, 64, 65, 66] or by adding mechanisms to detect and recover from the negative effects of excessive uncertainty [67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79]. Depending on the complexity and precision of an uncertainty analysis and the complexity of the analyzed computation, the time required for the analysis can range from a few seconds to several hours.

[1]In the scope of this dissertation, computations encompass programs and simulations of software-intensive systems.

Modern computations regularly undergo changes throughout their lifetime. Even after the initial development and testing of the computation is finished and they are deployed in the real world, developers continue to make changes to them to fix newly found bugs or add new features. These frequent changes complicate the task of *efficiently* using uncertainty analyses to prove that the uncertainty in the output of a computation is tolerable, or that the computation will not violate a safety property due to excessive uncertainty. Most uncertainty analyses must analyze the entire computation from scratch after each change, which is prohibitively expensive.

Compositional analyses of uncertainty would offer a clear benefit in such scenarios. A conventional compositional analysis first divides the computation into multiple components. Second, it performs sub-analyses on some or all of these components in isolation from the rest of the computation. Finally, it analyzes the computation as a whole. In this last step, the compositional analysis uses the results of the component-wise sub-analyses to simplify the end-to-end analysis. The goal of a compositional analysis is to *reuse* analysis results each time a small change is made to the computation. If only one component of the computation is modified, then it is often unnecessary to again perform the sub-analyses on the unmodified components. Instead, the compositional analysis performs the sub-analyses only on the modified components. Then, it uses the new results of the sub-analyses for the modified component when it analyzes the computation as a whole.

Researchers have developed several compositional analyses for testing, optimization, and traditional safety verification. Compositional symbolic execution [80] finds assertion violations in large computations in a scalable manner. SMART [81] tests functions in isolation and creates summaries that can be used when testing computations using those functions. Compilers regularly use intra-procedural analyses [82] to optimize each procedure within a computation in isolation.

Compositional analyses of uncertainty (and safety in the face of uncertainty) have received comparatively less attention from researchers. Chisel [36] creates and uses specifications that describe the effects of uncertainty on a function when analyzing the uncertainty of a computation using it. Pasareanu et al. [83] separately analyze learning based components of autonomous vehicles, and use the analysis results when reasoning about safety properties of the full vehicle system.

Compositional analyses are only useful if they do not lose significant precision over monolithic analyses that analyze the full computation at once. Moreover, compositional analyses must carefully consider how separately analyzed components interact with and affect the results of one another. If a component of a computation is modified, the compositional analysis must correctly account for the change in interaction of the modified component with

Table 1.1: Overview of work described in this dissertation

Technique	**Uncertainty source**	**Key aspects**
AxProf [84]	Algorithm, hardware	Specification language, statistical analysis
GAS [85]	Input, algorithm	Surrogate modeling to reduce simulation cost
Aloe [86]	Hardware	Specification language, probabilistic verification
FastFlip [87]	Hardware	Compositional error injection and propagation analysis

the rest of the computation.

For uncertainty analyses, an additional complicating factor is that components of a computation may propagate uncertainty differently depending on the input or input distribution. Uncertainty analyses such as Chisel [36] must make rigid assumptions about the input in order to mitigate this issue. For example, if Chisel separately analyzes a function and wishes to integrate the function into a computation, it must either 1) use an imprecise specification of the effect of uncertainty on the function, or 2) use a precise specification that is only applicable to a small subset of possible function inputs. Analyses of uncertainty caused by hardware errors must additionally consider how components may cause unexpected interactions in the form of side effects that would not occur in an error-free execution.

1.1 DISSERTATION STATEMENT

My dissertation statement is as follows:
A compositional analysis of the effects of uncertainty on a computation can have precision close to that of a monolithic analysis of the computation, while saving time over the monolithic analysis when the computation is modified throughout its lifetime.

1.2 DISSERTATION OVERVIEW

Table 1.1 provides an overview of my contributions in support of the dissertation statement. Column 1 shows the name of the technique. Column 2 shows the source of uncertainty in the computation that the technique analyzes. For all sources, the probability that uncertainty is present and the magnitude of uncertainty can vary. Column 3 briefly describes some key aspects of the technique. The subsequent chapters in this dissertation describe each contribution in detail:

Chapter 2: AxProf. AxProf [84] is a framework for statistical testing of implementations of approximate algorithms, and of computations running on power-saving but unreliable hardware. The authors of theoretical approximate algorithms provide probabilistic accuracy and performance guarantees when proposing the algorithm. To statistically test an implementation of such an algorithm to ensure that it satisfies these guarantees, it is necessary to gather the implementation's accuracy and performance metrics over a sufficient quantity of executions, and then apply one or more appropriate statistical tests. Similarly, manufacturers of unreliable hardware provide probabilistic guarantees about the likelihood of hardware failures. To statistically test a computation running on such hardware to ensure that it satisfies user-specified reliability guarantees, it is necessary to gather information about the computation's reliability over a sufficient quantity of executions, and then apply one or more appropriate statistical tests.

The AxProf framework automates most of this data gathering and statistical testing process. The developer provides the theoretical or desired accuracy and performance specifications in an unambiguous mathematical notation. Given these specifications, AxProf generates code to execute the implementation or computation a statistically significant number of times and then automatically apply the appropriate statistical test(s). We used AxProf to analyze twelve approximate algorithm implementations and three computations running on unreliable hardware. AxProf found five previously unknown bugs among the approximate algorithm implementations. After we fixed these implementations, AxProf verified that they now satisfied the accuracy and performance specifications.

The approximate algorithms whose implementations we analyzed with AxProf are used as sub-components of larger applications such as image search and clustering, efficient caching, web traffic estimation, data stream summarization, constrained optimization, etc. These applications depend on the correctness of the approximate sub-components to ensure the validity of the overall accuracy and performance guarantees. AxProf's statistical testing ensures that the sub-components behave as expected, allowing developers and compositional analyses of the end-to-end applications to rely on the algorithm's accuracy guarantees.

Chapter 3: GAS. GAS [85] is an approach for creating fast and accurate surrogate models of complex, end-to-end autonomous vehicle systems. High-fidelity simulations of these complex systems for evaluating vehicle safety are costly, especially as these systems are modified over the course of development. Such systems regularly contain complex state perception and control components, which hinder the creation of less computationally intensive surrogate models. GAS's two-stage approach first replaces complex perception components with a perception model. Then, GAS constructs a surrogate model of the complete vehicle system. GAS enables the construction of various types of surrogate models, but we found that Generalized Polynomial Chaos (GPC) produced the most consistently precise surrogate models. GAS's approach also allows for reuse of the perception model when vehicle control and dynamics characteristics are altered during vehicle development, saving significant time.

We demonstrate the use of these surrogate models in two applications. First, we estimate the probability that the vehicle will violate a safety property over time. For example, we estimate the probability that a crop monitoring vehicle will hit a row of crops. Second, we perform global sensitivity analysis of the vehicle system with respect to its state in a previous time step. For both applications, we consider five scenarios concerning crop management vehicles that must not crash into adjacent crops, self driving cars that must stay within their lane, and unmanned aircraft that must avoid collision. Each of these systems contains a complex perception or control component. GAS's surrogate models provide an average speedup of 3.7× when estimating the probability of violating a safety property and 1.4× for sensitivity analysis, while still maintaining high accuracy.

Chapter 4: Aloe. Aloe [86] is a static analysis that quantifies the reliability of computations that are susceptible to soft errors (e.g., bitflips due to noise) and which attempt to mitigate such errors using recovery mechanisms.

Aloe models recovery mechanisms using *try-check-recover* blocks. The *try* block performs an imprecise computation, the *check* block checks the result for errors, and the *recover* block re-executes the computation if an error is detected. Aloe analyzes the reliability of try-check-recover blocks separately from the rest of the computation. Aloe then uses this calculated reliability as an input when calculating the reliability of the end-to-end computation. Aloe can reason about either precise or imprecise error detection and recovery mechanisms.

We implemented Aloe and analyzed eight computations used for web search, media processing, financial analysis, graph processing, and numerical analysis. Compared to previous works, Aloe was able to verify stricter reliability guarantees of the try-check-recover blocks as well as the end-to-end computations.

Chapter 5: FastFlip. FastFlip [87] is a compositional error injection analysis that aims to find assembly instructions in a program that are vulnerable to Silent Data Corruptions (SDCs) in the presence of errors. Unlike detectable errors (e.g., out of bounds outputs) or crashes, SDCs may not be detected until they negatively affect the outputs of downstream applications. FastFlip divides the program into multiple sections and analyzes each section separately. FastFlip calculates how SDCs are propagated from one program section to another to obtain the set of vulnerable instructions for the whole program. FastFlip saves a significant amount of analysis time when the program is modified; in the best case scenario, FastFlip must only perform expensive error injection analysis on the modified section and can reuse the previous results for the other sections.

We instantiate FastFlip by using the Approxilyzer [54] error injection analysis and the Chisel [36] SDC propagation analysis and use it to analyze five benchmarks, along with two modified versions of each benchmark. FastFlip speeds up the analysis of incrementally modified programs by 3.2× on average. FastFlip selects a set of instructions to protect against SDCs; we compare this set to the set of instructions selected by an Approxilyzer-only approach. FastFlip's selection protects against a similar number of SDCs for a similar overhead as Approxilyzer's selection, even when programs are modified.

Analyses such as Aloe can use FastFlip's results to determine the reliability of sub-computations in the *try* and *recover* blocks in order to calculate the overall reliability of a computation with recovery mechanisms. Likewise, we also use FastFlip to confirm that modifications that add SDC detection mechanisms to computations increase the overall reliability of the computation.

Chapter 6: Conclusion and future work. Lastly, I conclude the dissertation and discuss some interesting directions for future research based on my current work.

Chapter 2: AxProf

Applications such as machine learning, data analytics, computer vision, financial and weather forecasting, and content search require low-latency processing of massive data sets. To process such large amounts of data in a timely manner, researchers have developed various approximate algorithms and data structures that reduce resource consumption at the cost of accuracy. Similarly, applications running on resource-constrained devices, such as those used in edge computing or digital agriculture, may have to use such approximate algorithms and data structures, as well as unreliable low-power hardware, to limit energy consumption.

Many approximate algorithms come with analytically derived specifications of accuracy that are typically probabilistic in nature. For example, Locality Sensitive Hashing (LSH) [13, 14] finds neighbors in a set of points, by using a hash function that produces similar hashes for nearby points. LSH guarantees that it will find nearby points *with high probability*. Such probabilistic specifications have been proposed for applications in areas as diverse as theoretical computer science [15, 16, 17, 18, 19, 88], numerical computing [20, 21, 22, 89], and databases [23, 24, 25, 26, 90]. Probabilistic specifications are also used to describe the reliability of low-power hardware and by compilers and analyses that target computations running on such hardware [36, 44, 47, 91, 92, 93].

In order to ensure the validity of the accuracy guarantees of the overall application, it is vital that these uncertain computations[2] satisfy their probabilistic specifications of accuracy. Unfortunately, despite the availability of rigorous specifications, a software developer who needs to implement, test, and tune these uncertain computations has little tool support for this effort. Standard profilers only track and build models of run time and memory consumption for individual inputs [94, 95, 96] or build performance models for multiple input sizes in the case of *algorithmic profiling* [97, 98]. While researchers have provided guidelines for rigorously applying statistical testing in software engineering [99], developers must manually perform numerous tasks to implement these guidelines: infer the properties of the mathematical (probabilistic) specification, write code to check this specification, decide on the appropriate statistical test and its parameters (e.g., confidence or power), provide appropriate inputs, and interpret obtained statistical metrics. Frustrated by such manual effort, developers often resort to ad-hoc testing that provides insufficient statistical confidence, or leads to overly conservative choices of test parameters. Moreover, manually written testing code can itself have subtle bugs that reduces its effectiveness. A more promising alternative is to automate these tasks with dedicated profiling and testing frameworks.

[2]In the scope of this chapter, uncertain computations refer to implementations of approximate randomized algorithms and computations running on unreliable hardware.

Our work. We present AxProf, an algorithmic profiling framework for analyzing primarily accuracy, but also resource consumption of uncertain computations. AxProf gathers data on accuracy and resource usage, uses statistical tests to check if any of this data deviates from the developer-provided specifications, and warns the developer if that is the case. AxProf is available as open-source software at `www.axprof.org`.

The key novelty of AxProf is the automatic generation of statistical testing code from a high-level mathematical probabilistic specification. AxProf also determines the number of inputs or executions required in order to achieve a desired level of statistical confidence. AxProf supports *probability predicates*, which reason about the probability that the output of an uncertain computation satisfies a certain condition, and *expectation predicates*, which reason about the expected value of the output. AxProf also supports *universal quantification*, which requires such predicates to hold over a range of input or output items. These predicates together capture the key properties of many representative randomized and approximate algorithms and unreliable computations. For instance, they are expressive enough to describe a majority of the accuracy specifications of the algorithms from [100]. An important concern of AxProf's language design is to present the specifications in an unambiguous manner. In general, probabilistic specifications can be defined over different sets of events, which may require different sampling procedures. In AxProf, a developer explicitly writes if a probabilistic specification is over inputs, items within an input, or executions, in line with the theoretical specification.

Statistically testing uncertain computations often requires a large number of concrete inputs. To automatically produce representative inputs, AxProf provides several input generators for scalars, vectors, and matrices. The developer can adjust the parameters of the input generators and use these parameters as part of the accuracy and performance specifications. We present a dynamic analysis that infers which input generator properties are relevant to the application's accuracy.

Results. We evaluated AxProf on a set of twelve implementations of well-known randomized approximate algorithms from the domains of data analytics and numerical linear algebra, as well as three computations running on simulated unreliable hardware. Each application has an analytically derived specification of accuracy, performance, and memory consumption. We demonstrate that AxProf can help developers in two scenarios: 1) profiling to understand the behavior of the application and 2) identifying potential errors.

AxProf helped us to identify and fix previously unknown problems with five different implementations of approximate algorithms. Our analysis shows that these problems could not have been identified with standard profilers that track only memory or runtime. The problems were caused by incorrect implementations of the algorithms or their key components like hash functions. AxProf also identified some implementations that used additional optimizations in their resource consumption. These optimizations resulted in a different complexity of resource consumption than that specified by the algorithm authors. For instance, some implementations allocated resources dynamically (only when needed) or used polyalgorithms (i.e., they composed multiple algorithms that work better for different input sizes, and switched algorithms dynamically).

These results demonstrate that AxProf's focus on accuracy analysis opens a new dimension in algorithmic profiling. AxProf's statistical testing ensures that vital approximate and randomized sub-components of data-centric applications behave as expected, allowing developers and compositional analyses of the end-to-end applications to rely on the algorithm's accuracy guarantees.

2.1 EXAMPLE

Locality Sensitive Hashing (LSH) [13, 14] is an algorithm for finding points that are near a given query point in multidimensional space. Instead of directly computing the distance of the query point to every other point in the space, LSH maintains a compact representation of the points and their locations using a set of hash maps. The keys of the maps are hash signatures and the values are the list of points with that signature.

To obtain a hash signature of a point, LSH uses a *locality sensitive* hash function. Depending on the desired similarity metric between points (ℓ_1 distance, ℓ_2 distance, cosine similarity, etc.), different LSH function families exist which hash similar points according to that metric to the same hash signatures with high probability.

When it receives a query, LSH calculates the query point's signature and returns all stored points with that signature. LSH can increase the number of similar points found by using l different hash maps. LSH can also concatenate signatures from k different hash functions as the keys in each hash map to increase the probability that dissimilar points will be mapped to different bins. Each of these hash functions must be drawn uniformly at random from the same hash family.

AxProf specification. We assign each point in the dataset an index. One representation of the LSH output is a list of pairs of indices (qi, di), where the first index qi is the index of the query point q and the second index di is the index of the detected neighbor d. For our example, we use the same set of points as both data points and query points, to find groups of neighboring points within the dataset.

```
1 Input list of (list of real);
2 Output list of (list of real);
3 TIME k*l*datasize;
4 SPACE l*datasize;
5 ACC forall di in indices(Input), qi in indices(Input) :
6   Probability over runs [ [qi, di] in Output ] ==
7     HashEqProb(Input[di], Input[qi], k, l)
```

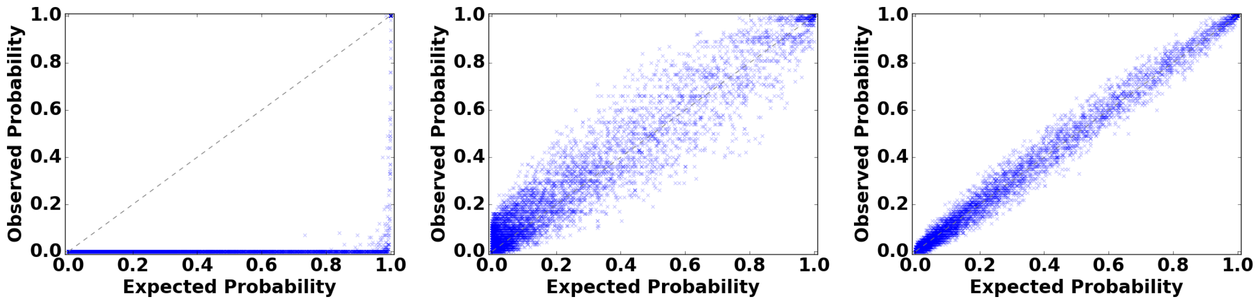
Figure 2.1: AxProf specification for LSH

Suppose an individual hash function chosen uniformly at random from the desired hash function family puts the points d and q in the same hash bin with probability $p_{d,q}$, where $p_{d,q}$ is higher when d and q are more similar according to the similarity metric. Then, for all d and q, the probability that LSH will include (qi, di) in its output is $1 - (1 - p_{d,q}^k)^l$ [13]. Figure 2.1 shows how we write the full specification in the AxProf specification language. Lines 1 and 2 indicate the data types of the input and the output, respectively. Lines 3 and 4 give the time and memory usage specification of LSH, respectively. As each point must be stored in each hash table, the memory usage is $O(ln)$, where n is the number of data points. Storing a point requires calculating lk hashes. The total time required to construct the hash tables is $O(lkn)$.

The last three lines give the accuracy specification. Informally, it specifies that, for all possible pairs of indices over the input (qi, di), the probability that a particular execution of LSH has `[qi,di]` in its output is equal to the return value of `HashEqProb`, which is a user-defined function that calculates the expression $1 - (1 - p_{d,q}^k)^l$.

AxProf uses this specification to automatically generate code to check that the property holds. The code aggregates the outputs of the implementation over multiple executions. Then, for each pair of indices, it calculates the fraction of executions for which the pair is in the output. It compares this fraction against the return value of `HashEqProb` using the binomial test. Finally, it combines the results of the binomial tests for each pair of indices using Fisher's method [101]. For time and memory, AxProf generates code to perform regression and checks if the time and memory usage varies according to the provided specifications when k, l, and n are varied. The full details of code generation are shown in Section 2.3.

Testing the implementation. We tested *TarsosLSH* [102], an implementation of LSH in Java that has its own testing framework and over 100 stars on GitHub. The algorithm can be configured through two parameters `k` and `l`, for which the developer specifies a list of values of interest. We instruct AxProf to uniformly generate random 2D points with each coordinate in the range $[-10, 10]$.



(a) Before (b) After fixes 1 and 2 (c) After fixes 1-4

Figure 2.2: *TarsosLSH*: comparison of accuracy of the implementation before and after bug fixes. Each dot compares the expected and observed probability for a query-datapoint pair. Ideally all dots should lie on the diagonal line.

Identifying and fixing bugs. While profiling *TarsosLSH* for the ℓ_1 distance metric, AxProf indicated errors for several values of k and l. We used the visualization feature of AxProf and observed that many points were not being considered similar at all, as shown in Figure 2.2a. Each dot in the plot represents a query-datapoint pair. The x and y coordinates of the point denote the expected and observed probability, respectively. Ideally, all dots should lie close to the diagonal line.

This prompted us to investigate the hash function used for ℓ_1 distance. We found that there were several inaccuracies in the implementation and use of the hash function. We fixed a bug that occurred due to operator precedence, followed by a bug caused by incorrect assumptions about the rounding of floating point values. Fixing these two bugs led to the result in Figure 2.2b. While this new result seemed to conform with the diagonal line as expected, AxProf's statistical tests reported that the number of outliers was still too high, indicating that more bugs were present.

On further investigation we found a bug in the method by which the implementation chose a hash function from the hash function family, and a bug in the method by which the outputs of the k different hash functions for a hash table were combined. Fixing these two bugs led to the result in Figure 2.2c. After fixing the fourth bug, AxProf indicated that the implementation conformed with the accuracy specification.

The implementation included a test method which tested the algorithm with various parameters. However, there was an error in the test code that miscounted the number of false negatives. This led the test to overestimate the recall of the implementation, i.e., the percentage of nearby points that were correctly identified. A user that depended on the results of this test would mistakenly believe that the implementation was correct, illustrating the need for automated tools like AxProf.

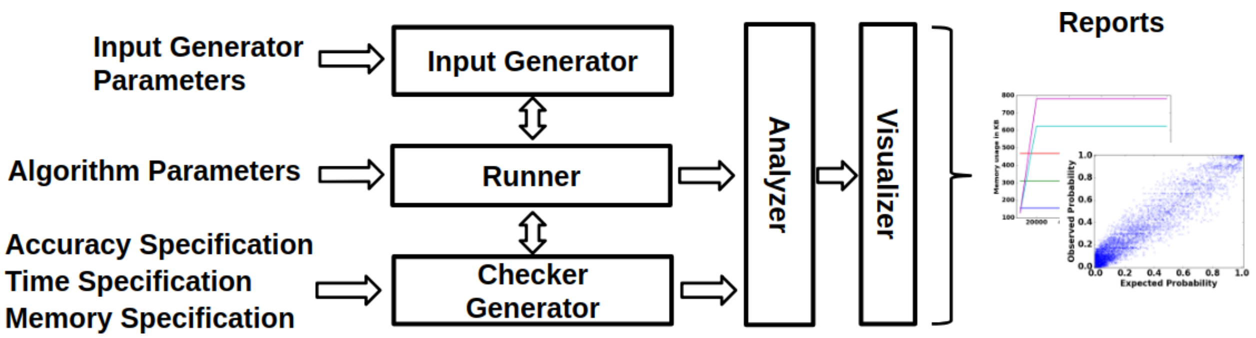


Figure 2.3: Overview of the AxProf framework

2.2 AXPROF OVERVIEW

Figure 2.3 presents an overview of the AxProf framework.

Inputs. AxProf takes three types of inputs:

- *Implementation and parameters:* AxProf takes an implementation of the uncertain computation to test and a set of typical configuration parameter ranges that the developer wishes to test the computation on.
- *Property specification:* The user provides an accuracy, time, and memory specification in a high-level language that resembles the mathematical specifications provided by the algorithm authors or hardware manufacturers.
- *Input generator settings:* AxProf provides several input generators for scalar, vector, and matrix data. Alternatively, the user can provide a custom input generator. The user can allow AxProf to infer parameters that are relevant to the accuracy of the uncertain computation, or fix some or all of the parameters to specific values that the user wants to test.

Components. AxProf has multiple components:

- The *checker generator* takes an accuracy, time, and memory usage specification from the user and generates code to aggregate output and resource consumption data and check that this data conforms to the specifications.
- The *input generator* generates inputs to test the implementation. It can experiment with various input parameters to determine which ones affect the accuracy of the uncertain computation.

$$
\begin{aligned}
c &\in \mathit{Constants}\\
x &\in \mathit{Vars} \cup \{\texttt{Input}, \texttt{Output}\}\\
f &\in \mathit{Functions}\\
\mathit{aop} &\in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\^{}}\}\\
\mathit{bop} &\in \{\texttt{\&\&}, \texttt{||}\}\\
\mathit{rop} &\in \{\texttt{==}, \texttt{>}, \ldots\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Spec} &::= \mathit{TDclr}\ \texttt{TIME}\ \mathit{DExpr}\ ;\ \texttt{SPACE}\ \mathit{DExpr}\ ;\ \texttt{ACC}\ \mathit{ASpec}\\
\mathit{TDclr} &::= \texttt{Input}\ \mathit{Type};\ \texttt{Output}\ \mathit{Type};\ (x\ \mathit{Type};)^{*}\ (f\ \mathit{Type};)^{*}\\
\mathit{Type} &::= \texttt{real} \mid \texttt{matrix} \mid \texttt{list of}\ \mathit{Type} \mid \texttt{map from}\ \mathit{Type}\ \texttt{to}\ \mathit{Type}\\
\mathit{ASpec} &::= \texttt{Probability over}\ \mathit{Qualif}\ [\mathit{BExpr}]\ \mathit{rop}\ \mathit{DExpr}\\
&\quad \mid \texttt{Expectation over}\ \mathit{Qualif}\ [\mathit{DExpr}]\ \mathit{rop}\ \mathit{DExpr}\\
&\quad \mid \texttt{forall}\ \mathit{Range}^{+}\ \texttt{:}\ \mathit{ASpec}\\
&\quad \mid \texttt{let x =}\ \mathit{DExpr}\ \texttt{in}\ \mathit{ASpec}\\
\mathit{Qualif} &::= \texttt{runs} \mid \texttt{inputs} \mid \mathit{Range}^{+}\\
\mathit{Range} &::= \texttt{x in}\ \mathit{DExpr} \mid \texttt{x in uniques}(\mathit{DExpr}) \mid \texttt{x in indices}(\mathit{DExpr})\\
\mathit{BExpr} &::= \mathit{BExpr}\ \mathit{bop}\ \mathit{BExpr} \mid \neg \mathit{BExpr} \mid \mathit{DExpr}\ \texttt{in}\ \mathit{DExpr} \mid \mathit{DExpr}\ \mathit{rop}\ \mathit{DExpr}\\
\mathit{DExpr} &::= c \mid x \mid x[\mathit{DExpr}] \mid |\mathit{DExpr}| \mid \mathit{DExpr}\ \mathit{aop}\ \mathit{DExpr} \mid f(\mathit{DExpr}^{+}) \mid [\mathit{DExpr}^{+}]
\end{aligned}
$$

Figure 2.4: The AxProf specification language

- The *runner* executes the implementation on an AxProf-provided input. It returns the generated output and resource consumption statistics to AxProf.
- The *analyzer* uses the code generated by the checker generator to test whether the implementation conforms to the provided specifications, and issues a warning if that is not the case. The developer can set parameters that influence how sensitive the statistical analyses are.
- The *visualizer* plots resource usage and accuracy statistics for visual inspection.

Specification language. The user writes probabilistic specifications of resource consumption and accuracy in a high-level language. Figure 2.4 presents its grammar. Specifications consist of type declarations, a time expression, a memory expression, and an accuracy specification. Knowing the input and output types allows AxProf to generate inputs in the correct format and correctly connect the checkers with the rest of the framework.

The specification can contain the special variables `Input` (the input data) and `Output` (the computation's output). The computation's configuration parameters can be accessed within the specification using their name. AxProf allows the developer to call helper functions written in Python. These functions may implement complicated testing conditions or compute exact solutions through an oracle.

The accuracy specification encodes two common predicates: 1) probability comparison (`Probability over` *Qualif*) and 2) expectation comparison (`Expectation over` *Qualif*).

Both predicates explicitly define the probability space via a qualifier. The qualifier can be a list of items, a set of executions (runs), or a set of inputs. If the qualification is over items in the input, each item has equal weight, as used in the average-case analysis of algorithms [103]. The accuracy specification can include universal quantification over one or more lists of items (`forall` *Range* $^+$). AxProf interprets these quantifiers as the requirement that the tests of the predicates inside the quantifiers should be correct for each item in the list.

2.3 CHECKER CODE GENERATION

AxProf derives statistical hypotheses from the accuracy specification and generates code to test them with common statistical tests. AxProf also calculates the number of executions or inputs necessary for obtaining sufficient statistical confidence. AxProf also generates code to check resource utilization specifications.

2.3.1 Background on Statistical Testing

A statistical hypothesis can be tested by observing samples of one or more random variables. A tester forms two hypotheses: a *null hypothesis* and an *alternative hypothesis*. Then they use an appropriate statistical test to calculate a p-value: the probability of obtaining a test statistic at least as extreme as the one observed, assuming the null hypothesis is true. If the p-value is too low, the tester can reject the null hypothesis.

Several statistical tests are available for various use cases. These tests are either parametric (they make some assumption about the population from which the data is drawn) or non-parametric (they make no such assumptions). Parametric tests are generally more powerful at detecting statistical anomalies, while non-parametric tests can handle more types of data and small sample sizes. AxProf uses the statistical tests described below.

The binomial test [104] is an exact non-parametric test used to compare the observed probability (p) of an event against an expected or threshold probability (p_0). For example, to test specifications that state $p \leq p_0$ we formulate a null hypothesis $H_0 : p \leq p_0$ and test it against the alternative hypothesis $H_1 : p > p_0$ using the binomial test. The *greater one-tailed*, *lesser one-tailed*, and *two-tailed* variants of this test respectively check if the observed probability is too high, too low, or different to the expected probability.

The one-sample t-test [105] is a parametric test used to compare the mean of *one* set of samples (μ) against an expected or threshold mean (μ_0). For example, to test specifications that state $\mu = \mu_0$ we formulate the null hypothesis $H_0 : \mu = \mu_0$ and test it against $H_1 : \mu \neq \mu_0$ using the t-test. This test too has one and two-tailed variants. Although the t-test requires that the sample mean is normally distributed, when the sample size is large enough, this requirement can be assumed to hold.

Another approach to perform hypothesis testing is to use sequential testing, where hypothesis testing is performed as samples are collected. The Sequential Probability Ratio Test (SPRT) [106] is a technique for selecting between two hypotheses with a minimum number of samples. SPRT maintains a likelihood ratio, updated after each sample, that rates two hypotheses based on the observed samples. Based on this ratio one hypothesis is accepted, or more samples are gathered until enough evidence is available to pick one.

Fisher's method [101] is a technique for combining the results of multiple statistical tests for the same null hypothesis. Each individual test produces a p-value for the hypothesis. Fisher's method is then used to calculate a single p-value for the entire set of tests. If this combined p-value is too low, then the null hypothesis can be rejected. Otherwise the null hypothesis cannot be rejected even if some of the individual tests failed. Fisher's method assumes that the results of the individual tests are independent, which is not always true. If the tests are dependent, using Fisher's method may result in a p-value lower (but never higher) than the correct p-value.

2.3.2 Generating Accuracy Checker Code

Based on the accuracy specification, AxProf needs to select what statistical test to use and how many samples are needed for the statistical test based on the required level of confidence. In addition, AxProf needs to decide how to aggregate output data over multiple executions or inputs and when to use the aggregated data to perform the statistical tests: after every execution, after multiple executions for the same input, or after executions on multiple inputs.

AxProf supports three main types of accuracy specifications and selects the statistical test based on the type of *ASpec* expression from the specification language and the comparison operator used in its predicate.

Probability predicates. Specifications of the form

```
Probability over Qualif [ BExpr ] rop DExpr
```

require testing the probability of satisfying the inner boolean expression (`BExpr`) against the probability `DExpr`.

Each element in the space defined by `Qualif` can be treated as one sample drawn from a Bernoulli distribution which is 1 if `BExpr` is satisfied and 0 otherwise. This allows us to use the binomial test to compare `DExpr` with the probability of the Bernoulli variable. Based on the probability space defined by the user in `Qualif`, AxProf needs to gather sample outputs of the implementation over multiple executions or inputs to calculate the fraction of elements that satisfy `BExpr`:

- If `Qualif` is `runs`, the accuracy specification defines a probability over a set of executions of the implementation on the same input. AxProf does so and calculates the fraction of executions that satisfy `BExpr`. At profiling time, AxProf sets the number of executions to the number of samples required for the binomial test (Section 2.3.3). If the developer provides multiple inputs, then the implementation is expected to pass the test for each input separately.

- If `Qualif` is `inputs`, the accuracy specification defines a probability over the inputs of the program. In this case, AxProf executes the implementation on multiple inputs and calculates the fraction of inputs for which the implementation satisfies `BExpr`. At profiling time, AxProf sets the number of inputs to the number of samples required for the binomial test (Section 2.3.3). AxProf generates the test inputs using its input generators (Section 2.4).

- If `Qualif` is a list of items (*Range*$^+$), after each execution of the implementation, AxProf calculates the fraction of items in the list that satisfy `BExpr`. AxProf performs a binomial test separately for each execution on each input. The specification should hold for each execution and input. While AxProf cannot prove this property with full certainty, it can do so with high confidence. In particular, we formulate the null hypothesis that the implementation succeeds with very high probability (0.999 or greater) and use the SPRT to estimate the number of executions and inputs that are sufficient to check this weaker property (Section 2.3.3).

In all cases, the fraction of samples (executions, inputs, or items) that satisfy `BExpr` is compared against the value of `DExpr` using the binomial test. If AxProf observes enough evidence to reject the null hypothesis, it issues a warning to the user. Depending on the comparison operator used (`rop`), AxProf chooses one of the three variants of the binomial test (greater one-tailed, lesser one-tailed, or two-tailed).

Expectation predicates. Specifications of the form

```
Expectation over Qualif [ DExpr1 ] rop DExpr2
```

require comparing the value in expression `DExpr1` against the expected value in expression `DExpr2`. AxProf gathers samples of the value of `DExpr1` over `Qualif` and calculates their mean. For large enough sample sizes, this sample mean value is an estimate of the *real* mean value and can be considered to be drawn from a normal distribution centered around the *real* mean value. This allows us to use of the t-test for comparing the sample mean against the expected value. Similar to the probability predicates, depending on the comparison operator used (`rop`), AxProf chooses one of the three variants of the t-test.

Based on the sample space defined by the user in `Qualif`, AxProf may have to gather sample outputs of the implementation over multiple executions or inputs (as in the case of the probability predicate). AxProf calculates the value of `DExpr1` for each sample and takes the mean. AxProf then compares this mean against the expected mean `DExpr2` using the appropriate t-test. The process of gathering samples is similar to the process used for probability predicates, except that AxProf calculates and gathers the number of samples required for the t-test.

Universally quantified predicates. Specifications of the form

```
forall Range+ :  ASpec
```

require that each element in `Range+` satisfy the accuracy predicate `ASpec` (a probability or expectation specification).

AxProf performs the statistical test required for the probability or expectation specification `ASpec` as described in the previous paragraphs for each individual element in `Range+`. The null hypothesis for each test is that the implementation satisfies `ASpec` for that element, the alternative being that it does not satisfy `ASpec`. This results in multiple p-values, one for each element in `Range+`.

Next, AxProf combines the p-values obtained from the individual tests into a single p-value, to test if the overall specification is satisfied. AxProf uses Fisher's method for this purpose.

2.3.3 Determining the Required Number of Samples

For statistical tests, the required number of samples depends on the test being used, the desired significance level α, the statistical power $1 - \beta$, and other test-specific parameters. The user can specify these parameters to control the rate of false positives and negatives and the runtime of AxProf.

Binomial test. The number of samples required also depends on the size of the indifference region, δ, which determines the minimum deviation from the expected probability that the test will be able to detect [107]. The minimum number of samples required to achieve the required statistical confidence is

$$\left(\frac{z_{sig}\sqrt{p_0(1-p_0)} + z_{1-\beta}\sqrt{p_a(1-p_a)}}{\delta}\right)^2 \tag{2.1}$$

where for any probability q, z_q is the critical value of the normal distribution for q. Here, z_{sig} is $z_{1-\alpha/2}$ for a two-tailed test and $z_{1-\alpha}$ for a one-tailed test.

One-sample t-test. The number of samples required also depends on the effect size $d = |\mu_0 - \mu_1|/\sigma$, the difference in the means corresponding to the null hypothesis (μ_0) and an alternative hypothesis (μ_1) divided by the standard deviation (σ) of the sample being tested. The minimum number of samples required is

$$\frac{(t_{sig} + t_\beta)^2}{d^2} \tag{2.2}$$

where for any probability q, t_q is the critical value of the student t-distribution for q. Here, t_{sig} is $t_{\alpha/2}$ for a two-tailed test and t_α for a one-tailed test.

SPRT. To calculate the minimum number of executions, AxProf uses SPRT with the minimum success probability H and the maximum failure probability L, as

$$\frac{\log(\beta) - \log(\alpha)}{\log(H) - \log(L)} \tag{2.3}$$

AxProf ensures that each individual execution passes the test.

2.3.4 Analyzing Resource Utilization

To analyze the time and memory consumption of a computation, AxProf employs curve fitting to build the most likely regression model and checks the quality of the fit. As the first step, AxProf generalizes specification expressions provided by the developer to capture the hidden constants in asymptotic notations. For example, if the specification of resource consumption is the expression $x + yz$, then AxProf will generate the general expression $p_0x + (p_1y + p_2)(p_3z + p_4) + p_5$. For this expression, AxProf searches for the values of the free variables $p_{0\ldots5}$ that best fit the data using statistical curve fitting [108]. The curve fitting procedure computes the R^2 metric, which quantifies how well the model fits the observed data. Higher R^2 values indicate better fitting models. AxProf triggers a warning if the R^2 metric is too low.

2.4 INPUT GENERATION

To generate random inputs to test the computations, AxProf can use one of three input generators, each of which can control different aspects of the generated data:

- *Scalar generator:* This generator can be used to generate a list of integers or real numbers. It can sample the numbers from a variety of distributions with controllable parameters. It also provides control over the ordering of numbers, or the difference between adjacent numbers.
- *Matrix generator:* This generator can be used to generate a matrix of given dimensions with random elements. In addition to the parameters from the scalar generator, this generator also provides control over matrix-specific properties such as sparsity.
- *Vector generator:* This generator can be used to generate a list of vectors with a given number of dimensions. This generator also provides control over the same parameters as the scalar generator. In particular, it provides control over the distance between vectors measured by various metrics.

AxProf also allows the developer to use specific random seeds to enable reproducibility. AxProf can be easily extended with additional input generators, or the developer can specify a custom input generator.

Automatically selecting input generator features. Identifying what input features affect the accuracy of a program is important to selecting a input generator and deciding what inputs to test. Each generator described above has input features that can be used to control the generated inputs.

We adapt a technique from [109] to select important input features that need to be explored. We use the *Maximal Information Coefficient* (MIC) [110] to identify relationships between input features and the accuracy of a program. For each input feature available in an input generator, we perform sample executions of the program implementation while varying that input feature and calculate the accuracy of the output. We use this data to calculate a MIC value. If the MIC value is above a certain threshold, we accept that the input feature affects output accuracy.

Table 2.1: Summary of the AxProf benchmarks. Unless specified otherwise, n is the size of the dataset.

Benchmark	**Parameters**	**(Informal) accuracy spec**
Locality Sensitive Hashing	k: hashes per table l: number of hash tables	$P[\mathit{neighbor}] = 1 - (1 - p^k)^l$ p depends on similarity
Bloom Filter	p: max false positive probability c: capacity	$P[\mathit{false\ positive}] \leq p$ If number of inserted elements $< c$
Count-Min Sketch	ϵ: error factor δ: error probability	$P[\mathit{error} < n\epsilon] > 1 - \delta$
HyperLogLog	k: Number of hashes	$P[\mathit{error} \leq 1.04/\mathrm{sqrt}(k)] >= 0.65$
Reservoir Sampling	s: reservoir size	$P[\mathit{in\ sample}] = \min(s/n, 1)$
Matrix Multiply	c: sampling rate A: $m \times n$ matrix B: $n \times p$ matrix	$P[\lVert \mathit{error} \rVert_F < C] > 1 - \delta$ $C = \eta/c\lvert\lvert A\rvert\rvert_F\lvert\lvert B\rvert\rvert_F$ where $\eta = 1 + \sqrt{-8\log(\delta)}$
Chisel/blackscholes	r: reliability factor	$P[\mathit{exact} = \mathit{approx}] > r$
Chisel/sor	r: reliability factor m, n: matrix dimensions i: iterations	$P[\mathit{exact} = \mathit{approx}] > r$
Chisel/scale	s: scale factor r: reliability factor	$E[\mathit{PSNR}(d, d')] \geq -10 \cdot \log_{10}(1 - r)$

2.5 METHODOLOGY

Table 2.1 presents a summary of the benchmarks we analyzed with AxProf. We chose these benchmarks to represent common randomized approximate components of data-centric applications as well as computations running on unreliable hardware. The table lists the benchmark parameters that can be controlled (Column 2) and the informal accuracy specifications (Column 3). Table 2.2 presents the accuracy specifications for each benchmark specified using AxProf's specification language.

2.5.1 Benchmark Details

Locality Sensitive Hashing (LSH). We describe this algorithm in Section 2.1.

Bloom Filter. Bloom Filter [19] checks if an item was present in a data stream. It starts with an all-zero bit filter of length m. For each data item, it calculates k different m-bit hash functions and sets the corresponding bits. To check if an item q was in the stream, the algorithm checks that each bit corresponding to the k hashes of q is 1. The user calculates optimal values for k and m from the desired capacity c and a maximum false positive rate

Table 2.2: Accuracy specifications provided to the checker generator

Benchmark	**Accuracy specification in the AxProf language**

Locality Sensitive Hashing	`forall i in indices(Input), q in indices(Input) :`
	`  Probability over runs [ [q,i] in Output ]`
	`    == L1HashEqProb(Input[i],Input[q],k,l)`

Bloom Filter	`Probability over i in excluded(Config,Input)`
	`  [ i in Output ] < p`

Count-Min Sketch	`Probability over i in uniques(Input)`
	`  [ (count(i,Input) - Output[i])`
	`    < `ϵ`*|Input| ] > 1 - `δ

HyperLogLog	`Probability over inputs`
	`  [ abs(datasize-Output)`
	`    < (datasize*1.04)/sqrt(2^k) ] >= 0.65`

Reservoir Sampling	`forall i in Input :`
	`  Probability over runs [ i in Output ]`
	`    == min(ressize/datasize,1)`

Matrix Multiply	`Probability over runs`
	`  [ Output < (`$\eta(\delta)$`)/samplingFactor)`
	`    *(frobNorm(Input[0])*frobNorm(Input[1])) ] > `δ

Chisel/blackscholes	`Probability over runs`
	`  [ Output == oracle(Config,Input) ] > r`

Chisel/sor	`Probability over runs`
	`  [ Output == oracle(Config,Input) ] > r`

Chisel/scale	`Expectation over runs`
	`  [ PSNR(Input, Output) ] >= -10*log10(1-r)`

p. In the specification in Table 2.2, `excluded` calculates the set of items in the input that were not inserted into the filter.

Count-Min Sketch. Count-Min Sketch [17] counts the frequency of items in a dataset. The algorithm maintains a set of uniform hash functions whose range is divided into a set of bins. For every item in the dataset, it calculates the hash functions and increments the counters in the mapped bins. The estimated frequency of an item is the minimum of all the counters in the corresponding bins. The accuracy can be improved by increasing the number of hash functions and bins. In the specification in Table 2.2, `uniques` calculates the set of unique items in the input, as some items appear multiple times.

HyperLogLog. HyperLogLog [15] is an algorithm for calculating the number of distinct elements in a large dataset. For each element in a dataset, the algorithm calculates k hash values. It uses the hash values with the maximum number of leading zeros to estimate the cardinality of the dataset. The user can reduce the variance of the result by increasing k.

Reservoir Sampling. Reservoir Sampling [18] uniformly samples a data stream of unknown length. For a reservoir of size s, the first s elements are directly inserted into the reservoir. Afterwards, for the i^{th} element to be inserted, it chooses an integer p uniformly at random from $[1, i]$. If $p \leq s$ then it replaces the item currently in the p^{th} position in the reservoir with the new item.

Randomized Matrix Multiplication. Randomized matrix multiplication methods [20] reduce computation time and resource consumption of matrix multiplication by randomly sampling the matrices. The authors provide guarantees for accuracy as an upper bound on the Frobenius norm of the errors. Users can control the error by changing the sampling rate.

Chisel kernels. Chisel [36] is a reliability aware optimization framework for programs that run on approximate hardware. We examine three kernels from Chisel's benchmarks: 1) *scale* scales an image by a specified scale factor, 2) *sor* performs the successive over-relaxation operation for a given matrix, and 3) *blackscholes* computes the price of a stock portfolio using the Black-Scholes formula. For *blackscholes* and *sor*, Chisel provides bounds on the probability that the output differs from the exact value. For *scale*, Chisel provides the expression for the expected PSNR value between the exact and approximate results.

2.5.2 Benchmark Implementations

For each algorithm benchmark, we selected two implementations from GitHub. We preferred implementations in Java, Python, or C/C++ due to our familiarity with those languages. We took several factors into account when selecting implementations to profile, such as the number of stars on GitHub and repository activity. All the selected implementations appear among the top ten search results in GitHub for the name of the algorithm. For the Chisel kernel benchmarks, we used implementations derived from those used in the evaluation of Chisel.

2.5.3 Experimental Setup

Parameters and their ranges. The parameter value choices offer a trade-off between profiling time and the confidence in the benchmark output's correctness. We chose param-

Table 2.3: Controlled parameters

Benchmark	**Parameters**	**Range**
Locality Sensitive Hashing	Hash functions per table	2, 4, 8
	Number of hash tables	2, 4, 8, 16
	Input size (performance)	1000, 2000, . . . , 10000
	Input size (accuracy)	100
Bloom Filter	Max. false positive prob.	0.1, 0.01, 0.001
	Load (fraction of capacity)	0.2, 0.4, 0.6, 0.8
	Capacity (performance)	20000, 40000, . . . , 100000
	Capacity (accuracy)	200, 400, . . . , 1000
Count-Min Sketch	ϵ : error factor	0.1, 0.01, 0.001
	δ : error probability	0.2, 0.1, 0.05
	Input size	10000, 20000, . . . , 100000
HyperLogLog	Number of hash function	2^8, 2^{10}, 2^{12}, 2^{14}
	Unique items in input	10000, 20000, . . . , 100000
Reservoir Sampling	Size of reservoir	10000, 20000, . . . , 100000
	Input: size	10000, 20000, . . . , 100000
Matrix Multiply	Size of matrices	20×20 - 200×200
	Sampling rate	0.2, 0.4, 0.8
Chisel/blackscholes	Load error rate	0.000048, 0.00024, 0.24
Chisel/sor	Size of matrices	4×4 - 50×50
	Iterations	1, 5, 10, 20
	Omega factor	0.1, 0.5, 0.75
Chisel/scale	Input image	5 images
	Scale factor	1, 2, 4, 8

eters across their valid range. For each parameter in the time or memory specifications, we used at least 3 values across the range (for curve fitting). Table 2.3 presents the parameter ranges we used in our experiments. Column 2 shows the parameter names, and Column 3 shows the range of each parameter.

Statistical tests. For statistical tests, we used the standard significance level $\alpha = 0.05$ and statistical power $1 - \beta = 0.8$ when calculating the number of executions. To get the number of executions for per-execution checkers, we used SPRT with a minimum acceptable probability of success of 0.999 and a maximum probability of failure of 0.99. Based on these values, AxProf calculated that 320 executions were sufficient. For the specifications requiring the binomial test, we chose a probability deviation factor $\delta = 0.1$. For those that require the t-test, we chose an effect size $d = 0.2$. For both of these, AxProf calculated (using the formulae from Section 2.3.3) that 190-200 executions were sufficient. For identifying resource model discrepancies, we used an R^2 threshold of 0.9.

Table 2.4: Summary of AxProf profiling results

Benchmark	**Implementation**	**Accuracy**	**Time**	**Memory**
Locality Sensitive Hashing	*TarsosLSH* [102]	WARN (12/12)	✓	✓
	java-LSH [112]	WARN (4/4)	✓	N.A.
Bloom Filter	*libbf* [113]	✓	WARN	✓
	BloomFilter [114]	✓	WARN	✓
Count-Min Sketch	*alabid* [115]	WARN (90/90)	✓	✓
	awnystrom [116]	WARN (81/90)	✓	WARN*
HyperLogLog	*yahoo* [117]	✓	WARN	WARN
	ekzhu [118]	WARN* (2/40)	✓	✓
Reservoir Sampling	*yahoo* [117]	✓	✓	WARN
	sample [119]	✓	WARN*	✓
Matrix Multiply	*RandMatrix* [120]	WARN (30/243)	✓	✓
	mscs [121]	✓	✓	✓
Chisel/blackscholes	Chisel [122]	✓	✓	✓
Chisel/sor	Chisel [122]	✓	✓	✓
Chisel/scale	Chisel [122]	✓	✓	✓

Environment. We ran experiments on a machine with a Xeon E5-1650 v4 CPU, 32 GB RAM, and Ubuntu 16.04. For time profiling, we used a real-time timer around the relevant functions. For memory, we used serialization in Java and Python, and the `time` Linux utility for C/C++ programs. For fitting the resource consumption models, we used the `scipy.optimize` module of SciPy [111].

2.6 EVALUATION

2.6.1 Effectiveness of AxProf in Profiling Accuracy

Table 2.4 presents a summary of our findings using AxProf. Columns 1-2 present the benchmark and the profiled implementation. Column 3 presents the results for accuracy analysis of each implementation. Columns 4 and 5 present the results for analysis of time and memory usage. In each column, a ✓ represents that AxProf did not find any issues, while WARN(X/Y) represents cases where AxProf issued a warning in X out of Y tested benchmark configurations. A * indicates that we determined a warning to be a false warning. For accuracy analysis, we tested each configuration independently. For time and memory analysis, data from all configurations is part of a single regression model.

Out of the 15 implementations that we profiled, 4 implementations passed all tests for compliance with accuracy, time and memory specifications. AxProf detected conditions that trigger warnings in the remaining 11 implementations. We manually analyzed the implementations that caused warnings. We found two causes for the 9 *real* warnings:

- *Errors in implementations:* Some implementations used hash functions with errors that caused higher than expected accuracy loss. One implementation had a misconfigured random number generator that affected sampling.
- *Performance optimizations:* Some implementations added performance optimizations that caused unexpected time or memory usage.

We observed false warnings (WARN*) in the time specification check for *sample* and the memory check for *awnystrom* due to noise in the measurements. For HyperLogLog, when the input-set cardinality is very low and close to a predefined threshold, the error in the accuracy estimate provided by the specification is high. This led to warnings for two configurations of *ekzhu* (*yahoo* avoided this issue by using a polyalgorithm [123]).

2.6.2 Errors in Implementations

LSH: *TarsosLSH.* We discuss this benchmark in detail in Section 2.1.

LSH: *java-LSH.* This is a MinHash-LSH implementation in Java for the Jaccard similarity metric [124]. We observed that sets were being considered similar to the query set more often than expected. The implementation used the simple hash function $h(x) = (a * x + b)$ mod m. This hash function is usable only when m is prime. However, the implementation often chose a composite value for m. We fixed the hash by setting m to a fixed, large prime. After this fix, the observed results matched the expected values.

Count-Min Sketch: *awnystrom*, *alabid.* Count-Min Sketch calculates multiple hashes chosen randomly from a family of hash functions. The output of the hash functions must be *pairwise independent* with respect to the hashed values. That is, if h is a function chosen uniformly at random from the hash family, for two values x and y, $h(x)$ and $h(y)$ must be uniformly distributed and pairwise independent.

AxProf detected that the observed error rate was higher than expected for many configurations in both implementations. By manual inspection, we identified that the buggy implementations used hash functions that do not satisfy the pairwise independence property. After replacing the hash functions, the error rate reduced to the expected level.

Matrix Multiplication: *RandMatrix*. This method subsamples the rows and columns of the matrices to reduce matrix multiplication time. The algorithm [20] provides an optimum sampling method that was not implemented correctly in the implementation, leading to incorrect results.

Developer-provided tests. In all cases, tests written by the developers failed to catch the bugs identified through AxProf. We observed three main reasons: 1) unit tests only partially checked the algorithm functionality (*java-LSH*), 2) tests used fixed inputs that did not trigger bugs (*alabid*, *awnystrom*, *RandMatrix*), and 3) the test framework itself was buggy (*TarsosLSH*).

2.6.3 Performance Optimizations

We also observed situations where warnings were issued by AxProf due to performance optimizations in implementations that were causing unexpectedly low resource usage. For both Bloom Filter implementations, instead of checking the entire filter to search for a 0 value, the implementations could return immediately when the first 0 was found. This property is not encoded in the basic algorithm accuracy specification. For the *yahoo* HyperLogLog implementation, AxProf was unable to model the runtime of the algorithm against the input size due to the use of a polyalgorithm [123]. For the *yahoo* Reservoir Sampling implementation, the memory usage was unexpectedly low due to the implementation incrementally allocating memory as opposed to doing so all at once.

2.6.4 Effectiveness of Accuracy Profiling

We analyzed the accuracy bugs that we found and confirmed that they did not affect the resource usage of the implementation. Table 2.4 shows how accuracy bugs rarely correlate with deviations in resource consumption. These results show the importance of augmenting regular algorithmic profiling techniques with accuracy analysis.

2.6.5 Effectiveness of Input Feature Selection

We studied four input features from AxProf's input generators and their effect on the accuracy of the algorithm benchmarks. Table 2.5 shows the results of the analysis. We only examined the correct implementations of the programs. We analyzed the benchmarks manually to confirm the results of the MIC-based approach (Section 2.4). Columns 2-4

Table 2.5: Input feature impact on accuracy

Benchmark	Order	Frequency	Distance	Sparsity
Locality Sensitive Hashing	(✗, ✗)	(✗, ✗)	(✓, ✓)	-
Bloom Filter	(✗, ✗)	(✓, ✓)	(✗, ✗)	-
Count-Min Sketch	(✗, ✗)	(✓, ✓)	(✗, ✗)	-
HyperLogLog	(✗, ✗)	(✗, ✗)	(✓, ✗)	-
Matrix Multiply	(✓, ✓)	-	(✓, ✓)	(✓, ✓)

correspond to different input features. Each column has the format (*automatic/manual*). A ✓ indicates that AxProf's automatic and our manual analyses, respectively, show that the input feature affects accuracy. For Reservoir Sampling, we were unable to derive an accuracy measurement due to the nature of the specification. For the Chisel benchmarks, the accuracy depends only on the underlying hardware, therefore input features we changed did not have any impact. AxProf's automatic MIC-based approach correctly identified most input features that were relevant to the algorithm's output accuracy.

2.7 RELATED WORK

Algorithmic profiling. Researchers have proposed several approaches to model performance of programs as a function of workload size [96, 97, 98]. Algorithmic profiling [98] is a framework that focuses on automating such profiling tasks by detecting algorithms and their inputs. COZ [125] is a causal profiler that estimates the effect of potential optimization of subcomputations on the performance of the whole program. Researchers proposed similar techniques for analyzing memory and recursive data structures, e.g., [126, 127]. AxProf's accuracy analysis is complementary to these existing approaches. Statistical debugging and profiling tools, e.g., [128, 129, 130] use statistical models of programs to predict and isolate bugs. This line of research is conceptually orthogonal to ours.

Analysis of accuracy. Researchers have explored various dynamic approaches [37, 49, 50, 61, 62] to empirically analyze the impact on accuracy from transformations that change program semantics. In contrast, AxProf uses theoretical specifications of uncertain computations and checks for discrepancies in their implementations. MayHap [91] converts program code to a Bayesian network and uses Chernoff bounds to check probabilistic assertions over a set of executions. In contrast, AxProf operates on a program as a black-box system, supports a richer set of predicates including inputs and items, and automatically selects the appropriate test. Ceramist [131] is a Coq framework for verifying accuracy specifications of *theoretical* approximate randomized algorithms. AxProf uses statistical tests to ensure that *implementations* of the algorithms satisfy the accuracy specifications.

Statistical model checking. Statistical model checking [107, 132, 133] is a general approach for verifying properties of black-box stochastic systems using statistical hypothesis testing. For instance, the framework of Sen et al. [132] expresses properties in Continuous Stochastic Logic (CSL) and samples the outputs from the tested system. CSL's probability predicate, like our `Probability over runs` predicate, estimates the probability that the system satisfies a specified logical property. However, expressing the predicates over inputs and items would be significantly more complicated in CSL. Additionally, it does not support expectation predicates or complex data structures, like lists or matrices. In addition to supporting these features, AxProf automatically generates code for collecting and aggregating data, thus giving a developer an intuitive tool to simultaneously explore various aspects of an implementation's accuracy and resource consumption.

Applications of AxProf. Fernando et al. [134] used autotuning to successfully relax conservative approximate algorithm parameters for restricted sets of inputs. We used AxProf within the autotuning process to ensure that candidate relaxed parameters still satisfied the algorithm's accuracy specification. Aloe (Chapter 4) statically analyzes computations running on unreliable hardware that attempt to recover from errors using potentially imperfect checker functions. Some imperfect checkers rely on assumptions about the computation's input in order to effectively detect errors [74, 135]. If a new set of inputs potentially violates these assumptions, AxProf can statistically check the protected computation to determine if the current imperfect checker can be used, or if a new checker is required.

Chapter 3: GAS

Autonomous vehicles, such as self driving cars, unmanned aircraft, and utility vehicles, are increasingly common. They navigate by perceiving the vehicle's state (position, heading, etc.), making a control decision based on this *perceived* state, and moving accordingly. To preserve life and property, developers specify safety properties that the vehicle must satisfy in certain scenarios. However, 1) many sensors have a nondeterministic output (e.g., GPS and LIDAR) and 2) the vehicle's software processes sensor values or makes decisions using complex, possibly imperfect components such as neural networks and lookup tables. Given this uncertainty in the output of the perception and control systems, proving that an autonomous vehicle will *never* violate safety properties in a scenario is impossible or impractical. Developers instead focus on proving that the vehicles will satisfy the safety property *with high probability*, using probabilistic and statistical techniques.

Monte Carlo Simulation (MCS) is perhaps the most used method for checking safety properties during development of vehicle systems. Simulations are capable of detecting software faults that would otherwise require much more expensive formal verification or real-world testing and are commonly used in industry for testing autonomous vehicles [136, 137]. However, using MCS still requires a large number of resource-intensive system simulations in order to get a sufficiently accurate estimate of the probability that the vehicle will violate a safety property [136, 138, 139]. This is particularly worrisome as the vehicle system requires constant retesting as it undergoes changes during its development. Even after initial development has ceased and the vehicle is deployed, developers may still make changes to it to add new capabilities and improve navigation and safety in real-world environments.

Surrogate Models aim to provide an accurate and faster replacement for the original costly model of many engineering systems. These models can be created using techniques such as abstraction refinement [138], machine learning [140], or Generalized Polynomial Chaos (GPC) [141]. For instance, previous research has created GPC surrogate models for the equations of vehicle dynamics [142]. However, the presence of complex perception and control components (which often dominate the original model's execution time) hinders the application of existing surrogate model construction techniques to creating surrogate models of the complete vehicle system. For example, the output of a neural network which processes a camera image to perceive the vehicle's state is not only affected by the ground truth state, but also multiple environmental parameters (e.g., weather, lighting, nearby objects, etc.) which may affect the neural network's output in a nondeterministic manner. Existing techniques struggle to capture this complicated relationship between environmental parameters and the neural network output.

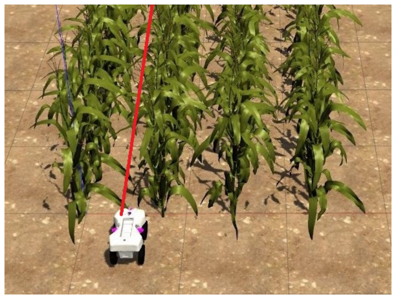
Our work. We present GAS (**G**PC for **A**utonomous Vehicle **S**ystems), the first approach for creating surrogate models of complete autonomous vehicle systems which compose complex perception and/or control components with vehicle dynamics. The resulting surrogate models are close approximations of the original vehicle models. They provide a faster alternative to MCS when developers experiment with system components, or tune various system parameters during the design and testing stages of vehicle development.

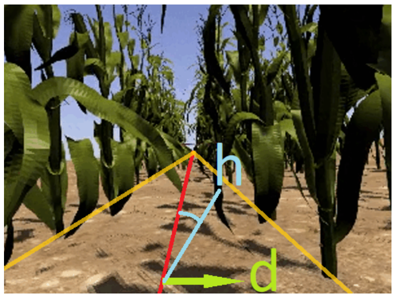
GAS first creates a *perception model* to calculate the *distribution* of error in the output of the perception system for any ground truth state. GAS directly samples this error distribution, reducing the need for costly experimentation with environmental parameters, image generation, and neural networks. Second, GAS constructs a surrogate model of the *complete* vehicle system (perception, control, and dynamics). GAS can create several types of surrogate models, such as polynomial models via regression, or neural network models. However, we observed that GAS produces the most consistently precise surrogate models using GPC. GAS also supports systems that contain categorical variables (e.g., in the control system) using an ancillary model, overcoming a limitation of GPC. Lastly, because GAS's perception model is created independently of downstream components, it can be reused when designers alter vehicle control and dynamics properties of the vehicle, potentially saving a significant amount of time. We can use GAS instead of the original vehicle model to calculate any property of the vehicle system which can be calculated by analyzing the evolution of the vehicle state distribution over time. We demonstrate the advantages of using the GAS-generated surrogate model to calculate two such properties:

First, we use the surrogate model to *estimate the probability that the vehicle will reach an unsafe state over time* in five realistic scenarios. These scenarios model systems used in crop management vehicles, self driving cars, and unmanned aircraft. Each system uses a complex perception component (ResNet-18 or LaneNet) or a complex control component (neural network controllers or lookup tables). We show that the probability of reaching an unsafe state calculated by the surrogate model closely matches that calculated via the original model for 97% or more of time steps, while being 3.7× faster on average (minimum 2.1×). We also investigate how various hyperparameters of the perception model affect GAS's overall accuracy and speedup.

Second, we use the surrogate model for *global sensitivity analysis of the vehicle system* to initial state perturbations for the same scenarios by calculating Sobol sensitivity indices [143] 1.4× faster on average (minimum 1.3×) with an average error of 0.0004 (maximum 0.06).

(a) Real-life vehicle


(b) Sim: overhead view



(c) Sim: front camera

Figure 3.1: Crop-Monitor vehicle scenario

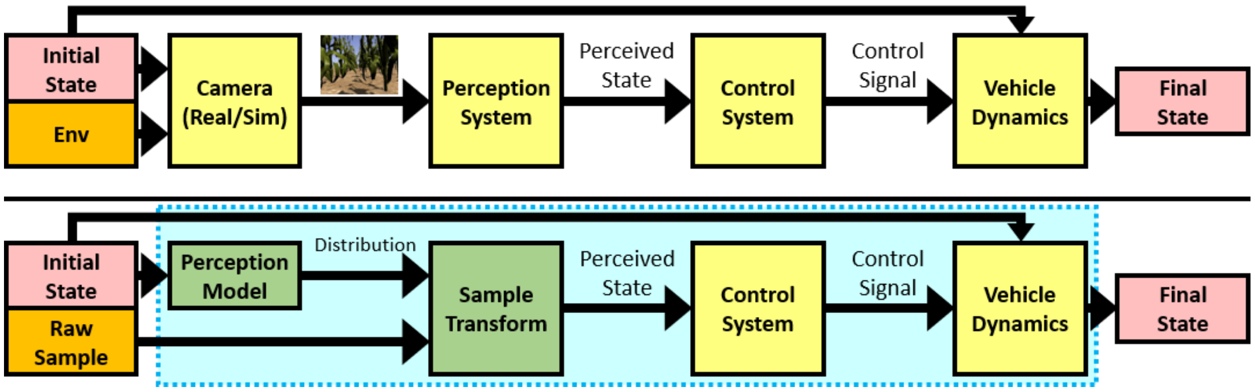


Figure 3.2: Crop-Monitor vehicle model: original (top) and abstract (bottom). GAS *replaces* the outlined section with a surrogate model.

3.1 EXAMPLE

Consider an autonomous vehicle which travels between two rows of crops in order to monitor crop growth and detect weeds. Farmers use this information to adjust fertilizer and herbicide levels for each location. We have adapted this scenario from [144]. Figure 3.1 illustrates the scenario. The desired path is shown in red.

The top half of Figure 3.2 shows a block diagram representation of the system model M_V responsible for driving the vehicle between two rows of crops. First, a camera captures the area in front of the vehicle. The image depends on the current vehicle state as well as environmental conditions. Environmental variables include crop types, crop growth stage, crop model, and lighting conditions. The neural network analyzes the image to perceive the current vehicle state. It is a regression neural network, producing a single state prediction. The relevant state variables in this scenario are:

- The *heading* angle h, which is the angle between the vehicle's current heading and the imaginary center line between the two rows of crops. h can take values in $[-\pi, \pi]$

radians, with 0 corresponding to the direction of the center line.

- The *distance* d of the vehicle from the center line. d can take values in $[-0.38, 0.38]$ meters, with 0 corresponding to the center line.

The vehicle state space is therefore $\mathbb{D}_S = [-\pi, \pi]_h \times [-0.38, 0.38]_d$.

As neural networks are inherently approximate, the state perceived by the neural network may not be the same as the ground truth. The vehicle uses this approximate heading and distance reading to calculate a steering angle in order to keep the vehicle on the center line. Finally, the vehicle moves according to its constant speed and commanded steering angle.

Unsafe states. We wish to avoid two undesirable outcomes: 1) if $|d| > 0.228m$, the vehicle will hit the crop stems, and 2) if $|h| > \pi/6$, the neural network output becomes highly inaccurate and recovery may be impossible. To avoid these outcomes, we want the vehicle to remain in the *safe region*, defined as all states within $\mathbb{D}_S^{safe} = [-\pi/6, \pi/6]_h \times [-0.228, 0.228]_d$. Because the vehicle makes steering decisions based on approximate data, we cannot be certain that the vehicle will remain safe. Instead, we want to calculate *the probability that the vehicle will remain safe over time.*

Monte Carlo Simulation. In *Monte Carlo Simulation* (MCS), we simulate the vehicle's movement a large number of times and count the number of times the vehicle reaches an unsafe state. We use the Gazebo simulator [145] to precisely control the simulation.

We simulate 1,000 samples over 100 time steps of 0.1 seconds each. We randomly sample environmental conditions from an environment distribution $\mathcal{D}_E$ that contains two types of crops, four crop growth stages, crop model variations, and a range of lighting conditions. We also choose the initial vehicle state from a normal distribution within $\mathbb{D}_S^{safe}$. For each sample at each time step, we invoke M_V, which 1) captures an image from an expensive Gazebo simulation in the current vehicle state and environment, 2) passes the image through the neural network to get the approximate perceived state, 3) calculates a steering angle based on the *perceived* state, and 4) calculates the position of the vehicle after one time step using the *actual* state and steering angle. At each time step, we calculate the fraction of samples that are still in a safe state.

3.1.1 GAS: Using Generalized Polynomial Chaos

We present GAS, a novel approach for creating surrogate models of complex vehicle systems. Here, we show an example construction using Generalized Polynomial Chaos (GPC).

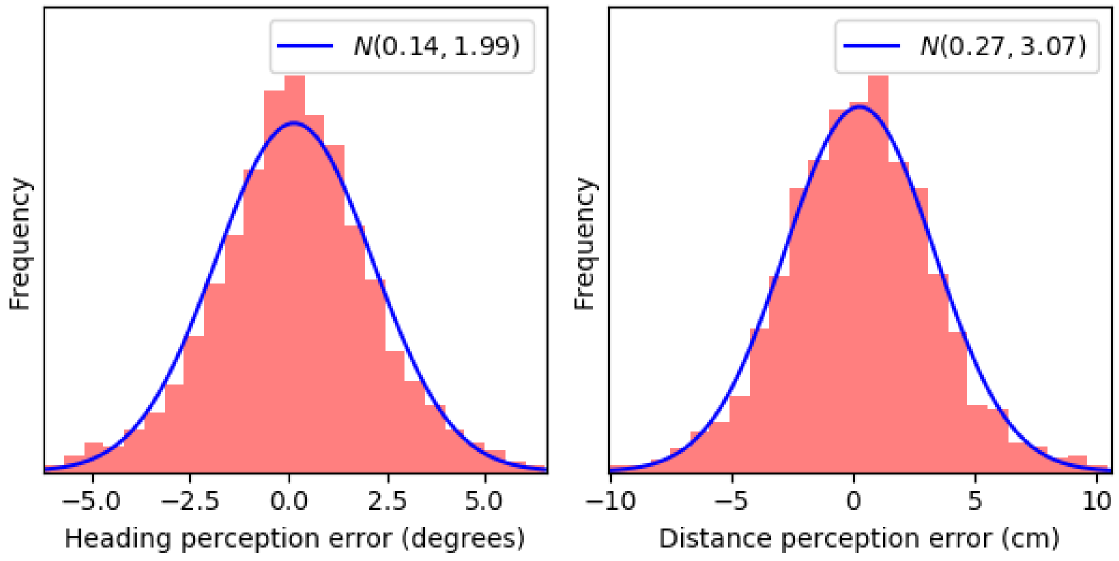


Figure 3.3: Error distribution of the ResNet-18 neural network for the validation set of *real-world* images.

While previous research (e.g., [142]) has explored using GPC to create models of vehicle *dynamics*, this is insufficient in our scenario as the process of capturing and processing the image contributes to over 99% of the simulation time. GAS aims to instead use GPC to create a surrogate for the *entire* vehicle model: perception, control, and dynamics.

Perception model construction. The output of the vehicle's perception neural network depends on the image captured by the front camera. This image depends not only on the vehicle's state, but also environmental parameters (e.g., lighting, crop type, crop age, etc.). Given a distribution of environmental parameters, there is a corresponding output distribution of the perception neural network for each ground truth state. Figure 3.3 shows the error distribution of the ResNet-18 networks used by this vehicle on the original validation dataset of *real-world* images collected by the vehicle's camera in a cornfield. The histogram shows the actual error frequency, while the line shows the fitted normal distribution. The distribution closely matches the histogram, indicating that the error of the neural network is normally distributed.

GAS must sample this output distribution when creating the GPC model. However, 1) the environmental parameters have a smaller effect on the perceived state as compared to the vehicle's actual state, and 2) directly using the numerous environmental parameters increases the input space over which GAS must construct the GPC model, which increases model construction time. GAS therefore abstracts away the actual environmental parameters by creating a *perception model*. GAS trains the perception model by selecting an 11×11

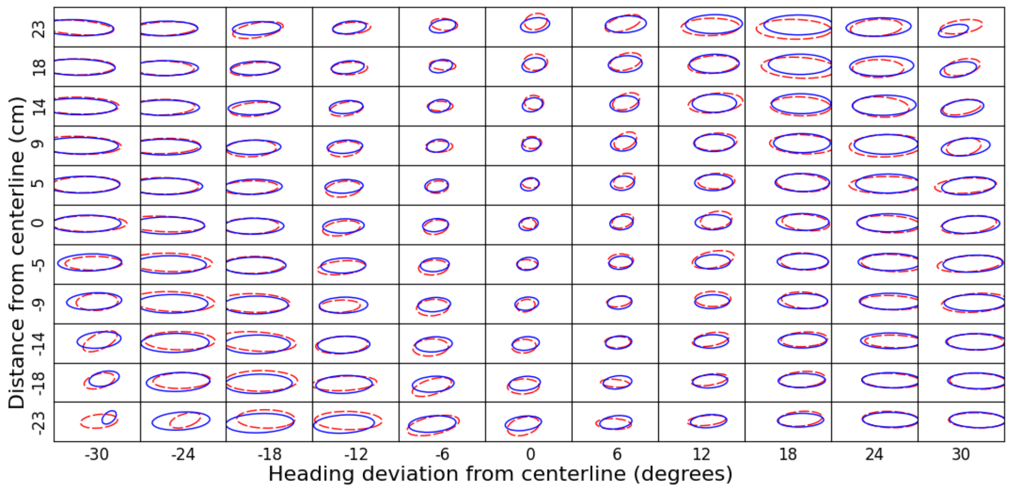
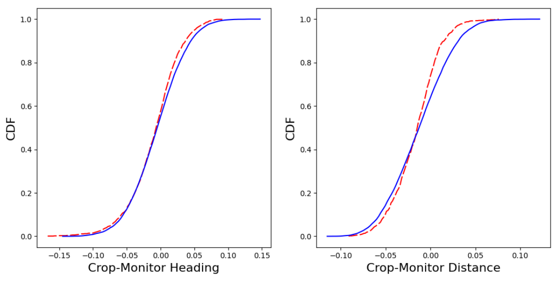


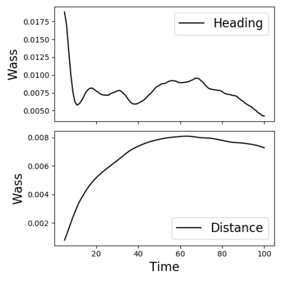
Figure 3.4: Comparison of neural network output distribution (dashed red ellipse) to distribution predicted by perception model (solid blue ellipse). Each box represents a distinct ground truth state used to construct the perception model. The X and Y-Axes vary the ground truth heading and distance, respectively.

grid of ground truth states in $\mathbb{D}_S^{safe}$. At each grid point (h, d), it randomly samples multiple images from $\mathcal{D}_E$, and records the neural network outputs. It calculates the mean $\mu(h, d)$ and covariance $\sigma^2(h, d)$ of the outputs at each state. GAS trains a degree 4 polynomial regression model M_{per} to predict each component of $\mu(h, d)$ and $\sigma^2(h, d)$ at any safe state. Figure 3.4 visually compares the neural network output distribution to the distribution predicted by M_{per} for images captured within Gazebo. The red dashed ellipse and the blue solid ellipse show the 3σ confidence boundaries for the neural network output distribution and the distribution predicted by the perception model, respectively. The two distributions closely match, especially when the vehicle is near the center and pointing straight ahead.

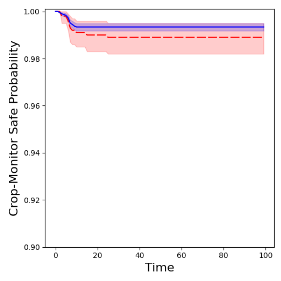
GPC surrogate model construction. Next, GAS creates an abstract vehicle model M'_V, shown in the bottom half of Figure 3.2. M'_V first uses M_{per} to obtain the neural network output distribution in the current state. Second, it transforms a sample from a 2D standard normal distribution into a sample from this output distribution using matrix operations. The rest of the vehicle model uses this transformed sample as the perceived state. GAS now uses GPC to create a polynomial model of the complete system (outlined section of Figure 3.2). This model M_{GPC} is a 4^{th} degree polynomial over 4 variables (2 state variables and a 2D normal distribution sample). GAS *replaces* M_V in the MCS procedure outlined above with M_{GPC} and uses it to estimate the state distribution of the vehicle over time. GAS also increases the number of samples for distribution estimation to 10,000 for M_{GPC} in order to take advantage of its speed and decrease sampling error.



(a) Final state distributions: heading (left) and distance (right)



(b) Wasserstein metric value over time



(c) Safe state probability over time

Figure 3.5: Visual comparison of GAS and MCS results for Crop-Monitor

3.1.2 Results

Accuracy. Figure 3.5a shows a comparison of the heading and distance distributions after 100 time steps. The X Axis shows the variable value and the Y Axis shows the cumulative probability. The blue solid and red dashed plots show the distributions estimated using GAS and MCS, respectively. We compare the GAS and MCS distributions using the Kolmogorov-Smirnov (KS) statistic and the Wasserstein metric. Figure 3.5b shows how the Wasserstein metric evolves over time. The X Axis shows time steps and the Y Axis shows the Wasserstein metric. The low Wasserstein metric value indicates good correlation between the two distributions at all times. The KS statistic also remains below 0.14. Figure 3.5c shows the probability of remaining in a safe state over time. The X Axis shows time steps and the Y Axis shows the probability of remaining safe. The blue solid and red dashed plots show the probabilities calculated using GAS and MCS, respectively. The shaded regions show the 95% bootstrap confidence interval. Around each plot, the bootstrap confidence interval indicates the variation that can occur as a result of sampling error. As GAS evaluates M_{GPC} for 10× more samples than M_V, its confidence interval is smaller. We use the t-test to check if the safe state probabilities are similar: it passes for 99 of 100 time steps, indicating high similarity between the estimated probabilities.

Time. GAS is 2.3× faster than MCS on our hardware. MCS using M_V required 19.5 hours. Gathering the training data for M_{per} required 8.5 hours. The time required for training M_{per}, constructing M_{GPC}, and using M_{GPC} was negligible in comparison ($<$ 1 minute). Increasing the number of samples or the number of time steps would further increase the gap between the two methods, since the time required to construct the perception model is fixed.

3.2 BACKGROUND

We present key definitions pertaining to the construction of GPC models. Dedicated books (e.g., [141]) provide more detailed descriptions.

Orthogonal polynomials. Assume X is a continuous random variable with support S_X and probability density $p_X : S_X \to \mathbb{R}^+$. Let $\Psi = \{\Psi_n | n \in \mathbb{N}\}$ be a set of polynomials, where Ψ_n is an n^{th} degree polynomial. Then Ψ is a set of orthogonal polynomials for X if:

$$n \neq m \Rightarrow \int\limits_{S_X} \Psi_n(x)\Psi_m(x)p_X(x)dx = 0 \tag{3.1}$$

The orthogonal polynomial Ψ_n has n distinct roots within S_X. Orthogonal polynomials exist for a wide range of probability distributions. For example, the Legendre, Hermite, Jacobi, and Laguerre polynomials are orthogonal over the uniform, normal, beta, and gamma distributions, respectively.

Orthogonal polynomial projection (GPC). Let $f : S_X \to \mathbb{R}$. Then the N^{th} order orthogonal polynomial projection of f, written as f_N, with respect to a set of orthogonal polynomials Ψ, is:

$$f_N = \sum_{i=0}^{N} c_i \Psi_i \quad where \quad c_i = \frac{\int_{S_X} f(x)\Psi_i(x)p_X(x)dx}{\int_{S_X} \Psi_i^2(x)p_X(x)dx} \tag{3.2}$$

If f is a polynomial of degree at most N, then $f_N = f$. Otherwise, f_N is the *optimal* N^{th} degree polynomial approximation of f with respect to X, in the sense that it minimizes ℓ_2 error, which is calculated as:

$$\ell_2 \ error = \int\limits_{S_X} (f(x) - f_N(x))^2 \, p_X(x)dx \tag{3.3}$$

As $N \to \infty$, the ℓ_2 error approaches 0, i.e., we can construct arbitrarily good approximations of f. f_N is called the *generalized polynomial chaos (GPC) approximation of order N.*

Lagrange basis polynomials. Given N points (x_i, y_i), $1 \leq i \leq N$, where all x_i are distinct, Equation 3.4 shows the *Lagrange basis polynomials* L_i for each i.

$$L_i(x) = \prod_{\substack{1 \leq j \leq N \\ j \neq i}} \frac{x - x_j}{x_i - x_j} \tag{3.4}$$

Gaussian quadrature. To use Equation 3.2, we must perform multiple integrations to calculate the coefficients c_i ($i \in \{0 \ldots N\}$). For any non-trivial function g, we must use numerical integration. We can approximate the integral as follows:

$$\int_{S_X} g(x) p_X(x) dx \approx \sum_{i=1}^{N} w_i g(x_i) \quad \textit{where} \quad w_i = \int_{S_X} L_i(x) p_X(x) dx \tag{3.5}$$

We choose w_i and x_i so as to minimize integration error. In *Gaussian quadrature*, we choose x_i to be the N roots of Ψ_N, the N^{th} order orthogonal polynomial with respect to X. We calculate the corresponding weights using the Lagrange basis polynomials L_i (Equation 3.4) passing through x_j $\forall j \neq i$.

Multivariate GPC. GPC can be easily extended to the multivariate case, as long as all *all random variables are independent.* Let $\mathbf{X} = (X_1, \ldots, X_d)$ be the d independent random variables (not necessarily following the same distribution) and let f be a function over $\mathbf{X}$. The orthogonal polynomials $\mathbf{\Psi_i}$ for $\mathbf{X}$ are simply the products of the orthogonal polynomials $\Psi_{i_1}, \ldots, \Psi_{i_d}$ for $X_1, \ldots, X_d$ respectively. The GPC approximation closely resembles the one for the univariate case:

$$f_N = \sum_{\mathbf{i}} c_{\mathbf{i}} \mathbf{\Psi_i} \quad \textit{where} \quad c_{\mathbf{i}} = \frac{\int_{S_{\mathbf{X}}} f(\boldsymbol{x}) \mathbf{\Psi_i}(\boldsymbol{x}) p_{\mathbf{X}}(\boldsymbol{x}) d\boldsymbol{x}}{\int_{S_{\mathbf{X}}} \mathbf{\Psi_i}^2(\boldsymbol{x}) p_{\mathbf{X}}(\boldsymbol{x}) d\boldsymbol{x}} \tag{3.6}$$

We can calculate $c_{\mathbf{i}}$ using a variation of Equation 3.5 in which we calculate the sum over all dimensions of $\mathbf{i}$.

Global sensitivity (Sobol) indices. Sobol indices [143] decompose the variance of the model output over the entire input distribution into portions that depend on subsets of the input variables X_i. The *first order* sensitivity indices show the contribution of a single input variable to the output variance. For a variable X_i, the sensitivity index is $S_i = V_i/V$. Here,

$V = Var_{\mathbf{X}}(f(\boldsymbol{x}))$ is the total variance and

$$\begin{aligned} V_i &= Var_{X_i}(E_{\mathbf{X}_{\neg i}}(f(\boldsymbol{x})|X_i = x_i)) \\ \textit{where} \quad \mathbf{X}_{\neg i} &= \{X_1, \ldots, X_d\} \setminus \{X_i\} \end{aligned} \tag{3.7}$$

We can evaluate Equation 3.7 analytically when f is a polynomial (such as those generated via GPC) and when it is possible to calculate the moments of each independent component of $\mathbf{X}$ analytically. For more complex functions and distributions, it becomes necessary to estimate Equation 3.7 empirically using Monte Carlo estimators [143, Equation 6].

3.3 GAS APPROACH

We present the GAS approach for creating a surrogate model of complex autonomous vehicle systems. GAS consists of three high level steps:

1. Create a deterministic complete vehicle model.
2. Train a perception model and use it to replace the regression neural network used for state perception in the vehicle model (Algorithms 3.1-3.2).
3. Construct a surrogate model for the complete vehicle model (Algorithm 3.3).

GAS automates almost the entire process of constructing and using the surrogate model. The user provides the environment distribution ($\mathcal{D}_E$), the distribution of other relevant random variables ($\mathcal{D}_R$), the initial state distribution ($\mathcal{D}_S^0$), and other simulation parameters as inputs to GAS.

3.3.1 Creating a Deterministic Vehicle Model

First, we represent the complete vehicle model as a deterministic function of independent random variables, $M_V : \mathbb{D}_S \times \mathbb{D}_E \times \mathbb{D}_R \to \mathbb{D}_S$. $S \in \mathbb{D}_S$ is a vector of state variables (e.g., position and rotation), $E \in \mathbb{D}_E$ is a vector of environment-related random variables that affect the neural network (e.g., weather and lighting conditions that affect the image processed by the neural network), and $R \in \mathbb{D}_R$ is a vector of random variables that do not affect the neural network, but affect other parts of M_V. Making M_V deterministic allows us to explicitly sample the output distribution of M_V for any given state. We remove any input variable dependencies by isolating independent components of input variables as necessary.

3.3.2 Replacing the Perception System

The output of regression neural networks which use camera images to perceive the vehicle's state is affected by environmental factors unrelated to the vehicle's state. For a given ground truth state, such perception systems produce a distribution of possible perceived states. To enable faster and systematic sampling of this output distribution, GAS replaces the perception system with a perception model prior to constructing the vehicle surrogate model.

Algorithm 3.1 Training the perception model

```
    Input $G$: set of ground truth states; $\mathcal{D}_E$: distribution of environment variables; $N_i$: number of images to capture for each $g \in G$
    Returns $M_{per}$: trained perception model; $d_{per}$: degree of perception model
 1: function TrainPerceptionModel($G, \mathcal{D}_E, N_i$)
 2:     $TrainTestData \leftarrow \{\ \}$
 3:     for $g \in G$ do
 4:         $I_g \leftarrow [\ ]$
 5:         for $i$ from 1 to $N_i$ do
 6:             $E \sim \mathcal{D}_E$
 7:             $Img \leftarrow$ CaptureImage($g, E$)
 8:             $I_g \leftarrow I_g :: Img$
 9:         $O_g \leftarrow$ NeuralNetwork($I_g$)
10:         $\mu_g \leftarrow$ Mean($O_g$)
11:         $\sigma_g^2 \leftarrow$ Covariance($O_g$)
12:         $TrainTestData \leftarrow TrainTestData[g \mapsto (\mu_g, \sigma_g^2)]$
13:     $M_{per}, d_{per} \leftarrow$ TrainPolynomialRegressionModel($TrainTestData$)
```

Algorithm 3.1 shows how GAS creates the perception model M_{per}. GAS first chooses a set of ground truth states G from the set of safe states $\mathbb{D}_S^{safe}$. For each ground truth state $g \in G$, GAS 1) captures a list of images I_g in environments E sampled from the environment distribution $\mathcal{D}_E$, 2) passes I_g through the perception neural network to obtain a list of outputs O_g, and 3) calculates the mean μ_g and covariance σ_g^2 of O_g. GAS trains a polynomial regression model M_{per} to predict the mean μ_S and covariance σ_S^2 of the output distribution at any ground truth state $S \in \mathbb{D}_S^{safe}$. GAS also infers the optimal polynomial degree d_{per} to maximize accuracy while preventing overfitting.

We create an *abstracted* vehicle model $M'_V : \mathbb{D}_S \times \mathbb{R}^n \times \mathbb{D}_R \to \mathbb{D}_S$ (Algorithm 3.2) which uses M_{per} instead of the neural network. For the input vehicle state S, M'_V first calculates the perception neural network output distribution. Specifically, GAS assumes that this output distribution is $\mathcal{N}(\mu_S, \sigma_S)$, where μ_S and σ_S^2 are the output distribution parameters at S predicted using M_{per}. Instead of a sample from $\mathcal{D}_E$, M'_V accepts a sample N from a multivariate standard normal distribution $\mathcal{N}(0, 1)$. M'_V transforms N to a sample O_S from

Algorithm 3.2 Abstracted vehicle model

```
    Input $S$: initial state of vehicle; $N$: raw sample to be transformed into a neural network output sample; $R$: other relevant random variables; $M_{per}$: trained perception model
    Returns $S'$: state of vehicle after one time step
1: function $M'_V(S, N, R, M_{per})$
2:     $\mu_S, \sigma_S^2 \leftarrow M_{per}(S)$
3:     $O_S \leftarrow \textsc{SampleTransform}(N, \mu_S, \sigma_S^2)$
4:     $S' \leftarrow \textsc{VehicleControlAndDynamics}(S, O_S, R)$
```

$\mathcal{N}(\mu_S, \sigma_S)$. M'_V uses this sample as the perceived state for the rest of the model consisting of the vehicle's control and dynamics systems.

GAS's assumption that the perception neural network output is distributed according to $\mathcal{N}(\mu_S, \sigma_S)$ is based on the the output distribution for real-world images, which has a normal distribution as shown in Figure 3.3. However, GAS is capable of using the same method for other distributions if the parameters of the fitted distribution vary smoothly as the ground truth changes.

3.3.3 GPC for the Complete Vehicle System

Algorithm 3.3 GPC surrogate model construction

```
    Input $\mathcal{D}_S^{safe}$: distribution over $\mathbb{D}_S^{safe}$; $\mathcal{D}_R$: distribution of other relevant random variables; $o_{gpc}$: order of GPC model; $M'_V$: abstracted vehicle model
    Returns $M_{GPC}$: GAS surrogate model
1: function $\textsc{CreateGPCModel}(\mathcal{D}_S^{safe}, \mathcal{D}_R, o_{gpc}, M'_V)$
2:     $J \leftarrow \textsc{Join}(\mathcal{D}_S^{safe}, \mathcal{N}(0,1), \mathcal{D}_R)$
3:     $\Psi \leftarrow \textsc{GenerateOrthogonalPolynomials}(o_{gpc}, J)$
4:     $X, W \leftarrow \textsc{GenerateQuadratureNodesAndWeights}(o_{gpc}, J)$
5:     $Y \leftarrow [M'_V(x)$ for $x \in X]$
6:     $M_{GPC} \leftarrow \textsc{QuadratureAndGPC}(\Psi, X, W, Y)$
```

Algorithm 3.3 shows how GAS constructs the GPC approximation of the abstracted vehicle model M'_V. GAS first constructs a joint distribution J over all input variables to M'_V by taking the product of a distribution for each input variable. GPC will produce a polynomial approximation that minimizes ℓ_2 error weighted by the probability distribution of J. For the state variables, GAS chooses a normal or truncated normal distribution $\mathcal{D}_S^{safe}$ over the safe state space $\mathbb{D}_S^{safe}$. For the raw sample that is transformed into a sample from the perception system output distribution O_S, GAS uses $\mathcal{N}(0,1)$, as that ensures that the transformed sample is indeed distributed according to O_S. For the other relevant random variables, GAS uses their actual distribution $\mathcal{D}_R$. Then, $J = \mathcal{D}_S^{safe} \times \mathcal{N}(0, 1) \times \mathcal{D}_R$. GAS calculates the basis polynomials Ψ which are orthogonal with respect to J. To efficiently calculate the coefficients of the polynomials in Ψ, GAS uses Gaussian quadrature for numerical integration. The surrogate model M_{GPC} is the sum of the orthogonal basis polynomials multiplied by these calculated coefficients, as per Equation 3.6.

Categorical state variables. Some vehicle models have categorical state variables. For example, many control systems operate in distinct modes. The control system can switch modes if certain conditions are met, and the current mode affects the control decisions. In this case, the current mode is a categorical state variable. Unlike categorical variables, polynomial inputs and outputs are continuous intervals. Therefore, we cannot use GPC for predicting categorical variables, or accept a categorical variable as an input to the GPC model. One option is to create a different type of surrogate model for such vehicle models. However, GAS can still enable the use of GPC by using multiple GPC sub-models and using a separate classifier for predicting output categorical variables. This procedure is known as *multi-element GPC* (ME-GPC).

Consider a vehicle model M_V whose state includes a categorical variable X with the domain $\mathbb{D}_X = \{x_1, \ldots, x_k\}$. GAS uses GPC to create a separate polynomial model for each $x_i \in \mathbb{D}_X$. The compound surrogate model chooses which of these sub-models to use based on the current value of X. In this way, GAS calculates all output state variables except X. For predicting X, GAS creates an ancillary classifier. GAS trains the ancillary classifier in a similar manner as the perception model in Algorithm 3.1, with the main distinction being that it creates a classification model as opposed to a regression model.

Alternate surrogate models. GAS's perception model also enables the creation of alternative complete vehicle system surrogate models. For example, we can use standard polynomial regression instead of GPC, or use a neural network as a surrogate. GAS samples the output of M'_V for a large sample of inputs, and uses standard regression model training techniques to train such alternative surrogate models.

3.3.4 Applications of the GAS Surrogate Model

Developers can use M_{GPC} instead of M_V to infer various properties about the vehicle system, as long as those properties can be calculated by observing the evolution of the vehicle's state distribution over time. This chapter explores two such properties:

Algorithm 3.4 Estimating the probability of remaining in a safe state over time

```
    **Input** $N_s$: number of samples to use for distribution estimation; $T$: number of time steps; $\mathcal{D}_S^0$: initial state distribution; $\mathcal{D}_R$: distribution of other relevant random variables; $Pred$: safety predicate; $M_{GPC}$: GPC surrogate model
    **Returns** $P_{safe}$: probability that the vehicle is safe until each time step
 1: function EstimateSafeProbability($N_s, T, \mathcal{D}_S^0, \mathcal{D}_R, Pred, M_{GPC}$)
 2:     $X \leftarrow [\,]$; $P_{safe} \leftarrow \{\,\}$
 3:     for $i$ from 1 to $N_s$ do
 4:         $x \sim$ Join($\mathcal{D}_S^0, \mathcal{N}(0,1), \mathcal{D}_R$)
 5:         $X \leftarrow X :: x$
 6:     for $t$ from 1 to $T$ do
 7:         $X' \leftarrow [\,]$
 8:         for $x \in X$ do
 9:             $S' \leftarrow M_{GPC}(x)$
10:             if Safe($S', Pred$) then
11:                 $N' \sim \mathcal{N}(0,1)$
12:                 $R' \sim \mathcal{D}_R$
13:                 $X' \leftarrow X' :: (S', N', R')$
14:         $X \leftarrow X'$
15:         $P_{safe} \leftarrow P_{safe}[t \mapsto |X|/N_s]$
```

Calculating probability of remaining in a safe state over time. GAS uses the surrogate model to estimate the probability that the vehicle will remain in a safe state over time (Algorithm 3.4). GAS creates initial state samples S using the initial state distribution $\mathcal{D}_S^0$. At each time step, GAS chooses random samples N and R from $\mathcal{N}(0,1)$ and $\mathcal{D}_R$ respectively. GAS evaluates the surrogate model on each joint sample to get the next state. Finally, GAS calculates and logs the fraction of samples that did not violate the safety predicate $Pred$.

Algorithm 3.5 Using estimators to calculate sensitivity indices

```
    **Input** $N_s$: number of samples to use for sensitivity estimation; $i$: index of state variable to calculate sensitivity for; $\mathcal{D}_S$: current state distribution; $\mathcal{D}_R$: distribution of other relevant random variables; $M$: model ($M_{GPC}$ or $M'_V$)
    **Returns** $S_i$: sensitivity index of selected state variable
 1: function EstimateSensitivity($N_s, i, \mathcal{D}_S, \mathcal{D}_R, M$)
 2:     $Y_0 \leftarrow [\,]$; $Y_1 \leftarrow [\,]$
 3:     for $i$ from 1 to $N_s$ do
 4:         $x_0, x_1 \sim$ Join($\mathcal{D}_S, \mathcal{N}(0,1), \mathcal{D}_R$)
 5:         $y_0 \leftarrow M(x_0)$; $y_1 \leftarrow M(x_1[i \mapsto x_0[i]])$
 6:         $Y_0 \leftarrow Y_0 :: y_0$; $Y_1 \leftarrow Y_1 :: y_1$
 7:     $S_i \leftarrow$ (Mean($Y_0 * Y_1$) $-$ Mean($Y_0$)$^2$)/Variance($Y_0$)
```

Computing Sobol indices. GAS uses M_{GPC} to calculate Sobol sensitivity indices in two ways. In the *analytical* approach, GAS calculates sensitivity indices by first calculating conditional expected values as polynomials and then calculating their variance (Equation 3.7). In the *empirical* approach, GAS instead uses Monte Carlo estimators ([143, Equation 6] as implemented in Algorithm 3.5).

While the analytical approach precisely calculates sensitivity indices, it is relatively slow, as GAS must calculate expected values as a function of the variable whose sensitivity is being calculated. The empirical approach becomes more accurate as the number of samples increases. Due to the speed of evaluating M_{GPC}, the empirical approach can still be faster than the analytical approach while providing acceptable accuracy.

Rapid iteration. During development of an autonomous vehicle system, the vehicle model can change rapidly as the perception neural network or vehicle control or dynamics parameters are tweaked. GAS enables faster testing of these prototypes thanks to its compositional approach. If the same perception system is used while changing the control and dynamics, then GAS saves time by reusing the existing perception model. If the perception system is changed, GAS must rerun Algorithms 3.1-3.3. If doing so is faster than using MCS, the time saved still adds up with each iterative change. Even after initial development is complete, GAS continues to provide speedups as new features are added in production.

3.3.5 Properties of the GAS Approach

Accuracy. Multiple GAS parameters affect the accuracy of the GPC model: the size of the tensor grid $|G|$, the number of images taken for each grid point N_i, the degree of polynomial regression used for the perception model, and the GPC order o_{GPC}. Under certain conditions, GAS converges in distribution to the exact solutions.

Lemma 3.1 (Perception model convergence)**.** Assume that 1) for the given environment distribution $\mathcal{D}_E$, the distribution of the outputs of a perception neural network $\mathfrak{N}$ in any ground truth state S is Gaussian over the perceived state, and 2) each component of the distribution parameters (μ_S, σ_S^2) is an analytic function of S. Then, the output distribution of the perception model M_{per} in any state approaches $\mathfrak{N}$'s output distribution at that state as $|G|$, N_i, and d_{per} increase.

Proof Sketch. Increasing $|G|$ increases the number of ground truth states used to train the perception model. Increasing N_i increases the accuracy of $\mathfrak{N}$'s output distribution parameters calculated at each S. Since these distribution parameters are analytic functions of S, they can be calculated using a Taylor series over S. After increasing the number and accuracy of training data points, the accuracy of the perception model can be arbitrarily increased by increasing d_{per}[3]. QED.

We use standard statistical tests such as the Shapiro-Wilk test to check if $\mathfrak{N}$'s outputs have a Gaussian distribution for the environment distribution $\mathcal{D}_E$ used in Section 3.3.2. We have observed this to be true in practice (Figure 3.3). We can also use a different base distribution (and corresponding orthogonal polynomials for GPC) if it fits the data better across the state space.

Practically, controlling the error of the perception model (or any approximation of a neural network) is an open problem [146, 147]. Precise analytic calculation of the perception model error is intractable, but we can empirically estimate the error.

Lemma 3.2 (GPC error bound)**.** Assume the control system and vehicle dynamics in M'_V are differentiable. Then, the ℓ_2 error of the output of the GAS model M_{GPC} with respect to the output of M'_V is bounded.

Proof Sketch. From [141, Theorem 3.6] and Ernst et al. [148], which state that the ℓ_2 error of a GPC approximation is proportional to o_{GPC}^{-p}, where p is a positive value that depends on the differentiability of the function being approximated. The process of generating a neural network output sample through the perception model is a polynomial evaluation followed by an affine transform; both are differentiable operations. The control system and dynamics are differentiable by assumption. Finally, composing differentiable functions yields a differentiable function. QED.

M_{GPC} is the optimal polynomial model of M'_V for any o_{GPC} ([141, Equation 5.9]), in terms of ℓ_2 error. In practice, control systems may not be differentiable everywhere (e.g., due to mode switching), but the differentiability of vehicle dynamics, coupled with a short duration time step, limits negative effects on accuracy.

Corollary 3.1 (GPC convergence)**.** As $o_{GPC} \to \infty$, the ℓ_2 error of GPC approaches 0, that is, M_{GPC} can be an arbitrarily close approximation of M'_V.

Proof Sketch. From Lemma 3.2, the ℓ_2 error is proportional to o_{GPC}^{-p}, where p is positive. Then, $\lim\limits_{o_{GPC}\to\infty} o_{GPC}^{-p} = 0$. QED.

Theorem 3.1 (GAS convergence)**.** Assume that the distribution of the outputs of a perception neural network $\mathfrak{N}$ for any ground truth state in $\mathbb{D}_S^{safe}$ is Gaussian. Then, the GAS model M_{GPC} converges in output distribution to the original vehicle model M_V.

[3]Increasing d_{per} without also increasing $|G|$ leads to overfitting.

Table 3.1: GAS benchmarks

Benchmark	**Perception**	**Control**	**Replacement**	$\mathbf{dim_{s/r}}$
Crop-Monitor	ResNet-18×2	Skid-steer	Perc → Polynomial regression	2/2
Car-Straight	LaneNet	Pure pursuit	Perc → Polynomial regression	2/2
Car-Curved	LaneNet	Pure pursuit	Perc → Polynomial regression	2/2
ACAS-Table	Ground truth	ACAS-Xu table	Ctrl → Ancillary decision tree	4/0
ACAS-NN	Ground truth	ACAS-Xu NN	Ctrl → Ancillary decision tree	4/0

Proof Sketch. Since we can construct an arbitrarily accurate perception model (Lemma 3.1), we can use it to obtain accurate neural network output samples for any state in $\mathbb{D}_S^{safe}$. The GPC model can be made an arbitrarily accurate approximation of M'_V (Corollary 3.1), and thus of M_V. QED.

Runtime. The dominant factor for runtime is the required number of evaluations of M_V. To gather data for the perception model, GAS requires $\Theta(|G|N_i)$ evaluations of M_V. The amount of time required to train M_{per} and construct M_{GPC} is insignificant in comparison.

For state distribution estimation over time, MCS requires $\Theta(N_sT)$ evaluations of M_V (N_s being the number of samples used for distribution estimation), while GAS requires the same number of evaluations of the much faster M_{GPC}. For estimating sensitivity indices using estimators, we must evaluate either M'_V or M_{GPC} $\Theta(N_s)$ times, followed by mean and variance calculations.

3.4 METHODOLOGY

Benchmarks. We chose five benchmarks that include autonomous vehicle systems such as self driving cars, unmanned aircraft, and crop monitoring vehicles. Table 3.1 shows details of the benchmarks. Columns 2 and 3 state the vehicle's perception and control system, respectively. Column 4 indicates if GAS made a replacement in the perception (Perc) or control (Ctrl) system, and the nature of the replacement, prior to creating the complete vehicle system surrogate. Column 5 shows the number of state and random variable dimensions. We describe the benchmarks further below:

- *Crop-Monitor:* A crop monitoring vehicle that travels between two rows of crops and must avoid hitting them. This is our main example (Section 3.1).

- *Car-Straight:* A self-driving vehicle that must drive within a road lane ($\mathbb{D}_S^{safe} \equiv |heading| \leq \pi/12 \wedge |distance| \leq 1.2m$). It uses LaneNet [30, 149] to perceive the lane boundaries and uses the pure pursuit controller. We derive this benchmark from [150].

- *Car-Curved:* Similar to the previous benchmark, but the vehicle must drive on a circular road of radius 100m.
- *ACAS-Table:* An unmanned aircraft that must avoid a near miss or collision with an intruder ($\mathbb{D}_S^{safe} \equiv |separation| \geq 0.1524km$). The aircraft uses the ACAS-Xu lookup tables from [151]. As this model's state includes a categorical variable (the previous ACAS advisory), we use ME-GPC and predict the next advisory using a decision tree as the ancillary model.
- *ACAS-NN:* Similar to the previous benchmark, but uses a neural network from [151] trained to replace the lookup table.

Implementation and experimental setup. We performed our experiments on machines with a Quadro P5000 GPU, using a single Xeon CPU core. We implement GAS in Python, using the `chaospy` library [152]. We use Gazebo 11 [145] to capture images for Crop-Monitor, Car-Straight, and Car-Curved benchmarks. We run all image processing neural networks on the GPU. We run ACAS-NN entirely on CPU as its network is small.

For our main evaluation, we create only GPC surrogate models with GAS. We briefly discuss the results of using some alternative surrogate models in Section 3.5.5. For estimating state distribution over time, we compare the GAS-generated M_{GPC} to a MCS baseline using M_V. We set GAS parameters as follows: G is a 11×11 grid in the safe state space, $N_i = 350$, and $o_{GPC} = 4$. We additionally experiment with alternate values for G and N_i, as they directly affect perception model training data generation time. We set the number of time steps $T = 100$. To keep MCS runtime within 24 hours, We set $N_s = 1,000$ for MCS. For GAS, we increase N_s to 10,000 as M_{GPC} is much faster than M_V and increasing the number of samples decreases sampling error for M_{GPC}.

For calculating sensitivity indices, we calculate sensitivity using both the analytical and empirical method described in Section 3.3. It is not possible to compare sensitivity indices against M_V, as M_V has a different set of inputs (environment specification instead of a sample from $\mathcal{N}(0,1)$). Therefore, we use sensitivity index calculation using M'_V as the baseline. For the empirical method, we set $N_s = 10^6$, but also monitor the results obtained by setting N_s to 10^4, 10^5, and 10^7. We calculate the sensitivity of state variables to those in the previous time step, as well as the sensitivity of the change in the state variables.

Environmental factors. For the Crop-Monitor benchmark, our test scenario includes two types of crops (corn and tobacco). There are four different corn growth stages. For each type and growth stage, there are multiple crop models (30 in total). For the Car benchmarks, our scenario includes the presence of 0-2 other cars, 0-2 pedestrians, and skid marks that may obstruct lane markings. We also vary lighting conditions from midday to dusk lighting.

Distribution similarity metrics. We use two complementary similarity metrics to separately compare each dimension of the MCS and GAS state distributions at each time step. The conservative *Kolmogorov-Smirnov (KS) statistic* quantifies the maximum distance between the cumulative distribution functions of the two distributions at any point. The *Wasserstein metric* quantifies the minimum probability mass that must be moved to transform one distribution into the other. For both metrics, a lower value indicates greater distribution similarity. We can use these distribution similarity metrics despite using more samples for GAS than for MCS.

Safe state probability similarity metrics. We also compare the fraction of simulated vehicles remaining in the safe region till each time step using three metrics. The *two sample t-test* is a statistical test to check if the underlying distributions used to draw two sets of samples are the same. The *RMS error* is the root-mean-square of the differences in safe state probability at each time step. Lastly, we calculate the *Pearson cross-correlation coefficient* between the two sets of safe state probabilities. When plotting safe state probability, we also draw the 95% bootstrap confidence interval. This confidence interval does not directly compare the two plots, but rather, for each individual plot, it provides an estimate of the variation that can occur in that plot as a result of sampling error.

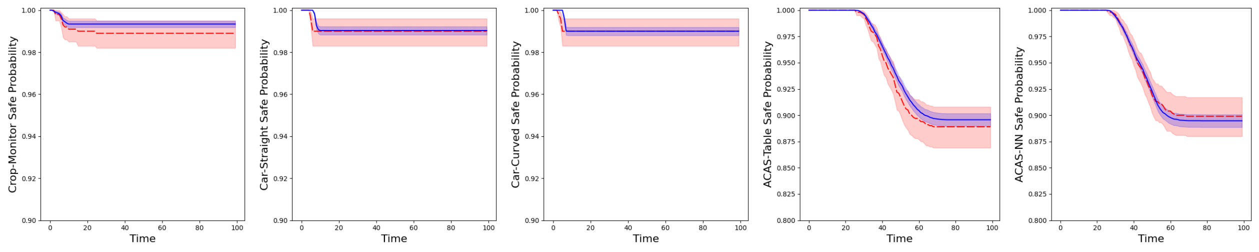
3.5 EVALUATION

3.5.1 Accuracy of GAS for Estimating the Probability That the Vehicle Will Remain in a Safe State Over Time

Table 3.2 compares the distributions calculated by GAS and MCS for each benchmark state variable. Columns 3-4 compare the mean and standard deviation of the distributions at the *final* time step. Columns 5-6 show the maximum values of the KS statistic and Wasserstein metric over *all* time steps. For most state variables, the mean and standard deviation of the distributions match closely up to the final time step. This is also indicated by the low values of the Wasserstein metric and the conservative KS statistic. The largest difference is for the Car-Straight benchmark distance distribution. This occurs because M_{GPC} and M_V converge towards slightly different states around the center of the safe state space in later time steps. However, during the initial time steps where more simulated

Table 3.2: Metrics for comparing state variable distributions

Benchmark	**Variable**	$\mu_{\mathbf{GAS}}/\mu_{\mathbf{MCS}}$	$\sigma_{\mathbf{GAS}}/\sigma_{\mathbf{MCS}}$	$\mathbf{KS_{max}}$	$\mathbf{Wass_{max}}$
Crop-Monitor	Heading (rad)	-0.004/-0.008	0.04/0.04	0.11	0.02
	Distance (m)	-0.01/-0.02	0.03/0.03	0.14	0.01
Car-Straight	Heading (rad)	0.0004/-0.0001	0.02/0.02	0.13	0.009
	Distance (m)	0.16/0.08	0.09/0.12	0.41	0.08
Car-Curved	Heading (rad)	-0.003/-0.005	0.006/0.007	0.17	0.003
	Distance (m)	0.18/0.19	0.04/0.05	0.15	0.02
ACAS-Table	Crossrange (km)	-0.07/-0.03	0.85/0.88	0.05	0.04
	Downrange (km)	-0.61/-0.53	0.23/0.22	0.13	0.08
	Heading (rad)	0.63/-0.54	2.82/2.80	0.23	1.23
ACAS-NN	Crossrange (km)	-0.01/-0.01	0.93/0.91	0.02	0.03
	Downrange (km)	-0.45/-0.58	0.17/0.11	0.31	0.13
	Heading (rad)	0.42/0.15	2.70/2.75	0.06	0.27



(a) Crop-Monitor (b) Car-Straight (c) Car-Curved (d) ACAS-Table (e) ACAS-NN

Figure 3.6: Evolution of the probability of remaining in a safe state over time. Blue solid: GAS, red dashed: MCS.

vehicles are close to entering unsafe states, the KS statistic does not exceed 0.15. A similar phenomenon affects the ACAS-NN downrange distance variable. For ACAS-Table, M_{GPC} and M_V occasionally turn in different directions to avoid an intruder approaching head-on, in situations where turning in either direction is equally beneficial. This leads to a large deviation in the heading variable.

Figure 3.6 shows how the probability that the vehicle remains in a safe state evolves over time. The blue solid and red dashed plots show the probability estimates obtained using GAS and MCS, respectively. The shaded region around each plot shows the 95% bootstrap confidence interval. Because we use 10× more samples when estimating safe state probability with GAS as compared to MCS, the sampling error is smaller for GAS, which leads to a smaller confidence interval. Since M_{GPC} approximates the behavior of M_V, as opposed to under/over approximation of reachable states, GAS's safe state probability estimate can be either greater or lesser than the MCS estimate.

Table 3.3 shows the metrics we use to measure the similarity of the safe state probabilities

Table 3.3: Metrics for comparing the probability of remaining in a safe state

Benchmark	**t-test**	**RMS err.**	**X-Cor**
Crop-Monitor	99/100	0.004	0.974
Car-Straight	97/100	0.001	0.862
Car-Curved	98/100	0.001	0.865
ACAS-Table	100/100	0.007	0.998
ACAS-NN	100/100	0.003	0.999

Table 3.4: Maximum difference in sensitivity indices

Benchmark	$\mathbf{x_0 \rightarrow y_1}$	$\mathbf{x_0 \rightarrow dy_0}$
Crop-Monitor	0.00003	0.0004
Car-Straight	0.0002	0.006
Car-Curved	0.00003	0.003
ACAS-Table	0.00001	0.009
ACAS-NN	0.00002	0.061

from Figure 3.6. Column 2 shows the number of time steps for which the t-test passed, meaning that we could not reject the null hypothesis that the probabilities are equal. Column 3 shows the RMS error, and Column 4 shows the cross-correlation. The similarity of the state distributions leads directly to the similarity of the safe state probability for most time steps.

We extended the Crop-Monitor and Car benchmark experiments to 500 time steps to confirm that the safe state probability does not deviate after 100 time steps. We did not similarly extend the ACAS experiments as the ACAS system is primarily relevant as the aircraft approach each other, which is no longer the case after the first 100 time steps.

3.5.2 Accuracy of GAS for Estimating Global Sensitivity Indices of the Vehicle Model

Table 3.4 presents the maximum difference between sensitivity indices calculated using M_{GPC} and those calculated using M'_V. Column 2 shows the sensitivity of the state variables in time step 1 to those in time step 0 ($x_0 \rightarrow y_1$). Column 3 shows the sensitivity of the *change* in the state variables ($x_0 \rightarrow dy_0$, where $dy_0 = y_1 - y_0$). The sensitivity indices calculated by M_{GPC} and M'_V match closely.

Figure 3.7 shows an example visual comparison of sensitivity indices calculated by GAS and MCS for Crop-Monitor. The input variables include the heading and distance at time step 0, and the two components of the raw sample (Raw0 and Raw1) that is transformed into the perception neural network output distribution sample. The output variables are the heading and distance at time step 1. There is one sensitivity index corresponding to each input/output variable pair. The blue solid line shows the sensitivity calculated analytically using M_{GPC}, the blue dotted line shows the sensitivity calculated empirically using M_{GPC}, and the red dashed line shows the sensitivity calculated empirically using M'_V. In each subplot, the X-Axis shows the number of samples used for empirical estimation, while the Y-Axis shows the calculated sensitivity index. As the number of estimation samples is increased, the empirically calculated sensitivity indices converge towards those calculated

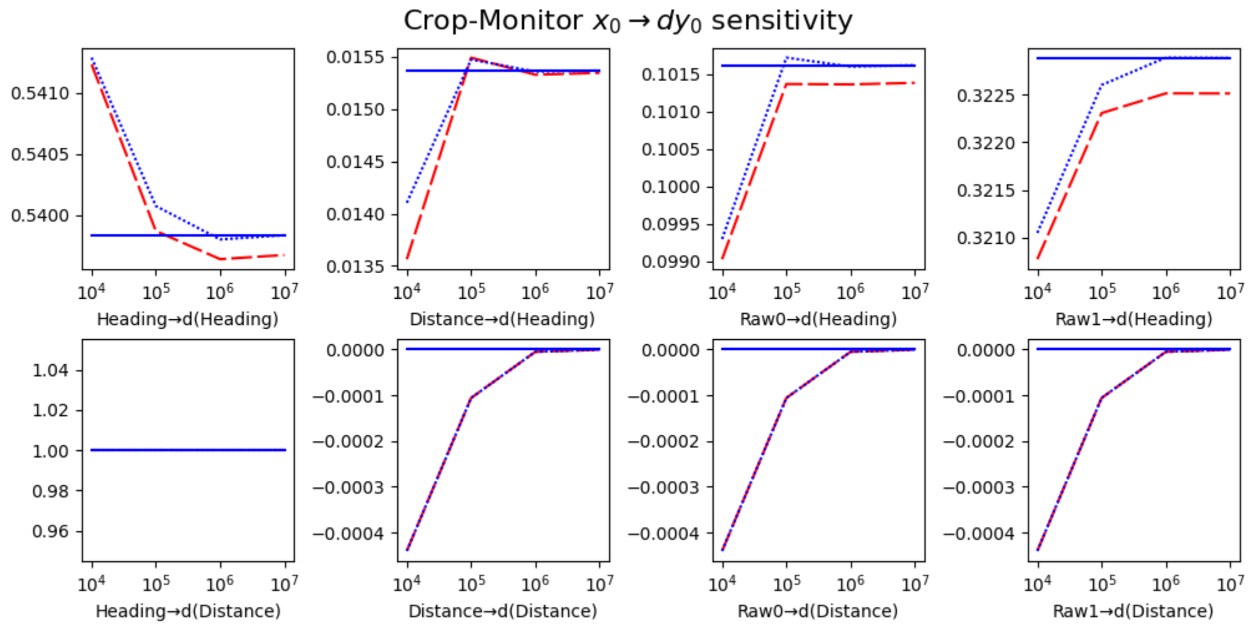


Figure 3.7: Calculated sensitivity index comparison. Blue solid: GAS (analytical method), blue dotted: GAS (empirical method), red dashed: MCS.

analytically. About 10^6 samples are needed for convergence.

3.5.3 Effect of GAS Perception Model Parameters on GAS Accuracy

Table 3.5 shows the effects of changing the perception model parameters G and N_i. Column 1 presents the value of the parameter. Columns 2-4 present the maximum KS statistic and Wasserstein metric ($\times 100$) for the Crop-Monitor, Car-Straight, and Car-Curved benchmarks, respectively. Column 5 presents the estimated speedup caused by changing the parameter value, calculated based on the reduction in the number of images that must be captured and processed. We exclude the ACAS benchmarks from this ablation study as they do not use a perception model.

We focus on the cases where the error metrics change by 10% or more. While the grid size G can be reduced to 9×9 without much loss of accuracy, further reducing it to 7×7 increases the error for all benchmarks. Similarly, reducing the number of images captured at each grid point (N_i) to 225 does not cause much loss of accuracy, but further reducing it to 100 images increases the error for all benchmarks. We prefer to err on the side of caution by collecting 350 images over a 11×11 grid. The minimal change in accuracy caused by increasing G from 9×9 to 11×11 or increasing N_i from 225 to 350 also shows that further increases are unlikely to improve the accuracy of GAS.

Table 3.5: Effect of changing perception model parameters on the KS statistic. An asterisk (*) indicates the primary value used in our evaluation. Changes of 10% or more are annotated with an $^{\uparrow}$ for increases and an $^{\downarrow}$ for decreases.

	$\mathbf{KS_{max}} \times \mathbf{100} / \mathbf{Wass_{max}} \times \mathbf{100}$			
Parameter Value	**Crop-Mon**	**Car-Str**	**Car-Cur**	**Relative Speedup**
Ground truth grid dimensions (G)				
7×7	**20.3**$^{\uparrow}$ / 2.43	**46.8**$^{\uparrow}$ / **9.63**$^{\uparrow}$	**23.8**$^{\uparrow}$ / **3.24**$^{\uparrow}$	2.5×
9×9	**12.2**$^{\downarrow}$ / 2.25	42.4 / 8.39	16.2 / **2.00**$^{\downarrow}$	1.5×
11×11*	13.8 / 2.37	41.1 / 7.97	17.0 / 2.23	1.0×
Images captured per grid point (N_i)				
100	**33.1**$^{\uparrow}$ / 2.34	**49.6**$^{\uparrow}$ / **9.77**$^{\uparrow}$	**18.7**$^{\uparrow}$ / **2.59**$^{\uparrow}$	3.5×
225	**16.1**$^{\uparrow}$ / 2.35	40.5 / 7.91	17.0 / 2.43	1.6×
350*	13.8 / 2.37	41.1 / 7.97	17.0 / 2.23	1.0×

Table 3.6: Time required for the construction of M_{GPC}

Benchmark	$\mathbf{t_{dat}}$	$\mathbf{t_{per/anc}}$	$\mathbf{t_{GPC}}$
Crop-Monitor	8.5h	1.1s	1.4s
Car-Straight	3.3h	1.1s	1.4s
Car-Curved	3.1h	1.1s	1.4s
ACAS-Table	N/A	0.3s	0.3s
ACAS-NN	N/A	0.3s	0.4s

Table 3.7: Time required for estimating the state distribution over time

Benchmark	$\mathbf{t_{MCS}}$	$\mathbf{t_{GAS}}$
Crop-Monitor	19.5h	8.5h (2.3×)
Car-Straight	6.8h	3.3h (2.1×)
Car-Curved	6.5h	3.1h (2.1×)
ACAS-Table	8.0s	1.1s (7.3×)
ACAS-NN	10.7s	1.2s (8.9×)

3.5.4 Speedup of GAS Compared to Monte Carlo Simulation

GAS model construction. Table 3.6 shows the time required to construct the GAS model. Column 2 shows the time required to gather training data for the perception model when necessary. Column 3 shows the time required to create the perception and/or ancillary model when necessary. Column 4 shows the time required to create M_{GPC}. While creating the perception, ancillary, and GPC models takes a few seconds, gathering the training data for the perception model takes several hours. As shown in Section 3.5.3, reducing perception model parameters such as $|G|$ and N_i can reduce this time, but can also lead to a reduction in accuracy.

State distribution estimation. Table 3.7 shows the time required by GAS and MCS for state distribution estimation. Column 2 shows the time required for using M_V for state distribution estimation. Column 3 shows the total time required by GAS for state distribution estimation; this includes the *total* time required to construct M_{GPC} (from Table 3.6) and then use it for state distribution estimation via Algorithm 3.4. Column 3 also shows the speedup of GAS over MCS. The costly process of gathering and processing images con-

Table 3.8: Time usage for sensitivity analysis (excluding t_{dat} and t_{per} from Table 3.6)

Benchmark	$\mathbf{t_{MCS}^{emp}}$	$\mathbf{t_{GAS}^{emp}}$	$\mathbf{t_{GAS}^{ana}}$
Crop-Monitor	9.5s	6.2s (1.3×)	11.4s (0.7×)
Car-Straight	9.7s	6.2s (1.3×)	11.4s (0.8×)
Car-Curved	9.8s	6.2s (1.3×)	11.3s (0.8×)
ACAS-Table	5.2s	5.6s (0.9×)	3.3s (1.6×)
ACAS-NN	4.8s	5.3s (0.9×)	3.1s (1.5×)

tributes to over 99% of t_{MCS} for the Crop-Monitor and Car benchmarks. While increasing N_s or T increases t_{MCS} proportionally, the corresponding increase in t_{GAS} is negligible as the time required to create the perception model is independent of N_s and T. Consequently, the speedup of GAS for state distribution estimation increases for longer experiments or a higher number of samples. For the ACAS benchmarks, the control component of M_V contributes to over 90% of t_{MCS}. M_{GPC} is faster than even the dynamics component of M_V, leading to significant speedups for these benchmarks.

Sensitivity analysis. Table 3.8 shows the time required by GAS and MCS for sensitivity analysis. Columns 2-3 show the required for calculating *all* sensitivity indices empirically with M'_V and M_{GPC} respectively. Column 4 shows the time required calculating all sensitivity indices analytically with M_{GPC}. Columns 3-4 also show the speedup of GAS for the two approaches using M_{GPC}. Since both M'_V and M_{GPC} use the perception model for the Crop-Monitor and Car benchmarks, we exclude the time required to train and construct the perception model for those benchmarks. The replacement of the perception system by the perception model significantly speeds up M'_V as compared to M_V. As this optimized version is the baseline for sensitivity indices calculation, the speedup of GAS for this application is lower than that for distribution estimation. While the analytical and empirical methods for calculating sensitivity using M_{GPC} have similar accuracy, different methods are faster for different benchmarks.

Rapid iteration. For the Crop-Monitor and Car benchmarks, if the perception system is altered, GAS must create a new perception and GPC model. Consequently, while the speedup of GAS stays the same, the total amount of time saved over MCS increases with each such change. If only the vehicle control or dynamics components are altered, then GAS does not need to create a new perception model. Because gathering data for the perception model is the major contributor to the runtime of GAS as shown in Tables 3.6 and 3.7, this allows vehicle developers to rapidly make changes to the control and dynamics systems of the vehicle and re-analyze the system with these changes within seconds.

Table 3.9: t-test results comparison among surrogate model candidates

Benchmark	**GPC**	**Poly reg**	**Neural net**
Crop-Monitor	99/100	100/100	$(52 \pm 40)/100$
Car-Straight	97/100	97/100	$(49 \pm 39)/100$
Car-Curved	98/100	97/100	$(46 \pm 35)/100$
ACAS-Table	100/100	99/100	$(46 \pm 1)/100$
ACAS-NN	100/100	98/100	$(46 \pm 4)/100$

3.5.5 Comparison of GPC to Other GAS Surrogate Model Options

Although we focus on GPC surrogate models, GAS also enables the use of other surrogate models by replacing the complex perception system with a perception model. That is, we can train a different type of surrogate model instead of a GPC model. We can then use the alternative surrogate model for state distribution estimation or sensitivity analysis. Here, we briefly describe our results for two alternative surrogate models.

We experimented with using standard polynomial regression for creating the surrogate model. For training and testing data, we chose points in the safe state space using a Sobol sequence. Column 3 of Table 3.9 shows the t-test results for the regression surrogate model. For most benchmarks, the accuracy of the regression surrogate model is slightly lower than that of the GPC surrogate model (Column 2). This is in line with the fact that GPC produces the most optimal polynomial surrogate model for any order ([141, Equation 5.9]) in terms of ℓ_2 error. Further increasing the order of the regression surrogate model does not increase accuracy, but rather causes overfitting. The other safe state probability similarity metrics also show similar trends.

We also experimented with using a neural network surrogate model. Once again, we chose points in the safe state space using a Sobol sequence to generate training/testing data. We experimented with various neural network topologies and activations, with the number of parameters being at least the number used by the GPC model. We found that the accuracy of this surrogate model varied widely with the initial seed. Column 4 of Table 3.9 shows the t-test results for the neural network surrogate model. Due to the dependence on the initial seed, the results are presented in the form (mean $\pm$ standard deviation). The standard deviation is high for the Crop-Monitor and Car benchmarks, and the mean is low for all benchmarks. Without the original vehicle model for comparison, it would not be possible to know which random restart produced the most accurate model. In contrast, GPC deterministically produces an optimal polynomial model.

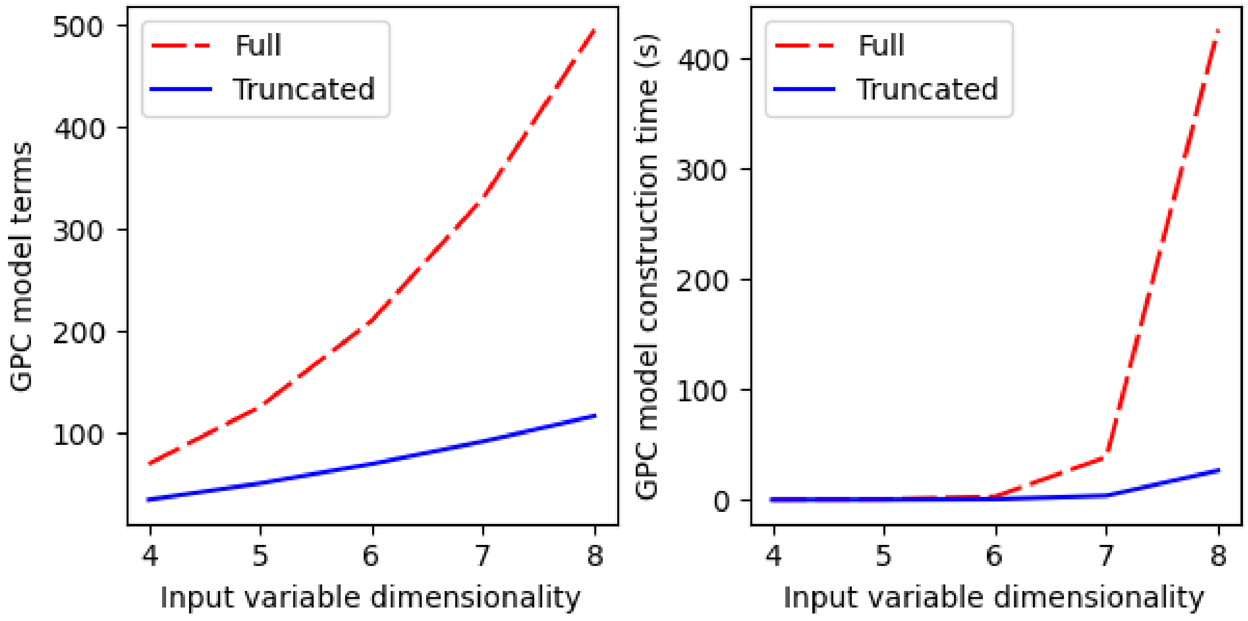


Figure 3.8: Effect of truncation on GPC model terms and construction time for an order 4 GPC model.

3.6 DISCUSSION

3.6.1 Scalability

The GPC polynomial model must consider all possible interactions between input variables. As a result, it can suffer from the *curse of dimensionality* (combinatorial explosion of the number of polynomial terms). This is a general limitation of GPC, for which researchers in engineering applications have proposed solutions such as low-rank approximation [153].

Another solution is to omit higher-order polynomial terms in which multiple state variables interact [154]. This is similar to the method of eliminating high-frequency components of images used by JPEG [34] for lossy image compression. Figure 3.8 shows the effects of this polynomial truncation approach for an order 4 GPC model. For both plots, the X-Axis shows the number of dimensions in the input state space. The Y-Axis of the left plot shows the number of polynomial terms in the constructed GPC model, and the Y-Axis of the right plot shows the model construction time. The dashed red lines and the solid blue lines show the results for the full polynomial and the truncated polynomial, respectively. The speedup from truncation increases rapidly with dimensionality. Critically, this method retains lower-order interaction terms, ensuring that the constructed GPC model can still account for interactions between state variables (albeit to a lesser degree of fidelity).

3.6.2 Limitations

Environment distribution. GAS samples the provided environment distribution $\mathcal{D}_E$ when generating training data for the perception model in Algorithm 3.1. If GAS uses the perception model created for $\mathcal{D}_E$ to create a GPC model for a scenario with a significantly different environment distribution $\mathcal{D}'_E$, it can lead to a reduction in accuracy of the final GPC model. To prevent this issue, we must carefully choose $\mathcal{D}_E$ to represent the distribution of environments the vehicle is expected to operate in. If it becomes necessary to create a new perception model for $\mathcal{D}'_E$, we may be able to partially reuse existing training data from $\mathcal{D}_E$ that also fits into $\mathcal{D}'_E$, reducing the additional time required to create the new perception model.

Supported distributions. To construct the GPC model in Algorithm 3.3, all component distributions of the multivariate distribution J must have corresponding orthogonal polynomials. This is true for a wide variety of distributions such as the uniform, normal, beta, and gamma distributions. We must ensure that all input variables for the GPC model are independent and are instances of these distributions. The wide variety of supported distributions is usually sufficient for modeling relevant input variables.

Polynomial models. As GPC produces a polynomial model, its accuracy is limited when modeling functions with discontinuities or limited differentiability [141, Theorem 3.6]. Practically, this means that the GPC model will have a systemic bias that can be reduced, but not eliminated. Our evaluation shows that GAS's hyperparameters must be carefully chosen to maximize accuracy while still providing speedups over MCS. Once the ideal hyperparameters are found for modeling a particular vehicle system, developers can reuse them when making iterative changes to the vehicle model.

3.6.3 Alternative Formulations

The GPC model construction process in Algorithm 3.3 evaluates the abstracted vehicle model at specific quadrature nodes. Instead of constructing a perception model as in Section 3.3.2, we attempted to directly calculate and use the actual perception neural network output distributions at these quadrature nodes. However, if the quadrature nodes change (e.g., by changing the input distribution or order of GPC), then new images must be captured and processed for the new quadrature nodes. In contrast, the perception model can be directly reused as it is agnostic to the quadrature nodes.

We also experimented with replacing only parts of M'_V with GPC as follows: first, we replaced the perception system with the perception model to create M'_V as in Section 3.3.2. Then, instead of replacing all of M'_V with a GPC model as in Section 3.3.3, we replaced only the vehicle control and dynamics systems (as the perception system had already been replaced by a polynomial model M_{per}). While the accuracy of this approach was the same as our main approach, the partially replaced model was about 3× slower than the fully replaced model during evaluation for all benchmarks.

3.7 RELATED WORK

Analysis and verification of vehicle systems. DryVR [155] is a system for verifying the safety of vehicle models that consist of a whitebox mode-switching control system composed with a blackbox dynamics system. DryVR obtains multiple samples from the blackbox dynamics and uses it give an over-approximate guarantee on the reachable set of states. In comparison, GAS focuses on perception and control systems which include blackbox components such as neural networks.

Pasareanu et al. [83] verify safety properties for vehicle systems with learning enabled components. They separately analyze the learning enabled components to derive guarantees of the component's behavior when certain assumptions about the input to that component hold. Such precise verification unfortunately does not scale to large image processing neural networks used to perceive the vehicle state from camera images. GAS is able to handle vehicle systems with such complex neural networks by using sampling to provide precise estimates of the probability that the vehicle will violate safety properties.

Jha et al. [156] and DeepDECS [157] analyze perception system uncertainty to create a correct by synthesis controller that satisfies a temporal logic safety constraint. GAS allows us to estimate the safety of existing, well known controllers for broad applications. Althoff et al. [158] focus on online verification of vehicle safety properties to adapt to unique traffic situations, assuming an upper bound on the noise of the perception system. In contrast, GAS allows the programmer to directly specify the perception system, and handles the dynamic nature of perception error through the perception model.

Musau et al. [159] complement complex neural network controllers with a safety controller which takes over if a runtime reachability analysis detects a potential collision. GAS's state distribution estimates can be used to determine the degree of reliability of the neural network controller and therefore the optimal criteria for switching to the safety controller.

Yang et al. [160] propose a runtime system for detecting environmental conditions that were not part of the training data for a pre-trained perception system. Cheng et al. [161]

ensure that the training data covers different combinations of environmental factors, based on their relative importance. Scenic [162] is a probabilistic language for specifying scenes in a virtual world for generating training images for perception neural networks in various environments. These works can be used when sampling images and/or prioritizing environments for training GAS's perception model in order to ensure that GAS's results are valid for the entire range of expected environments. For our evaluation, we used simple but realistic distributions of environmental parameters when generating images for training the perception model.

Recent surveys [136, 137] show that developers in the autonomous vehicle industry use vehicle simulation for regression testing of proposed vehicle system modifications. These surveys also show that developers cite the high cost of simulation as a major roadblock to integrating such tests more extensively into their workflow. GAS aims to help overcome this obstacle by significantly reducing the cost of testing vehicle systems as they are modified over time.

System simulation and modeling. Kewlani et al. [142] create and use GPC surrogate models of vehicle *dynamics* components. Lin et al. [140] do the same using small neural networks as surrogate models. Replacing only vehicle dynamics with GPC surrogates leads to negligible speedup as the perception and control components of the vehicle model contributed to 90% or more of the runtime of the MCS analysis of our benchmarks. GAS creates GPC surrogate models of *complete* vehicle systems; GAS's GPC model also replaces the expensive perception and control components. ARIsTEO [138] uses abstraction refinement to create surrogate models of cyber-physical systems with low dimensional inputs. GAS handles high-dimensional image inputs by first creating a perception model, and then creates a surrogate model of the reduced-dimensionality abstract vehicle model.

Li et al. [163] create small neural network surrogate models to calculate a *fitness function* for an autonomous driving system in various traffic scenarios. GAS's surrogate models instead estimate the state distribution of the vehicle over time, and this information can then be used to calculate various fitness metrics. Michelmore et al. [164] evaluate the safety of end-to-end Bayesian Neural Network (BNN) controllers. While GAS's perception model focuses on replacing regression DNNs, vehicle models with BNNs are an interesting target for extending GAS.

Simplifying complex perception and control systems. Cheng et al. [146] explore the correspondence between neural networks and polynomial regressions. Unlike their approach, GAS's perception model only needs to predict the *distribution* of neural network outputs, instead of individual input-output relationships.

Several approaches focus on sound and complete verification of safety properties of ACAS-Xu neural networks (e.g., that the network will not give a COC advisory when an intruder aircraft is directly ahead on a collision course [165, 166]). Others focus on providing probabilistic guarantees (e.g., bounding the probability that similar inputs will result in different advisories [167], or the probability of violating the safety property described above [168]). Unlike GAS and the approaches described below, these approaches focus on checking safety properties of the networks *in isolation*, that is, they do not take into account how the aircraft moves over time.

Several other recent approaches (e.g., [151, 169, 170]) focus on deterministically verifying properties of a *closed-loop system* containing ACAS-Xu neural networks and aircraft dynamics. In particular, Bak and Tran [169] checked the safety of the neural networks from [171] in 32 minutes on a 128 core machine. In contrast to those approaches, GAS does simulation-based probabilistic testing of ACAS-Xu neural networks coupled with aircraft dynamics. While GAS can find unsafe scenarios, it cannot provide non-probabilistic safety guarantees. Julian et al. [171] performed 1.5 million simulations to test their ACAS-Xu neural network, and Owen and Kochenderfer [172] performed 10 million simulations to determine whether horizontal or vertical advisories are safer in different states. We view GAS as a useful tool for speeding up such simulations and sanity checks of the closed-loop aircraft system when making experimental changes to it, before moving on to full verification.

Ghosh et al. [173] iteratively synthesize perception models and controllers guided by counterexamples to temporal logic safety properties. Hsieh et al. [174] create a perception model where the mean is calculated using piecewise linear regression and the allowable variance is calculated based on the controller code using program analysis tools like CBMC. Astorga et al. [175] create perception contracts, which describe the uncertainty that the perception system can generate, and the control and dynamics system can tolerate, without violating safety properties. Unlike these works, GAS's perception model is independent of the controller, and can be reused without requiring additional sampling when iterative changes are made to the controller during development.

Chapter 4: Aloe

With the end of Dennard scaling and the slowdown of Moore's law, hardware is becoming increasingly susceptible to operational errors, due to imperfect manufacturing, aging, and variations of environmental factors such as temperature, voltage, or radiation [4, 5, 6, 176, 177]. While many hardware errors are still routinely caught and corrected at the hardware level, some errors can propagate across the system stack and corrupt program data. Classical solutions for detecting and correcting errors come at a high cost. For example, indiscriminate instruction replication or n-modular redundancy can greatly increase a program's resource usage, even after applying various optimizations [70, 71].

Many modern applications have the freedom to produce results with a variable level of accuracy. Example workloads include audio and video processing, machine learning, big-data analytics, and probabilistic inference. These applications can tolerate *selective* error detection and recovery. That is, we can choose to check for and recover from errors in only certain critical components of the application.

Various automated and developer-assisted techniques enable developing acceptably reliable executions with reduced resource usage [48, 51, 52, 64, 65, 67, 68, 69, 72, 73, 87, 178, 179, 180]. Several reliability-aware languages like Relax [63] and Topaz [74], expose *try-check-recover blocks* to the developer, which have the form shown in Figure 4.1. Potentially unreliable code executes inside the `try` block. The `check` function analyzes the program state and attempts to detect potential errors in the computation. It can use various exact procedures (e.g., NP-hard problem solutions can be verified in polynomial time) or approximate procedures (e.g., anomaly detection [74]). If the check fails, the execution runs the recovery code inside the `recover` block. try-check-recover blocks are appealing as they empower developers with fine-grained control over customized recovery strategies and can provide hints to program analyses and compilers on how to generate code.

Analyzing the quantitative reliability and safety of computations has been a topic of increased interest in the programming languages community. Quantitative reliability denotes the probability with which a computation with unreliable operations produces the same (or acceptably similar) results as a fully reliable execution. For instance, a reliability of 0.99 states that 99% of executions of an unreliable program will generate the same result as a

```
try { code; }
check { check(programState); }
recover { code; }
```

Figure 4.1: The try-check-recover block

perfectly reliable execution. Example analyses of quantitative reliability include Rely [44], Chisel [36], Decaf [47], and Parallely [45]. Unfortunately, existing reliability analyses are significantly imprecise when analyzing computations with error detection and recovery mechanisms. At best, they treat these constructs as arbitrary conditional code, leading to a uniform reduction in the reliability of the computation. This is imprecise, because try-check-recover blocks will typically not reduce the reliability of the computation, even if the code within the `try` block is unreliable.

Our work. We present Aloe, the first static analysis of quantifiable reliability of programs that include constructs for selective error detection and recovery. The Aloe language exposes try-check-recover blocks and multiple detection and recovery choices to the developer. Aloe precisely computes the overall reliability of each try-check-recover block and uses this information to calculate the overall reliability of the whole program. Aloe supports both *perfect* checkers (that always detect whether or not an error occurred while executing the `try` block) and *imperfect* checkers (that can suffer from false positives and negatives). Aloe focuses on computations with *idempotent* code inside the `try` block (i.e. the code can be re-executed without previous executions affecting the results of the current execution) and `recover` blocks that re-execute the computation with different levels of reliability.

Aloe's analysis builds on top of Rely's precondition generator, which computes a quantitative reliability predicate for each statement in the program. Aloe specifies the rules for precisely analyzing try-check-recover blocks with multiple recovery strategies. Aloe also provides new rules for simplification of reliability predicates that improve its applicability. The analysis time is proportional to the number of statements in the program after unrolling bounded loops (as in Rely). Our approach directly extends to other similar analyses [36, 45, 47]. Aloe's precise analysis of try-check-recover blocks enables more precise analysis of programs using them, allowing them to save resources by making less conservative assumptions of the reliability of such protected sub-components.

Results. We implemented Aloe and applied it to eight programs from various benchmarks that were previously used in approximate computing research. We used the specifications of the unreliable hardware from EnerJ [65] and the specifications of the approximate checkers from Topaz [74]. Our experiments showed that Aloe's analysis of the try-check-recover blocks is significantly more precise than the existing analysis in Rely; Aloe verified all kernels identified in each benchmark for the reliability bound 0.9999. Aloe also verified the end-to-end reliability of each program's output for the bound 0.99. These results are orders of magnitude more precise than the baseline Rely analysis, which was not able to verify any

```
newPagerank = 0.15;
j = 0;
repeat MaxEdges {
  if j < NumIn {
     neighbor = Edges[j];
     numOut = Outlinks[neighbor];
     current = Pageranks[neighbor];
     temp = 0.85 * current;
     temp = temp / numOut;
     newPagerank = newPagerank + temp;
     j = j + 1;
  };
};
```

Figure 4.2: PageRank kernel

of the kernel or end-to-end reliability bounds. We also showed that kernels with try-check-recover blocks produced results with acceptable reliability even when the checkers themselves were imperfect (i.e., they could miss errors or report spurious errors).

4.1 EXAMPLE

PageRank is a link analysis algorithm that ranks web pages according to their importance. Figure 4.2 presents the implementation of the PageRank kernel for a single node in a graph. The PageRank of a node is a weighted sum of the PageRanks of each incoming edge. The PageRank is updated in this manner over multiple iterations. This computation is known to be tolerant to errors, partly due to its iterative nature.

We study the execution of this kernel on unreliable hardware, in which the arithmetic operations can produce incorrect results with some probability. For example, EnerJ [65] presents several approximation strategies that provide arithmetic instructions that save energy but produce erroneous results with probability 10^{-6}, 10^{-4}, or 10^{-2}. We model such instructions using the probabilistic choice statement $e_{correct}$ $[p]$ e_{wrong}, which states that the operation produces the correct result with probability p, and otherwise produces some incorrect result.

4.1.1 Try-Check-Recover Blocks

One possible approach for increasing the reliability of computations running on unreliable hardware is to check if an error occurred during the execution of the computation, and redo the computation using precise hardware if such an error is detected. Figure 4.3 presents this approach for the PageRank kernel. The program first runs the statements in the `try`

```
newPagerank = 0.15;
j = 0;
repeat MaxEdges {
  if j < NumIn {
    neighbor = Edges[j];
    numOut = Outlinks[neighbor];
    current = Pageranks[neighbor];
    try {
      temp = 0.85 * current [0.999] rand();
      temp = temp / numOut [0.999] rand();
      sum = newPagerank + temp [0.999] rand();
    } check { checker(sum, current, numOut, newPagerank) }
    recover {
      temp = 0.85 * current [1] rand();
      temp = temp / numOut [1] rand();
      sum = newPagerank + temp [1] rand();
    };
    newPagerank = sum;
    j = j + 1;
  };
};
```

Figure 4.3: Protected PageRank kernel for use on unreliable hardware

block. The arithmetic operations in the `try` block are unreliable and only produce the correct value with probability 0.999. Otherwise, they produce a random garbage value. The execution continues until the end of the `try` block, even if an arithmetic operation produces an incorrect result.

Checking and recovery. Next, the program evaluates the `check` function on the program state to detect if there was an error in the computation. If the checker detects an error, it runs the `recover` block, which re-executes the computation in the `try` block on fully reliable hardware. If the checker can identify all errors, the overall computation becomes fully reliable. Moreover, if the execution of the checker is inexpensive, the overall computation run on unreliable hardware may be more energy efficient than the original computation in Figure 4.2 run on reliable hardware.

For full reliability, the checker must ensure that all output variables have correct values. Error detection mechanisms can either be implemented in hardware [75, 181] or in software. Software techniques include re-executing the computation in the `try` block and comparing the results, verifying the result with a verification algorithm (e.g. the results of NP-hard problems are verifiable in polynomial time), or identifying outliers using machine learning [74]. For our example, we assume that the hardware sets an internal flag when an error occurs in the `try` block (similar to Relax [63]), and this internal flag is then read by

```
S_try;
if ( checker(...) ) {
    skip;
} else {
  S_rec;
}
```

Figure 4.4: Recovery mechanism implemented via a conditional statement

the `check` function.

4.1.2 Verification of Reliability

We define reliability as the probability that the results of the computation are correct (equal to the results of an execution with no errors). We want to verify that the calculated variable `newPagerank` is fully reliable, i.e., its reliability is equal to 1.

Encoding detection and recovery in Rely. The existing quantitative reliability analysis from Rely cannot be used to accurately calculate the reliability of a try-check-recover block. To represent this computation in Rely, one can convert the try-check-recover statement to a conditional statement, as shown in Figure 4.4. In Figure 4.4, $\texttt{S}_{\texttt{try}}$ and $\texttt{S}_{\texttt{rec}}$ represent the instructions in the `try` and `recover` block, respectively. The semantics of this computation closely mirror those of the original try-check-recover block. However, Rely's analysis cannot infer that $\texttt{S}_{\texttt{rec}}$ only executes when $\texttt{S}_{\texttt{try}}$ produces incorrect results and that it eliminates the error produced by $\texttt{S}_{\texttt{try}}$. Instead, Rely conservatively assumes that errors in $\texttt{S}_{\texttt{try}}$ can remain uncorrected. On the other hand, Aloe's precise analysis of try-check-recover blocks allows it to correctly determine that the overall computation is fully reliable.

4.1.3 Results

For `MaxEdges` $= 8$, Aloe's analysis automatically shows that the reliability of `newPagerank` is 1.0 due to the recovery mechanism detecting and fixing all errors. The analysis runs in 3ms. In contrast, Rely's analysis is too conservative; it calculates a reliability of 0.976. Aloe can analyze several other interesting scenarios which we describe next.

Recovery may also be unreliable. If the `recover` block is also executed on unreliable hardware, then it may also experience errors. Suppose arithmetic instructions in the `recover` block can produce an incorrect result with probability 10^{-4}. Aloe computes that the reliability of the `try` and `recover` blocks in isolation is 0.997 and 0.9997 respectively. Since both `try` and `recover` must produce incorrect results for the overall try-check-recover block to produce incorrect results, the overall reliability of the try-check-recover block is 0.9999991. When `MaxEdges` = 8, the reliability of the entire kernel is 0.999993. In this way, it is possible to combine multiple unreliable sub-components to produce more reliable components. In contrast, Rely's analysis calculates the reliability of the kernel as 0.976.

Checkers may be unreliable. We also analyzed the impact of an imperfect `check` function on the reliability of `newPagerank`. For a `check` function that detects 95% of erroneous runs, and may detect spurious errors 5% of the time, Aloe calculated that the `newPagerank` reliability was 0.9999.

4.2 BACKGROUND

Aloe's quantitative reliability analysis builds on the similar analysis of Rely [44], which we review in brief below.

Reliability predicates. We generate reliability predicates to constrain the reliability of an approximate program. A reliability predicate Q has the following form:

$$\begin{aligned} Q &:= r \leq R_f \mid Q \wedge Q \\ R_f &:= r \mid \mathcal{R}(O) \mid r \cdot \mathcal{R}(O) \end{aligned} \tag{4.1}$$

A *reliability factor* (R_f) is either a number $r \in [0,1]$, a *joint reliability predicate* $\mathcal{R}(O)$ representing the probability that a set of variables O has the same values in an approximate execution as an exact, error-free execution, or a product of the two. A reliability predicate (Q) is either a comparison between a number and a reliability factor or a conjunction of multiple reliability predicates.

For example, we can specify the constraint that the reliability of some variable x in a program is at least 0.99 (99%) using the reliability predicate $0.99 \leq \mathcal{R}(\{x\})$. In this predicate, $\mathcal{R}(\{x\})$ is the probability that an approximate execution of the program generates the same value for x as an exact, error-free execution.

Reliability precondition generation. The reliability precondition generator is a function $C \in S \times Q \mapsto Q$ that takes as inputs a statement and a postcondition that must be satisfied after executing the statement and produces the corresponding precondition as the output. We use the existing precondition generation rules as in Rely and extend them to Aloe's new recovery constructs as described in Sections 4.4 and 4.5. These rules include:

$$\begin{aligned} C(\, x = e,\, Q\,) &= Q\; [\, \mathcal{R}(\, \rho(e) \cup X\,) \;/\; \mathcal{R}(\, \{x\} \cup X\,)\,] \\ C(\, x = e_1\; [r]\; e_2,\, Q\,) &= Q\; [\, r \cdot \mathcal{R}(\, \rho(e_1) \cup X\,) \;/\; \mathcal{R}(\, \{x\} \cup X\,)\,] \\ C(\, \texttt{if}\; x\; \{S_1\}\; \texttt{else}\; \{S_2\},\, Q\,) &= C(\, S_1,\, Q\,) \wedge C(\, S_2,\, Q\,) \end{aligned} \tag{4.2}$$

For a simple assignment $x = e$, any reliability specification containing x is updated such that x is replaced by the variables occurring in e ($\rho(e)$) as the reliability of x is only dependent on the reliability of variables used in e. For a probabilistic assignment $x = e_1\; [r]\; e_2$, the reliability of x is equal to r (the probability of assigning the value of the correct expression e_1 to x) times the reliability of variables occurring in e_1. Conditionals are analyzed as a nondeterministic choice between the *if* and *else* branches. The precondition for a conditional statement is the conjunction of the preconditions generated from the two branches. The reliability of the branch variable is incorporated into the analysis via conditional flattening ([44, Section 5.1]). Bounded loops in Rely are unrolled into a sequence of nested conditionals.

Substitution. The substitution for reliability predicates in Rely, $e_0\,[\,e_2 \,/\, e_1\,]$, replaces all occurrences of the expression e_1 with the expression e_2 within the expression e_0. The substitution matches set patterns, e.g., the pattern $R(\{x\} \cup X)$ is a joint reliability factor that contains the variable x, alongside with the remaining variables in the set X. The result of $\mathcal{R}(\{x, z\})\,[\,\mathcal{R}(\{y\} \cup X) \,/\, \mathcal{R}(\{x\} \cup X)\,]$ is $\mathcal{R}(\{y, z\})$.

4.3 LANGUAGE

4.3.1 Syntax

Figure 4.5 presents the syntax of the Aloe language, which is similar to the Rely language [44] with added support for recovery mechanisms. Aloe uses the probabilistic choice statement from Parallely [45] to explicitly represent unreliable operations and their failure probability (in lieu of the `+.` notation and implicit hardware model from Rely).

To analyze a program, Aloe requires three main components:

- *Program:* a program written in the Aloe language.
- *Approximation models:* specifies the probabilities that instructions produce incorrect results. We can define different models for the `try` and `recover` blocks.

n	$\in \mathbb{N}$	*quantities*
$\mathtt{m}$	$\in \mathbb{N} \cup \mathbb{F}$	*values*
r	$\in [0,\ 1.0]$	*probability*
x, b	$\in \mathit{Var}$	*variables*
a	$\in \mathit{ArrVar}$	*array variables*
f	$\in \mathit{Func}$	*external functions*
op	$\in \{+, -, \ldots\}$	*arithmetic ops*
Exp	$\rightarrow m \mid x \mid f(\mathit{Exp}^*)$	*expressions*
	$\mid \mathit{Exp}\ op\ \mathit{Exp}$	
t	$\rightarrow$ `int<n>` $\mid$ `float<n>`	*basic types*
D	$\rightarrow$ t x $\mid$ t a$[n^+]$	*variable*
	$\mid D$ `;` D	*declarations*
P	$\rightarrow D$ `;` S	*program*

recovery $\rightarrow$	
redo$[n]$	redo up to n times
$\mid$ *redo*$[\psi]$	redo on reliability model ψ

$S \rightarrow$	
`skip`	empty program
$\mid x$ `=` Exp	assignment
$\mid x$ `=` $\mathit{Exp}\ [r]\ \mathit{Exp}$	probabilistic choice
$\mid S$ `;` S	sequence
$\mid x$ `=` a`[`Exp^+`]`	array load
$\mid a$`[`Exp^+`]` `=` Exp	array store
$\mid$ `if` Exp $\{S\}$ `else` $\{S\}$	branching
$\mid$ repeat n {S}	repeat n times
$\mid x$ `=` *(T)Exp*	cast
$\mid$ `try` {S} `check` $\{\mathit{Exp}\}$ `recover` {recovery}	try-check-recover

Figure 4.5: Aloe language syntax

- *Specification of checkers:* specifies the false-positive rate ($p_{_{FP}}$) and the false-negative rate ($p_{_{FN}}$) of `check` functions used to check for errors.

4.3.2 Semantics

References. A *reference Ref* is a pair $\langle n_b, \langle n_1, ..., n_k\rangle\rangle$ that consists of a base address n_b and a dimension descriptor $\langle n_1, ..., n_k\rangle$. References describe the location and the dimension of variables in the heap.

Frames, stacks, and heaps. A *frame* σ is an element of the domain $\mathrm{E} = \mathit{Var} \rightarrow \mathit{Ref}$, which is the set of finite maps from program variables to references. A heap $h \in H = \mathbb{N} \rightarrow \mathbb{N} \cup \mathbb{F}$ is a finite map from addresses (integers) to values. Values can be integers or floats. An environment $\epsilon \in \mathrm{E} \times H$ is a pair of a frame and a heap.

Approximation models. An unreliable program executes within *approximation model* ψ, which contains the parameters for approximation (e.g., the probability of selecting the correct expression in a probabilistic choice statement). We define a special reliable model 1_ψ, which executes the program without approximations.

Expressions. Aloe uses typical imperative language expression semantics. For example, variables are evaluated by looking up the variable in the current stack frame and then retrieving the variable value from the heap. In Aloe, most expressions always evaluate

S-Assign-Prob-True
$$\frac{}{\langle x\ \texttt{=}\ e_1\ [r]\ e_2, \sigma, h\rangle \xrightarrow{C,\psi(r)}_{\psi} \langle x\ \texttt{=}\ e_1, \sigma, h\rangle}$$

S-Assign-Prob-False
$$\frac{}{\langle x\ \texttt{=}\ e_1\ [r]\ e_2, \sigma, h\rangle \xrightarrow{F,1-\psi(r)}_{\psi} \langle x\ \texttt{=}\ e_2, \sigma, h\rangle}$$

S-Try
$$\frac{\langle \text{S1}, \sigma, h\rangle \xrightarrow{\lambda,p}_{\psi} \langle \text{S1}', \sigma', h'\rangle}{\langle \texttt{try}\ \{\text{S1}\}\ \texttt{check}\ \{\text{e}\}\ \texttt{recover}\ \{\text{S2}\}, \sigma, h\rangle \xrightarrow{\lambda,p}_{\psi} \langle \texttt{try}\ \{\text{S1'}\}\ \texttt{check}\ \{\text{e}\}\ \texttt{recover}\ \{\text{S2}\}, \sigma', h'\rangle}$$

S-Check-1
$$\frac{\langle e, \sigma, h\rangle \xrightarrow{C,1}_{\psi} \langle e', \sigma, h\rangle}{\langle \texttt{try}\ \{\texttt{skip}\}\ \texttt{check}\ \{\text{e}\}\ \texttt{recover}\ \{\text{S2}\}, \sigma, h\rangle \xrightarrow{C,1}_{\psi} \langle \texttt{try}\ \{\texttt{skip}\}\ \texttt{check}\ \{\text{e'}\}\ \texttt{recover}\ \{\text{S2}\}, \sigma, h\rangle}$$

S-Check-True
$$\frac{n \neq 0}{\langle \texttt{try}\ \{\texttt{skip}\}\ \texttt{check}\ \{\text{n}\}\ \texttt{recover}\ \{\text{S2}\}, \sigma, h\rangle \xrightarrow{C,1}_{\psi} \langle \texttt{skip}, \sigma, h\rangle}$$

S-Check-False
$$\frac{}{\langle \texttt{try}\ \{\texttt{skip}\}\ \texttt{check}\ \{0\}\ \texttt{recover}\ \{\text{S2}\}, \sigma, h\rangle \xrightarrow{C,1}_{\psi} \langle S2, \sigma, h\rangle}$$

Figure 4.6: Semantics of new statements introduced in Aloe

correctly; unreliability is only introduced via specific statements, or via expressions that are calls to external unreliable functions.

Statements. The small-step relation $\langle s, \sigma, h\rangle \xrightarrow{\lambda,p}_{\psi} \langle s', \sigma', h'\rangle$ defines the program evaluating in a stack frame σ, and heap h with the transition label λ with probability p. The label λ is either C, indicating the correct transition, or F, indicating the faulty transition. Figure 4.6 presents the semantics rules for the new statements introduced in Aloe. The semantics of other Aloe statements are derived from those of Rely. The next paragraphs describe the new statements in detail.

Probabilistic choice. The probabilistic choice statement $x = e_C\ [r]\ e_F$ assigns the value of the correct expression (e_C) to x with probability $\psi(r)$, or otherwise assigns the value of the faulty expression (e_F). The probability is provided as a symbolic variable r whose concrete value $\psi(r)$ is defined in the approximation model ψ. It can be used to model many approximate computations. For example, we can model unreliable arithmetic instructions as $z = x\,\langle op\rangle\,y\ [r]$ `randVal()`, which models an arithmetic operation that can produce a garbage result with probability $1 - \psi(r)$.

S-Assign-Prob-Exact

$$\frac{}{\langle x = e_1\ [r]\ e_2, \sigma, h\rangle \xrightarrow{C,1}_{1_\psi} \langle x = e_1, \sigma, h\rangle}$$

S-TRY-Exact

$$\frac{}{\langle \texttt{try}\ \{\texttt{skip}\}\ \texttt{check}\ \{\mathrm{e}\}\ \texttt{recover}\ \{\texttt{S2}\}, \sigma, h\rangle \xrightarrow{C,1}_{1_\psi} \langle \texttt{skip}, \sigma, h\rangle}$$

Figure 4.7: Exact execution semantics of statements

Idempotency. Aloe requires that the computations inside the `try` and `recover` blocks are *idempotent*. That is, one execution of the computation must not influence the result of a future execution of the same computation on the same input variables. This is typically achieved by ensuring that the input and output variables of the computations are disjoint sets. In addition, they must not cause any side effects that are visible to future executions of those computations.

Try-Check-Recover. The recovery mechanism `try` $\{S_1\}$ `check` {e} `recover` $\{S_2\}$ first executes the statement S_1 *in its entirety.* It then evaluates the check expression e, which is used to check if an error occurred during the execution of S_1. If an error is detected ($e = 0$), then it executes S_2. As the computation in S_1 is idempotent, the recovery code in S_2 can continue from the current state. This behavior is similar to that of `try-catch` statements commonly used in exception handling, except that S_1 is fully executed before the check.

Exact execution semantics. In the fully reliable model 1_ψ, probabilistic choice statements always assign the value of the correct expression. In try-check-recover statements, the `try` block always executes correctly and the check always passes. Figure 4.7 shows the exact execution semantics of relevant statements.

4.3.3 Reliability

Aggregate semantics. We use the following aggregate semantics from Rely to define the reliability of a program:

Definition 4.1 (Trace semantics for programs). $\langle \cdot,\ \epsilon\rangle \overset{\tau,p}{\Longrightarrow}_\psi \epsilon'$ represents the program trace $\langle \cdot, \epsilon.\sigma, \epsilon.h\rangle \xrightarrow{\lambda_1,p_1}_\psi \ \ldots\ \xrightarrow{\lambda_n,p_n}_\psi \langle \texttt{skip}, \epsilon'.\sigma, \epsilon'.h\rangle$ where $\tau = \lambda_1, \ldots, \lambda_n$, and $p = \prod_{i=1}^{n} p_i$

This big-step semantics is the reflexive transitive closure of the small-step global semantics for programs and records a *trace* of the program. A trace τ is a sequence of small step global transitions. The probability of the trace is the product of the probabilities of each transition. The trace semantics are defined for environments ϵ (pairs of frames σ and heaps h).

Definition 4.2 (Aggregate semantics for programs)**.** $\langle \cdot,\ \epsilon\rangle \Downarrow^{p}_{\psi} \epsilon'$ represents the aggregate of traces $\langle \cdot,\ \epsilon\rangle \xrightarrow{\tau, p_\tau}_{\psi} \epsilon'$ where $p = \sum\limits_{\tau \in \mathrm{T}} p_\tau$

The big-step aggregate semantics enumerates over the set of all finite length traces and sums the aggregate probability that a program starts in an environment ϵ and terminates in an environment ϵ'. It accumulates the probability over all possible traces τ that end up in the same final state.

Paired execution semantics. For reliability analysis, we define a *paired execution semantics* that couples an original execution of a program with an approximate execution, following the definition from Rely:

Definition 4.3 (Paired execution semantics)**.** $\langle s,\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_{\psi} \langle \epsilon',\ \varphi'\rangle$ represents the paired execution semantics where $\langle s,\ \epsilon\rangle \Downarrow^{1}_{1_\psi} \epsilon'$ and where $\varphi'(\epsilon'_a) = \sum\limits_{\epsilon_a} \varphi(\epsilon_a) \cdot p_a$ and $\langle s,\ \epsilon_a\rangle \Downarrow^{p_a}_{\psi} \epsilon'_a$

This relation states that from a configuration $\langle \epsilon,\ \varphi\rangle$ consisting of an environment ϵ and an *environment distribution* $\varphi \in \Phi$, the paired execution yields a new configuration $\langle \epsilon',\ \varphi'\rangle$. The fully reliable execution reaches the environment ϵ' from the environment ϵ with probability 1 (expressed by the deterministic execution, 1_ψ). The environment distributions φ and φ' are probability mass functions that map an environment to the probability that the unreliable execution is in that environment. In particular, φ is a distribution on environments before the execution of s, whereas φ' is the distribution on environments after executing s.

Reliability transformer. Reliability predicates and the semantics of programs are connected through the view of a program as a reliability transformer, following the definition from Rely:

Definition 4.4 (Reliability transformer relation)**.** $\psi \models \{Q_{pre}\}\ s\ \{Q_{post}\}$ is equivalent to stating $\forall \epsilon, \varphi, \epsilon', \varphi'.\ \ (\epsilon, \varphi) \in [\![Q_{pre}]\!] \wedge \langle s,\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_{\psi} \langle \epsilon',\ \varphi'\rangle \implies (\epsilon', \varphi') \in [\![Q_{post}]\!]$

Similar to the standard Hoare triple relation, if an environment and distribution pair $\langle \epsilon,\ \varphi\rangle$ satisfy a reliability predicate Q_{pre}, then the program's paired execution transforms them into a new pair $\langle \epsilon',\ \varphi'\rangle$ that satisfies a predicate Q_{post}.

4.3.4 Preprocessing

To simplify the analysis, we perform multiple preprocessing steps beforehand.

```
try{ s1 }
check{e}
recover{ redo[n] }
```

$\mapsto$

```
try{ s1 }
check{e}
recover{
    try{ s1 }
    check{e}
    recover{ redo[n-1] }
}
```

Figure 4.8: Desugaring `redo[n]`

Desugaring recovery mechanisms. The Aloe language provides several syntactic constructs to help write programs:

- *`redo[n]`:* If the recovery mechanism of a try-check-recover block is `redo[1]`, Aloe replaces it with the code in the `try` block. If the recovery mechanism is `redo[n]` where $n > 1$, Aloe replaces it with nested try-check-recover blocks as shown in Figure 4.8.
- *`redo[`ψ_2`]`:* Another common recovery mechanism is to execute the program on a different hardware configuration with higher reliability. In this case, Aloe replaces `redo[`ψ_2`]` with the code in the `try` block, but replaces all probabilities in probabilistic choice statements with those from ψ_2.
- *Bounded loops:* As in Rely, Aloe supports finite bounded loops, that it unrolls prior to the reliability analysis.

4.4 RELIABILITY ANALYSIS: PERFECT CHECKER

Aloe's reliability calculation extends that of Rely by adding additional rules for generating reliability preconditions for the try-check-recover mechanism. These rules are only applicable to postconditions whose joint reliability factor contains at least one variable updated within the try-check-recover block. All other postconditions are unaffected by the try-check-recover block, and become part of the generated precondition unmodified, as in [44].

Consider the following try-check-recover block:

```
try { s1; } check { e } recover { s2; }
```

For a try-check-recover block to satisfy a predicate Q after execution, it should be satisfied by all possible execution paths through the try-check-recover block.

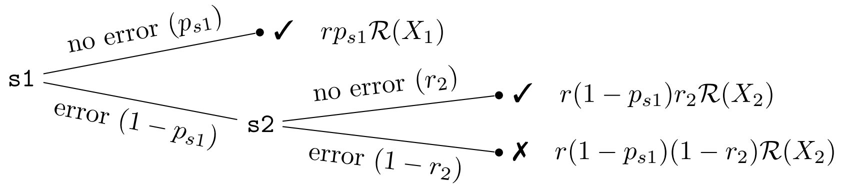

s1
no error (p_{s1})
✓ $rp_{s1}\mathcal{R}(X_1)$
error ($1 - p_{s1}$)
s2
no error (r_2)
✓ $r(1-p_{s1})r_2\mathcal{R}(X_2)$
error ($1 - r_2$)
✗ $r(1-p_{s1})(1-r_2)\mathcal{R}(X_2)$

Figure 4.9: Probabilities for perfect checkers

4.4.1 Precondition Generation

Assume a perfect check. Figure 4.9 shows the tree of possible executions of the try-check-recover block. If the checker detects an error, then `s2` is executed and the results of `s1`'s execution are discarded. There are two possible execution paths through the try-check-recover block.

1. *`s1` executes correctly and the checker passes:* In this case the check ensures that instructions in `s1` do not degrade the reliability of any variables calculated in the `try` block. The reliability of these variables only depends on the reliability of values flowing into them in `s1`.

2. *At least one instruction in `s1` executes incorrectly, the checker fails indicating an error, and `s2` executes:* As we ensure that the computation in `s1` is idempotent, the error in `s1` does not affect the reliability of variables calculated in `s2`. Instead, it depends on the probability that statements in `s2` update variables reliably and the reliability of values flowing into them in `s2`.

To handle these two scenarios, Aloe uses and combines the preconditions generated independently for `s1` and `s2`. Suppose the try-check-recover block must satisfy the postcondition $c \leq r \cdot \mathcal{R}(X)$. Similar to the precondition generation steps in Rely, we need to replace $\mathcal{R}(X)$ in the postcondition with the *total* probability of reaching a state where the variables in X have the correct value after executing the try-check-recover block.

Case 1. Suppose that running Rely precondition generation on `s1` results in the predicate $c \leq r \cdot r_{s1} \cdot \mathcal{R}(X_{s1})$. Here, r_{s1} is the minimum probability that all instructions in `s1` that affect variables in X execute correctly, and $\mathcal{R}(X_{s1})$ is the reliability of variables that flow into X in `s1`. However, the check only passes if *all* instructions in the `try` block execute correctly (as they may impact the checker), not just the ones affecting variables in X. Instead of r_{s1}, the probability that X is calculated correctly *via case 1* depends on the probability that *all* instructions in `s1` execute correctly, which we denote as p_{s1} (We discuss how to calculate p_{s1}

in Section 4.4.2). Therefore, the total contribution of case 1 towards the probability that variables in X are calculated correctly is $p_{s1} \cdot \mathcal{R}(X_{s1})$.

Case 2. `s1` executes incorrectly and fails the check with probability $1 - p_{s1}$. Suppose that running Rely precondition generation on `s2` results in the predicate $c \leq r \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$. Here, r_{s2} is the minimum probability that all instructions in `s2` that affect variables in X execute correctly, and $\mathcal{R}(X_{s2})$ is the reliability of variables that flow into X in `s2`. However, case 2 only occurs if `s1` produces incorrect results. Therefore, the total contribution of case 2 towards the probability that variables in X are calculated correctly is $(1 - p_{s1}) \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$.

Calculating $\mathcal{R}(X)$. We can now calculate the total probability that variables in X are calculated correctly as the sum of the probabilities of the two cases:

$$\mathcal{R}(X) = p_{s1} \cdot \mathcal{R}(X_{s1}) + (1 - p_{s1}) \cdot r_{s2} \cdot \mathcal{R}(X_{s2}) \tag{4.3}$$

We extended the reliability predicates with the addition operator following the usual meaning of addition, since reliability factors denote probabilities over program states [44]. We can simplify this expression using the following ordering proposition from [44], which states that for two sets of variables A and B, if $B \subseteq A$ then $\mathcal{R}(A) \leq \mathcal{R}(B)$. Therefore,

$$\begin{aligned}\mathcal{R}(X) &= p_{s1} \cdot \mathcal{R}(X_{s1}) + (1 - p_{s1}) \cdot r_{s2} \cdot \mathcal{R}(X_{s2}) \\ &\geq p_{s1} \cdot \mathcal{R}(X_{s1} \cup X_{s2}) + (1 - p_{s1}) \cdot r_{s2} \cdot \mathcal{R}(X_{s1} \cup X_{s2}) \\ &\geq (p_{s1} + (1 - p_{s1})\ r_{s2}) \cdot \mathcal{R}(X_{s1} \cup X_{s2})\end{aligned} \tag{4.4}$$

Following the subsumption rules in [44, Proposition 2], we can replace $\mathcal{R}(X)$ with this probability in the postcondition to get the precondition:

$$c \leq r\ (p_{s1} + (1 - p_{s1})\ r_{s2}) \cdot \mathcal{R}(X_{s1} \cup X_{s2}) \tag{4.5}$$

We present the proof of soundness in Section 4.6. Note that some variables in X may not be read or written to by either block. Such variables become part of $X_{s1} \cup X_{s2}$ unchanged, following Rely's precondition generation rules.

4.4.2 Minimum Probability of Success of `s1`

The `check` function ensures *complete* correctness of the execution of the `try` block. Recall that r_{s1} is the probability that `s1` does not make any error that affects the variables tracked by the postcondition (X). We must separately calculate the probability p_{s1} that the `try` block executes without *any* error and therefore satisfies the check. For example, suppose the `try` block contains the statements $y = x\ [p_0]\ 0$ and $z = x\ [p_1]\ 0$. If the postcondition only tracks the correctness of y, then r_{s1} only depends on p_0. However, the checker also ensures that z is correct. Therefore, p_{s1}, the probability that the check passes, also depends on p_1.

To compute p_{s1}, we consider all possible control flow paths within the `try` block. If a control flow path has multiple probabilistic choice statements with correct execution probabilities $p_1, p_2, \ldots, p_n$, then the probability that that path as a whole executes correctly is equal to their product. Finally, p_{s1} is the minimum of the products over all paths.

4.4.3 Simplifying the Preconditions of `s1` and `s2`

In Section 4.4.1, we assumed that the Rely precondition generation algorithm generated a simple precondition of the form $c \leq r \cdot r_{s1} \cdot \mathcal{R}(X_{s1})$ for `s1`. In general, the Rely precondition generation algorithm may generate multiple precondition conjuncts from the postcondition. When generating preconditions for `s1` or `s2` for Aloe's analysis, we can combine the precondition conjuncts into a single precondition that *subsumes* the original precondition conjuncts using Rely's subsumption rules (Proposition 2 of [44]).

Suppose we have the following precondition for `s1`:

$$c \leq r_{11} \cdot \mathcal{R}(X_{11}) \wedge c \leq r_{12} \cdot \mathcal{R}(X_{12}) \wedge \ldots c \leq r_{1n} \cdot \mathcal{R}(X_{1n}) \tag{4.6}$$

Similarly, suppose we have the following precondition for `s2`:

$$c \leq r_{21} \cdot \mathcal{R}(X_{21}) \wedge c \leq r_{22} \cdot \mathcal{R}(X_{22}) \wedge \ldots c \leq r_{2n} \cdot \mathcal{R}(X_{2n}) \tag{4.7}$$

Using the subsumption rule, we can replace `s1`'s precondition with $c \leq r \cdot r_{s1} \cdot \mathcal{R}(X_{s1})$ and `s2`'s precondition with $c \leq r \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$, such that

$$r_{s1} = \min(r_{11}, r_{12}, \ldots, r_{1n}) \qquad r_{s2} = \min(r_{21}, r_{22}, \ldots, r_{2n}) \tag{4.8}$$

$$X_{s1} = X_{11} \cup X_{12} \cup \ldots, X_{1n} \qquad X_{s2} = X_{21} \cup X_{22} \cup \ldots, X_{2n} \tag{4.9}$$

4.4.4 Example: Redo as Recovery

Simply re-doing the computation that experienced an error is a common recovery pattern. In Figure 4.10, the left side shows the analyzed code, and the right side shows Aloe's generated preconditions for the postcondition $0.99 \leq \mathcal{R}(\{x\})$. The try-check-recover block

```
try {
  x = y [$p_1$] rand();
} check (f(x, y))
recover {
  x = y [$p_2$] rand();
}
```

$\mapsto$

```
{$0.99 \leq (p_{s1} + (1 - p_{s1})p_2)\mathcal{R}(\{y\})$}
try { /* $p_{s1} = p_1$ */
  {$0.99 \leq p_1 \cdot \mathcal{R}(\{y\})$}
  x = y [$p_1$] rand();
  {$0.99 \leq \mathcal{R}(\{x\})$}
} check (f(x, y))
recover {
  {$0.99 \leq p_2 \cdot \mathcal{R}(\{y\})$}
  x = y [$p_2$] rand();
  {$0.99 \leq \mathcal{R}(\{x\})$}
}
{$0.99 \leq \mathcal{R}(\{x\})$}
```

Figure 4.10: Example: recovering with a perfect checker via redo

Table 4.1: `check` function correctness probabilities

`check` **function detects error?**	**Yes**	**No**
`s1` **experiences error**	p_{TP}	p_{FN}
`s1` **does not experience error**	p_{FP}	p_{TN}

increases the reliability of x from $p_1 \cdot \mathcal{R}(\{y\})$ to $(p_1 + (1 - p_1)p_2) \cdot \mathcal{R}(\{y\})$. Similarly, Aloe's analysis shows that repeating the calculation in the `try` block on the same hardware at most n times upon detecting errors can increase the reliability of x to $(1 - (1 - p_1)^n) \cdot \mathcal{R}(\{y\})$. Such precise reliability calculations cannot be done with existing methods such as Rely.

4.5 RELIABILITY ANALYSIS: IMPERFECT CHECKER

Aloe's reliability analysis can also be extended to `check` functions that fail to capture all errors, or detect spurious errors. We specify the characteristics of imperfect `check` functions through their false positive probability (p_{FP}) and false negative probability (p_{FN}). We define them in Table 4.1, together with the probabilities of true positives (p_{TP}) and true negatives (p_{TN}). These probabilities impact the overall reliability of try-check-recover blocks.

4.5.1 Precondition Generator

Consider the same program and postcondition $c \leq r \cdot \mathcal{R}(X)$ as in Section 4.4. Figure 4.11 shows the tree of possible executions of the try-check-recover block for an imperfect checker. In this case, there are three possible execution paths through the try-check-recover block that result in a correct calculation of variables in X.

1. *`s1` executes correctly and the checker passes:* This case is similar to case 1 for the

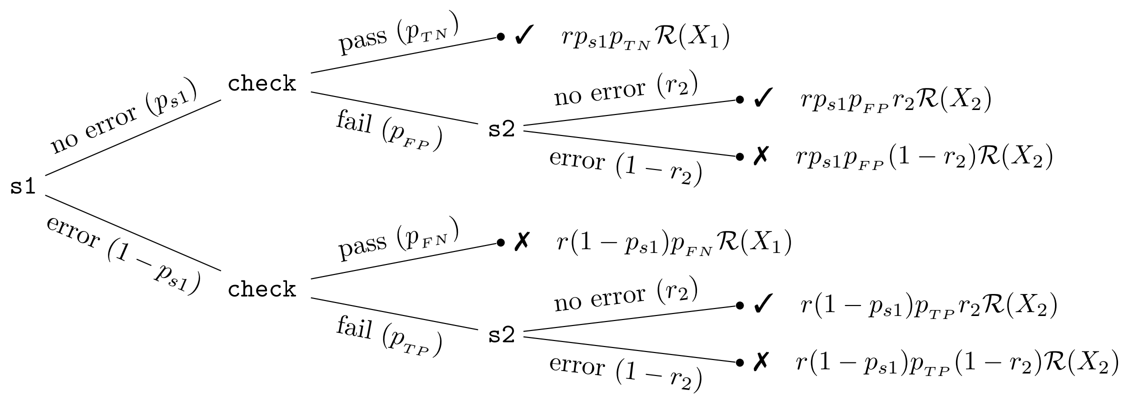

pass (p_{TN})
$rp_{s1}p_{TN}\mathcal{R}(X_1)$
check
no error (p_{s1})
no error (r_2)
$rp_{s1}p_{FP}r_2\mathcal{R}(X_2)$
fail (p_{FP})
s2
error ($1-r_2$)
$rp_{s1}p_{FP}(1-r_2)\mathcal{R}(X_2)$
s1
pass (p_{FN})
$r(1-p_{s1})p_{FN}\mathcal{R}(X_1)$
error ($1-p_{s1}$)
check
no error (r_2)
$r(1-p_{s1})p_{TP}r_2\mathcal{R}(X_2)$
fail (p_{TP})
s2
error ($1-r_2$)
$r(1-p_{s1})p_{TP}(1-r_2)\mathcal{R}(X_2)$

Figure 4.11: Probabilities for imperfect checkers

perfect checker. However, the check can still fail with probability p_{FP}.

2. *`s1` executes correctly but the checker fails, and `s2` executes:* This case consists of situations where an error free execution is classified as having an error.

3. *At least one instruction in `s1` executes incorrectly, the checker fails and indicates an error, and `s2` executes:* This case is also similar to case 2 for the perfect checker. However, the check can still pass with probability p_{FN}.

As before, we start with the postcondition $c \leq r \cdot \mathcal{R}(X)$ and combine the preconditions generated independently for `s1` and `s2`.

Case 1. Suppose that running Rely precondition generation on `s1` results in the predicate $c \leq r \cdot r_{s1} \cdot \mathcal{R}(X_{s1})$. The statement `s1` executes correctly and the check passes with probability $p_{s1} \cdot p_{TN}$. If these two events occur, then the probability that X is calculated correctly only depends on the reliability of variables flowing into variables in X that are updated in `s1` ($\mathcal{R}(X_{s1})$). The probability that these two events occur *and* variables in X are calculated correctly in the first scenario is therefore $p_{s1} \cdot p_{TN} \cdot \mathcal{R}(X_{s1})$.

Case 2. Suppose that running Rely precondition generation on `s2` results in the predicate $c \leq r \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$. The statement `s1` executes correctly but the check fails with probability $p_{s1} \cdot p_{FP}$. If these two events occur, then `s2` is run, so the probability that X is calculated correctly is $r_{s2} \cdot \mathcal{R}(X_{s2})$. The probability that these two events occur *and* variables in X are calculated correctly in the second scenario is therefore $p_{s1} \cdot p_{FP} \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$.

Case 3. `s1` executes incorrectly and the check fails with probability $(1 - p_{s1})p_{TP}$. If these two events occur, then `s2` is run, so the probability that X is calculated correctly is

```
try {
  x = y [p1] rand();
} check (f(x, y))
recover {
  x = y [p2] rand();
}
```

$\mapsto$

```
{0.99 ≤ (ps1 pTN + ps1 pFP p2
  +(1 − ps1) pTP p2) R({y})}
try { /* ps1 = p1 */
  x = y [p1] rand();
} check (f(x, y))
recover {
  x = y [p2] rand();
}
{0.99 ≤ R({x})}
```

Figure 4.12: Example: recovering with an imperfect checker via redo

$r_{s2} \cdot \mathcal{R}(X_{s2})$. The probability that these two events occur *and* variables in X are calculated correctly in the third scenario is therefore $(1 - p_{s1}) \cdot p_{_{TP}} \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$.

Calculating $\mathcal{R}(X)$. We can now calculate the total probability that variables in X are calculated correctly as the sum of the probabilities of the three cases.

$$\begin{aligned}\mathcal{R}(X) \ &= p_{s1} \cdot p_{_{TN}} \cdot \mathcal{R}(X_{s1}) + p_{s1} \cdot p_{_{FP}} \cdot r_{s2} \cdot \mathcal{R}(X_{s2}) \\ &\quad + (1 - p_{s1}) \cdot p_{_{TP}} \cdot r_{s2} \cdot \mathcal{R}(X_{s2})\end{aligned} \tag{4.10}$$

Using the ordering preposition, we simplify this as follows:

$$\begin{aligned}\mathcal{R}(X) &\geq p_{s1} \cdot p_{_{TN}} \cdot \mathcal{R}(X_{s1} \cup X_{s2}) + p_{s1} \cdot p_{_{FP}} \cdot r_{s2} \cdot \mathcal{R}(X_{s1} \cup X_{s2}) \\ &\quad + (1 - p_{s1}) \cdot p_{_{TP}} \cdot r_{s2} \cdot \mathcal{R}(X_{s1} \cup X_{s2}) \\ \mathcal{R}(X) &\geq (p_{s1} p_{_{TN}} + p_{s1} p_{_{FP}} r_{s2} + (1 - p_{s1}) p_{_{TP}} r_{s2}) \cdot \mathcal{R}(X_{s1} \cup X_{s2})\end{aligned} \tag{4.11}$$

We replace $\mathcal{R}(X)$ in the postcondition with this probability to get the precondition:

$$c \leq r \cdot \Big(p_{s1} p_{_{TN}} + p_{s1} p_{_{FP}} r_{s2} + (1 - p_{s1})\, p_{_{TP}} r_{s2}\Big) \cdot \mathcal{R}(X_{s1} \cup X_{s2}) \tag{4.12}$$

4.5.2 Example: Imperfect Checkers

The example in Figure 4.12 shows the effect of an imperfect checker `f` on reliability precondition generation. We use the same naming convention as in Table 4.1. Aloe's analysis shows that while the overall reliability of the computation is sensitive to the presence of imperfect `check` functions, using checkers still significantly improves the reliability over the unchecked computation.

4.5.3 Classes of Imperfect Checkers

There are three main classes of imperfect checkers. The first class are those executed on unreliable hardware, just like the `try` and `recover` blocks. Consequently, the checker itself may execute incorrectly. The simplest example is where the check performs the same computation as in the `try` block and compares the results.

The second class of imperfect checkers are randomized computations called *property checkers* [182]. Property checkers identify all incorrect results ($p_{_{FN}} = 0$), but may fail to identify a correct result ($p_{_{FP}} > 0$).

The third class of imperfect checkers use machine learning to infer a function `f'` that approximates the perfect checker `f` using data about valid input/output correlations for the computation in the `try` block (e.g., outlier detection [74] or DNN checkers [135]). For such checkers, we can estimate the probabilities $p_{_{FP}}$ and $p_{_{FN}}$ from the checker's results on the training and testing data. However, these estimates are only valid if the inputs to the `try` block have a similar distribution as the training and testing data. If this is not the case, we can use AxProf (Chapter 2) to statistically check the reliability of the program with the out-of-distribution inputs to see if it is lower than the reliability calculated by Aloe. If so, this indicates that we must train a new checker for such inputs.

4.6 CORRECTNESS OF RELIABILITY PRECONDITION GENERATION FOR TRY-CHECK-RECOVER

Theorem 4.1. If 1) p_t is the minimum success probability of s_{try}, 2) $p_{_{TN}}$, $p_{_{FP}}$, and $p_{_{TP}}$ are the true negative, false positive, and true positive rates of the checker function f respectively, 3) s_{try} and s_{rec} are idempotent and perform the same computation, and 4) the relations $\psi \models \{c \leq r r_t \mathcal{R}(Y_t)\}\ s_{try}\ \{c \leq r\mathcal{R}(X)\}$ and $\psi \models \{c \leq r r_r \mathcal{R}(Y_r)\}\ s_{rec}\ \{c \leq r\mathcal{R}(X)\}$ hold true, then, $\psi \models \{c \leq r s_{tcr} \mathcal{R}(Y_t \cup Y_r)\}$ `try` $\{s_{try}\}$ `check` $\{f\}$ `recover` $\{s_{rec}\}$ $\{c \leq r\mathcal{R}(X)\}$, where $s_{tcr} = p_t p_{_{TN}} + p_t p_{_{FP}} r_r + (1 - p_t) p_{_{TP}} r_r$.

Proof. To prove Theorem 4.1, we first replace the precondition of s_{try} with $c \leq r p_t \mathcal{R}(Y_t \cup Y_r)$ and that of s_{rec} with $c \leq r r_r \mathcal{R}(Y_t \cup Y_r)$. As $p_t \leq r_t$ and $Y_t, Y_r \subseteq Y_t \cup Y_r$, this is a sound replacement (Proposition 2 of [44]).

The variables in X and $Y_t \cup Y_r$ fall into one of three categories:

1. Variables that are neither read nor written to by the `try` and `recover` blocks.

2. Variables that are read by the `try` and `recover` blocks.

3. Variables that are written to by the `try` and `recover` blocks.

The variables in Category 1 are transferred from the postcondition's joint reliability predicate to the precondition's joint reliability predicate unchanged, as per Rely's precondition generation rules. Aloe's idempotency requirement ensures that Categories 2 and 3 are mutually exclusive; otherwise, variables written to in the current execution would affect future executions. Any variables that are both read by and written to by the `try` and `recover` blocks must be internal temporary variables invisible to the rest of the program.

Let ϵ, φ be the environment and environment distribution before executing the try-check-recover block, and ϵ', φ' be the environment and environment distribution after executing the try-check-recover block. By definition, $[\![\mathcal{R}(X)]\!](\epsilon', \varphi') = \sum_{\epsilon_u \in \mathcal{E}(\{X\}, \epsilon')} \varphi'(\epsilon_u)$. We consider two ways in which we can reach an environment in which variables in X are calculated correctly ($\mathcal{E}(\{X\}, \epsilon')$):

1. We start from an initial environment in which variables in Y_t have been calculated correctly, execute the `try` block, which calculates the variables in X correctly, and then the check passes.

2. We start from an initial environment in which variables in Y_r have been calculated correctly, execute the `try` block, fail the check, and then execute the `recover` block, which calculates the variables in X correctly.

The total probability of reaching a state in $\mathcal{E}(\{X\}, \epsilon')$ is the sum of the probabilities of these two cases. For simplification, we can soundly replace Y_t, Y_r in the two cases with $Y_t \cup Y_r$. Then we assume we start from an environment in $\mathcal{E}(\{Y_t \cup Y_r\}, \epsilon)$. By definition, $[\![\mathcal{R}(Y_t \cup Y_r)]\!](\epsilon, \varphi) = \sum_{\epsilon_u \in \mathcal{E}(\{Y_t \cup Y_r\}, \epsilon)} \varphi(\epsilon_u)$.

Case 1. From the precondition / postcondition of the `try` block *in isolation*, we know that $\sum_{\epsilon_u \in \mathcal{E}(\{X\}, \epsilon')} \varphi'(\epsilon_u) \geq \sum_{\epsilon_u \in \mathcal{E}(\{Y_t \cup Y_r\}, \epsilon)} \varphi(\epsilon_u) \times p_t$. Within the try-check-recover block, after a correct execution of the `try` block, the check passes with probability p_{TN}. Therefore, the contribution of this case is $\sum_{\epsilon_u \in \mathcal{E}(\{Y_t \cup Y_r\}, \epsilon)} \varphi(\epsilon_u) \times p_t \times p_{TN}$.

Case 2. From the precondition / postcondition of the `recover` block *in isolation*, we know that $\sum_{\epsilon_u \in \mathcal{E}(\{X\}, \epsilon')} \varphi'(\epsilon_u) \geq \sum_{\epsilon_u \in \mathcal{E}(\{Y_t \cup Y_r\}, \epsilon)} \varphi(\epsilon_u) \times r_r$. Within the try-check-recover block, for this case, the check must fail. This happens in two ways: either the `try` block executes correctly and the check fails, or the `try` block causes an error and the check fails. The first sub-case happens with probability $p_t p_{FP}$ and the second sub-case with probability $(1-p_t)p_{TP}$. Therefore the contribution of this case is $\sum_{\epsilon_u \in \mathcal{E}(\{Y_t \cup Y_r\}, \epsilon)} \varphi(\epsilon_u) \times r_r \times (p_t p_{FP} + (1 - p_t)p_{TP})$. The idempotency constraint ensures that r_r is unaffected by the `try` block's probability of experiencing an error.

Table 4.2: Aloe benchmark details

		Lines of code			
Benchmark	**Source**	**Kernel**	**Total**	**Unrolled**	**Input**
PageRank	CRONO [183]	20	45	720K	10 iterations, 1000 node graph
Scale	Chisel [36]	57	79	561K	512×512 image (baboon.ppm)
BScholes	Chisel [36]	81	97	270K	4000 options from Parsec [184]
SSSP	CRONO [183]	22	31	800K	1000 node graph
BFS	CRONO [183]	14	30	720K	1000 node graph
SOR	Chisel [36]	24	37	1500K	10 iterations, 1000×1000 array
Motion	Rely [44]	14	32	150K	1000×1000 array
Sobel	AxBench [185]	34	45	160K	10 blocks, 1600 pixels each

Combining the two cases. Adding up the probabilities of the two cases of the try-check-recover block execution, we finally get

$$\begin{aligned}\textstyle\sum_{\epsilon_u \in \mathcal{E}(\{X\},\epsilon')} \varphi'(\epsilon_u) &\geq \textstyle\sum_{\epsilon_u \in \mathcal{E}(\{Y_t \cup Y_r\},\epsilon)} \varphi(\epsilon_u) \times p_t \times p_{_{TN}} \\ &\quad + \textstyle\sum_{\epsilon_u \in \mathcal{E}(\{Y_t \cup Y_r\},\epsilon)} \varphi(\epsilon_u) \times r_r \times (p_t p_{_{FP}} + (1-p_t)p_{_{TP}}) \\ \textstyle\sum_{\epsilon_u \in \mathcal{E}(\{X\},\epsilon')} \varphi'(\epsilon_u) &\geq \textstyle\sum_{\epsilon_u \in \mathcal{E}(\{Y_t \cup Y_r\},\epsilon)} \varphi(\epsilon_u)(p_t \times p_{_{TN}} + r_r \times (p_t p_{_{FP}} + (1-p_t)p_{_{TP}})) \quad (4.13)\end{aligned}$$

That is, $[\![\mathcal{R}(X)]\!](\epsilon', \varphi') \geq [\![\mathcal{R}(Y_t \cup Y_r)]\!](\epsilon, \varphi)(p_t \times p_{_{TN}} + r_r \times (p_t p_{_{FP}} + (1-p_t)p_{_{TP}}))$.

QED.

4.7 METHODOLOGY

Benchmarks. To evaluate Aloe, we implemented a set of benchmarks from several application domains. These benchmarks can tolerate some error in their output and have been previously explored in approximate computing literature. Table 4.2 presents benchmark statistics, including the benchmark suite from which we obtained the benchmark (Column 2), the size of the computation kernel, full benchmark, and the unrolled program (Columns 3-5), and the input size (Column 6). The line counts exclude setup and I/O code. We briefly describe each benchmark below:

- *PageRank:* Computes the PageRank [186] for nodes in a graph
- *Scale:* Creates a bigger version of an image
- *BScholes:* Computes the prices of a portfolio of stock options
- *SSSP:* Computes the Single Source Shortest Path in a graph

- *BFS:* Performs a Breadth First Search in a graph
- *SOR:* Uses Successive Over-Relaxation to model an iterative process
- *Motion:* A pixel block search algorithm from the x264 video encoder
- *Sobel:* The Sobel edge-detection filter

Unreliable operations. We used probabilistic choice statements to simulate two different unreliable architectures where arithmetic operations can fail and produce an incorrect output with probability 10^{-3} and 10^{-4} respectively. That is, the reliability of arithmetic operations in the two architectures is 0.999 and 0.9999 respectively. We executed each benchmark with a runtime library that would randomly replace the result of arithmetic operations with an incorrect value based on the error probability of the architecture.

Specifications and checkers. Each benchmark has at least one try-check-recover block. The `try` block executes on the architecture where arithmetic operators have reliability 0.999. The `recover` block executes the same code on the more reliable architecture where arithmetic operators have reliability 0.9999. For perfect checkers, we assumed a hardware technique with support for detecting errors [63, 75, 181]. For imperfect checkers, we used multiple false-positive and false-negative values in the range of those explored in Topaz [74].

Environment. We ran all experiments on a computer with an Intel Xeon 3.6GHz CPU with 32 GB RAM that was running Ubuntu 18.04. We used ANTLR for parsing and Python as the backend for Aloe. For empirical evaluation, we added instrumentation to track the number of errors and the end-to-end error magnitude.

4.8 EVALUATION

4.8.1 Static Reliability Analysis: Perfect Checkers

Table 4.3 shows the reliability postcondition we verified for each benchmark kernel in the presence of a perfect checker. Column 2 shows the verified reliability postcondition for the benchmark kernel and Column 3 indicates if Aloe was able to verify the bound (✓) or not (✗) along with the runtime. Column 4 shows if the same bound could be verified through a naive Rely analysis that treats the try-check-recover statement as an if-then-else statement as discussed in Section 4.1.2, along with the runtime. Table 4.4 shows the reliability postcondition we verified for each end-to-end benchmark in the same format.

Table 4.3: Verified kernel reliability postconditions

Benchmarks	**Kernel postcondition**	**Aloe**	**Rely**
PageRank	$0.9999 \leq \mathcal{R}(\text{newPagerank})$	✓(3ms)	✗(3ms)
Scale	$0.9999 \leq \mathcal{R}(\text{newPixel})$	✓(2ms)	✗(2ms)
BScholes	$0.9999 \leq \mathcal{R}(\text{optionPrice})$	✓(3ms)	✗(2ms)
SSSP	$0.9999 \leq \mathcal{R}(\text{dist})$	✓(3ms)	✗(3ms)
BFS	$0.9999 \leq \mathcal{R}(\text{visited})$	✓(3ms)	✗(4ms)
SOR	$0.9999 \leq \mathcal{R}(\text{result})$	✓(1ms)	✗(1ms)
Motion	$0.9999 \leq \mathcal{R}(\text{MinSSD})$	✓(45ms)	✗(45ms)
Sobel	$0.9999 \leq \mathcal{R}(\text{ImageOut})$	✓(1ms)	✗(1ms)

Table 4.4: Verified end-to-end reliability postconditions

Benchmarks	**E2E postcondition**	**Aloe**	**Rely**
PageRank	$0.99 \leq \mathcal{R}(\text{PageRank})$	✓(23.33s)	✗(19.73s)
Scale	$0.99 \leq \mathcal{R}(\text{ImageOut})$	✓(10.48s)	✗(8.79s)
BScholes	$0.99 \leq \mathcal{R}(\text{Prices})$	✓(6.51s)	✗(5.60s)
SSSP	$0.99 \leq \mathcal{R}(\text{Distances})$	✓(18.60s)	✗(18.25s)
BFS	$0.99 \leq \mathcal{R}(\text{Visited})$	✓(15.22s)	✗(15.14s)
SOR	$0.99 \leq \mathcal{R}(\text{ArrayOut})$	✓(21.02s)	✗(17.90s)
Motion	$0.99 \leq \mathcal{R}(\text{MinSSD})$	✓(4.42s)	✗(4.19s)
Sobel	$0.99 \leq \mathcal{R}(\text{ImageOut})$	✓(2.10s)	✗(1.80s)

The results show that Aloe can verify all reliability bounds, while the Rely approach fails to do so. When Rely treats the try-check-recover block as an if-then-else statement, it conservatively considers the lower reliability of the two branches to be the reliability of the entire statement. In contrast, when using Aloe's approach with a perfect checker, the reliability of the try-check-recover block is *greater* than the reliability of the `try` and `recover` blocks in isolation, since Aloe takes into account the fact that both `try` and `recover` blocks need to produce incorrect results together for an error to be introduced in the output.

For most kernels, the Aloe analysis requires about 3 milliseconds. The Motion kernel executes a computation for a large number of iterations, leading to increased analysis time after unrolling. For end-to-end reliability, Aloe's analysis requires 25 seconds or less.

4.8.2 Static Reliability Analysis: Imperfect Checkers

Table 4.5 shows the maximum reliability that Aloe can verify with imperfect checkers for the same reliability postconditions for kernels as in Table 4.3. Column 2 shows the highest reliability that Aloe can verify with a perfect checker. The next columns show the highest reliability that Aloe can verify with imperfect checkers with a particular true positive (p_{TP}) rate and true negative (p_{TN}) rate. For all kernels, the verifiable postcondition reliability decreases due to the possibility of false classifications by the checker. The reliability of the

Table 4.5: Maximum verifiable reliability postconditions for kernels with an imperfect checker

	Perfect	**Imperfect**			
Benchmarks	$\mathbf{p_{TP}}$ **: 1.00** $\mathbf{p_{TN}}$ **: 1.00**	$\mathbf{p_{TP}}$ **: 0.95** $\mathbf{p_{TN}}$ **: 0.95**	$\mathbf{p_{TP}}$ **: 1.00** $\mathbf{p_{TN}}$ **: 0.95**	$\mathbf{p_{TP}}$ **: 0.90** $\mathbf{p_{TN}}$ **: 0.90**	$\mathbf{p_{TP}}$ **: 0.80** $\mathbf{p_{TN}}$ **: 0.80**
PageRank	0.9999	0.9982	0.9968	0.9964	0.9375
Scale	0.9999	0.9993	0.9999	0.9987	0.9976
BScholes	0.9999	0.9992	0.9999	0.9985	0.9971
SSSP	0.9999	0.9995	0.9999	0.9991	0.9982
BFS	0.9999	0.9959	0.9999	0.9994	0.9839
SOR	0.9999	0.9997	0.9999	0.9994	0.9989
Motion	0.9999	0.9912	0.9918	0.8385	0.7031
Sobel	0.9999	0.9995	0.9999	0.9991	0.9982

Table 4.6: Empirical reliability results

		Perfect checker		**No recovery**
Benchmarks	**Error metric**	**Error**	**Fail%**	**Error**
PageRank	ℓ_2	1.54×10^{-5}	0.7%	4.04×10^{-4}
Scale	PSNR	56.023 dB	0.55%	38.280 dB
BScholes	ℓ_2	6.26×10^{-8}	0.15%	4.80×10^{-5}
SSSP	ℓ_2	2.82×10^{-3}	0.1%	4.27×10^{-3}
BFS	ℓ_2	0	0.15%	4.25×10^{-5}
SOR	ℓ_2	9.25×10^{-5}	0.75%	1.00×10^{-3}
Motion	SSD	0	0.15%	9.85×10^{-4}
Sobel	ℓ_2	2.63×10^{-5}	0.95%	2.69×10^{-5}

Motion kernel degrades faster as it executes a large number of iterations that compound the additional unreliability present as a result of the imperfect checker.

4.8.3 Empirical Results

We empirically confirmed the results of Aloe's static analysis of end-to-end programs for a perfect checker. We ran each program 2000 times and calculated the average output error due to unreliable operations and the fraction of runs where the output was different from a completely reliable execution. 2000 runs allows us to statistically verify that the empirically calculated failure rate is less than 0.01 (the verified reliability) using a one-sided binomial test ($\alpha = 0.05$, $\beta = 0.2$).

Table 4.6 gives the result of the empirical evaluation for each program. Column 2 shows the error metric we used. We used error metrics that were used in prior work for the same benchmarks. Columns 3 and 4 show the average output error and the failure rate (percentage of runs with incorrect results). Column 5 shows the average output error when the program does not use a recovery mechanism. The average output error is only calculated over runs where an error occurred.

The results show that the empirically calculated reliability is always within the bounds verified by Aloe for a perfect checker. Further, when these benchmarks *do* fail, the error in the final output is small, which shows the amenability of these benchmarks to approximations. For BFS and Motion, the program automatically corrected errors that occurred, without any additional intervention.

The results also show how the error increased, in some cases significantly, when the program did not have a recovery mechanism. Further, in all benchmarks, the failure rate also exceeded the 1% limit without a recovery mechanism.

4.9 RELATED WORK

Approximate program analyses. Many static analyses have been proposed in recent years for analyzing approximate or unreliable computations [36, 44, 45, 47, 48, 65, 187, 188, 189]. These existing analyses suffer from imprecision when analyzing computations with recovery mechanisms. Our analysis extends Rely by adding additional precondition generation rules for recovery mechanisms which generate less conservative preconditions compared to Rely. Dynamic analyses of reliability such as Diamont [46] and FastFlip (Chapter 5) can provide better reliability bounds for specific inputs to a program. While Aloe's verifiable reliability bounds are looser, they are input-agnostic.

FastFlip (Chapter 5) efficiently analyzes programs to determine the likelihood that an error in any particular instruction will lead to data corruption. FastFlip relies on the assumption that exactly one error occurs during the execution of the program. In comparison, Aloe allows for the possibility that multiple errors will occur during the program execution.

Error detection. Hardware error detectors [63, 66, 75, 190] typically consist of special circuits within the processors and memory units which can detect if an error has occurred. The accuracy of these detectors is limited by circuit size and energy requirements [75]. Another approach is to run the same computation on multiple processors at the same time [181, 191] and report an error if the computations disagree. However this requires a significant amount of redundant computation and physical space on the processor. Abdulrahman et al. [135] propose augmenting complex calculations with small neural networks that analyze the input and output to approximately determine if an error occurred. Topaz [74] uses outlier detection by constructing feature vectors from inputs and outputs. Our approach is agnostic of the nature of the error detection mechanism. We expose the checker interface (via the probabilities of false positives and false negatives) and show how to incorporate both perfect and imperfect checkers into the analysis.

Recovery mechanisms. Many systems deal with hardware failures using the checkpoint/restore method [192], which periodically saves the program state to reliable memory and restores the program state should an error occur. Languages such as Relax [63] and Topaz [74] expose recovery mechanisms to the developer. Relax allows retrying the unreliable computation (on the same unreliable hardware) as well as discarding incorrect calculations. Topaz instead opts for re-executing the computation on a perfectly reliable hardware or dropping tasks (following [51]). Several approaches in approximate computing provide implicit recovery from inaccuracy by dynamically adapting the approximation to the input properties [76, 77, 78, 79]. Containment Domains [193] provide programming constructs to define various software error detection and recovery mechanisms. Many of these approaches can be modeled as the types of recovery mechanisms supported by Aloe.

As-Is [194] executes supported approximate programs in a manner such that their accuracy increases over time. Given sufficient time, the output is exact, but the user can interrupt the program anytime to get an approximate output to accommodate time constraints. Unlike As-Is, Aloe aims for specific reliability targets (as opposed to best effort reliability in the time available) and similarly requires additional runtime to execute the recovery mechanism when necessary to meet those specific targets.

Fluid [195] relaxes data dependence relations in approximate programs to allow dependent tasks to execute before the program has calculated the exact value of their inputs. Fluid allows developers to check the results of such eagerly executed dependent tasks and re-execute them with exact inputs if the result is unsatisfactory. Aloe supports such an execution model by representing the eagerly executed task in the `try` block, with information about the reliability of the input, and using the developer defined check to determine whether to execute the `recover` block with a reliable input.

Chapter 5: FastFlip

As conventional technology scaling approaches its end, hardware is becoming increasingly susceptible to errors [4, 9]. Unlike crashes, timeouts, or detectable output corruptions, the presence of Silent Data Corruptions (SDCs) in program outputs caused by such hardware errors may not become apparent until long after the program has terminated. Researchers have proposed various hardware and software techniques to protect programs against SDCs. *Software protection techniques* such as instruction or task duplication [70, 71, 72, 196] are particularly attractive as they can be used on existing hardware. They typically use additional computational resources to detect SDCs and/or recover from them. To limit their time and energy overhead, it is necessary to selectively use such techniques on the parts of the program that are most vulnerable to SDCs.

Finding vulnerable instructions is the task of instruction-level error injection analyses. These analyses inject errors into different architectural components of a simulated CPU at various points in a program's execution, and record the effect of the error on the program output. For targeted SDC protection, the analysis must provide information on how errors at *each instruction* in the program affect the output (e.g., [53, 54]). Such detailed per-instruction analyses are time-consuming, requiring thousands of core-hours even for small programs and inputs. While sampling-based tools like [55, 56] are faster, they cannot provide the necessary vulnerability information for every instruction.

This high cost of per-instruction error injection analysis is concerning because modern programs are continuously evolving; developers change code to fix bugs, add features, or add optimizations manually or using compilers. The modified program sections react differently to errors, so they must be re-analyzed. However, current error injection analyses must be rerun on the whole program, not just the modified sections, before applying software protection techniques.

An intuitive solution is to leverage the compositional nature of programs by dividing them into multiple dynamic sections (such as function calls or executions of code blocks or nested loops), applying the error injection analysis on each section separately, and then combining the results. However, such a compositional error injection analysis must overcome multiple challenges: 1) SDCs in the output of one section must be propagated through downstream sections to determine the SDCs in the final output of the program, and 2) an error in one section can corrupt data that will be used only by downstream sections, thus causing unexpected side effects that do not affect the current section's output. To the best of our knowledge, none of the previously proposed analyses (e.g., [54, 55, 58, 59, 197, 198, 199, 200, 201, 202, 203, 204, 205]) can selectively analyze only the modified parts of the program.

Our work. We present FastFlip, a unique combination of empirical (error injection) and symbolic (SDC propagation) techniques, which makes it the first approach for compositional and incremental error injection analysis of evolving programs. Given a program and an input that FastFlip has not previously analyzed, FastFlip proceeds as follows: First, FastFlip analyzes the sections of the program for that input with a per-instruction error injection analysis to determine the outcome of each possible injected error. Second, FastFlip uses a local sensitivity analysis to determine how each section propagates and amplifies SDCs within its input, and then uses an SDC propagation analysis to determine how SDCs propagate across different sections to affect the final output. Third, FastFlip combines the results of the error injection analysis and the SDC propagation analysis to determine how injected errors in any section affect the final output. FastFlip uses this information to select a set of static instructions to protect, such that the dynamic cost of protecting the selected instructions is minimized, while ensuring that the degree of protection against SDC-causing errors is above a developer-specified threshold. FastFlip correctly accounts for side effects that only occur due to errors.

If developers modify a program after FastFlip has analyzed it, FastFlip can reuse large portions of its analysis results. In particular, FastFlip must only rerun the expensive error injection analysis on the modified program section, and any downstream sections whose input changes as a result of the modification. Consequently, FastFlip saves significant analysis time with each program modification.

We instantiate FastFlip using the Approxilyzer [54] per-instruction error injection analysis and the Chisel [36] SDC propagation analysis. Approxilyzer focuses on analyzing the effects of hardware errors that manifest within CPU architectural registers as bitflips. While analyses exist that also inject errors in other architectural components (e.g., [56]), they rely on sampling, and would be impractical if used to obtain per-instruction results. We analyze five benchmarks with FastFlip, each with three or more separately analyzed sections, and compare against a baseline Approxilyzer-only approach, which treats the entire program as a single section.

We evaluate the utility of protecting the static instructions selected by FastFlip and Approxilyzer using two metrics. The *value* of protecting a selection of instructions is the total probability that an SDC-causing error will be detected through said protection. The *cost* of a selection is roughly equal to the runtime overhead of protecting the selected instructions. For our evaluation, we use the value and cost model from [68], which assumes that value is proportional to the number of SDC-causing errors that can occur in the selected instructions, and that cost is proportional to the number of dynamic instances of the selected instructions that must be protected. FastFlip's selection provides a protection value within 0.3% (geomean) of the target protection value for a cost of protection within 0.7% (geomean) of the cost of protecting Approxilyzer's selection. FastFlip can also use more complex value and cost metrics to make its selection.

Next, we make two modifications to each benchmark: a *small*, simple modification and a *large* modification that uses a lookup table. FastFlip does not need to inject errors in the unmodified sections, providing a 3.2× speedup (geomean) over Approxilyzer. *Crucially, FastFlip provides this speedup with minimal additional loss in protection value or increase in cost with respect to Approxilyzer.* We also experiment with a modification that adds error detection mechanisms to a benchmark; FastFlip is able to efficiently verify that such a modification significantly decreases the likelihood of SDCs occurring due to errors in the protected sections.

5.1 BACKGROUND

5.1.1 Error Injection Analyses

Error injection analyses analyze the effect of injecting errors such as bitflips in the program execution. The analysis first enumerates error injection sites in the correct dynamic trace of the program execution, which is a sequential list of instructions executed by a program for a particular input. Depending on the analysis, these error sites can be bits in the source and destination registers in each instruction, bits in control registers, caches, etc. The analysis then injects errors at each site one at a time, and then executes the rest of the program (which may deviate in control flow from the correct execution), to record the effect of the error on the final output. Such analyses can operate at different levels of abstraction, including hardware, assembly, and IR (e.g., RAEs [197], Approxilyzer [54], and FlipIt [59] respectively.). An error can have five possible effects on the output of the program:

- The error is *masked*, i.e., the program output is unaffected.
- The error causes a program *crash*, i.e., the program terminates unexpectedly.
- The error greatly extends the program runtime (e.g., by creating a long loop), causing a *timeout*.
- The error changes the program output in a *detectable* manner (e.g., by producing a misformatted output or an output that is outside the expected range).

- The error changes the program output in an *undetectable* manner, i.e., it causes a *Silent Data Corruption* (SDC).

For the last case, the analysis typically reports the magnitude of the output SDC, calculated using an applicable SDC metric. The result of the analysis is a map from each error site to the outcome of an error at that site. The crash, timeout, and detectable error outcomes can be handled through detection and recovery mechanisms such as checkpoints. The SDC outcome is the most dangerous; the presence of an SDC may not be detected until long after the program has terminated. However, many applications are capable of tolerating small SDCs up to a threshold magnitude ε. Thus, similar to Approxilyzer, we further categorize SDCs above this threshold as *SDC-Bad* and smaller SDCs as *SDC-Good*.

Some error injection analyses use sampling to inject errors at random points in the program (e.g., [206, 207]). With a sufficient number of samples, these analyses provide statistically significant information on the relative frequencies of the above outcomes. Other error injection analyses aim to provide information on the outcome of error injections at *all* potential error sites of a particular class within a program's execution (e.g., [53, 54]). These non-sampling analyses are slower, but FastFlip can use their detailed results to find optimal static instructions to protect against SDCs.

5.1.2 SDC Propagation Analyses

SDC propagation analyses propagate SDCs present in a program's input, or SDCs that occur during program execution, to calculate their effect on the program's final output. These can be either *forward* analyses, or *backward* analyses (e.g., Diamont [46] and Chisel [36] respectively), which propagate SDC bounds in the respective directions through the program. An SDC bound $\Delta(o) \leq f(\Delta(i))$ states that the SDC in the output o of a section of code, calculated by an appropriate SDC metric Δ, is bounded by a function f of the SDC in the input i.

Sensitivity analysis. Sensitivity analysis [208] is a component of SDC propagation analyses that is used to determine how a section of code amplifies SDCs in its input. In particular, *local* sensitivity analysis focuses on determining the effect of perturbations around a single input value.

A local sensitivity analysis varies an input x_0 to a program section s by various amounts φ up to some maximum perturbation $\varphi_{\max}$. The analysis executes s to calculate the output perturbation and divides it by the input perturbation to calculate the SDC amplification factor K, which is the Lipschitz constant for s at x_0:

$$K = \max_{\varphi \leq \varphi_{\max}} \frac{|s(x_0 + \varphi) - s(x_0)|}{\varphi} \tag{5.1}$$

If the program is differentiable, it may be possible to analytically calculate K via static analysis (e.g., [209, 210]). In the general case, we can approximate K by sampling a set of φ values. Increasing the number of samples increases the quality of the approximation of K, though the rate of convergence also decreases, leading to diminishing returns.

5.1.3 Protecting Against SDCs in Software

Software techniques for protecting against SDCs have the advantage that they can be used on existing, non-specialized hardware. While systems can detect crashes and detectable errors (e.g., incorrectly formatted outputs) with relatively lightweight mechanisms, SDCs are harder to detect by nature. Detecting SDCs typically involves re-executing parts of the program and comparing the outcomes. Coarse grained solutions re-execute entire tasks [86, 211, 212], while fine grained solutions re-execute individual instructions or small blocks of instructions [70, 71, 72, 196].

Duplicating tasks or instructions does not necessarily double the program's resource usage. SWIFT [70] shows how instruction reordering by the compiler and the CPU can reduce the runtime cost of instruction duplication to an average of 41% of the program's original runtime. DRIFT [71] checks the results of multiple duplicated instructions at once to further increase instruction level parallelism and reduce the runtime overhead to 29% on average.

We can also choose to selectively protect only the instructions that are most vulnerable to SDCs. We can use error injection analyses that inject errors in all instructions [53, 54] to find such vulnerable instructions and selectively protect them. Each instruction has a value/cost tradeoff associated with it: the cost of protecting a static instruction using instruction duplication is roughly proportional to the number of dynamic instances of the instruction that will have to be duplicated, while the value is roughly proportional to the likelihood that an error that causes an SDC will occur in that instruction.

5.2 EXAMPLE

Lower-Upper decomposition (LU) is a key matrix operation that is used to solve systems of linear equations and to compute the matrix inverse or determinant. The blocked LU decomposition algorithm improves performance by dividing the matrix into blocks and processing a small fraction of blocks at a time. The algorithm consists of an outer loop whose loop body has four logical sections:

1. The algorithm decomposes a block B on the matrix diagonal
2. The algorithm updates blocks below B in the matrix
3. The algorithm updates blocks to the right of B in the matrix
4. The algorithm updates blocks below *and* to the right of B

Given the widespread use of LU decomposition, it is inevitable that hardware errors will occasionally occur in large computations that use this operation. While memory can be protected using ECC, data in registers is more vulnerable. If a bitflip error causes a detectable effect, then the software can be restarted or a previous checkpoint can be reloaded. However, if a bitflip causes an SDC, the corruption may not be detected until much later. Thus, we wish to protect the program against SDCs using techniques such as instruction duplication [70, 71, 72].

We can use an error injection analysis like Approxilyzer [54, 199] to analyze the effect of bitflips on the full program, and use the results to guide protection against SDCs. However, if developers modify one section of the program during development (e.g., to add optimizations), then we must rerun the analysis on the full modified program, which requires thousands of core-hours even for simple programs. We cannot naively re-analyze only the modified section with Approxilyzer, because the modified section most likely responds differently to injected errors, which leads to a different distribution of injection outcomes at the end of the full program.

To solve this issue, we present FastFlip, a compositional approach for error injection analysis of programs. FastFlip first separately analyzes each section of the program using Approxilyzer (to determine the effects of errors occurring within that section) and a local sensitivity analysis (to determine how the section propagates existing SDCs). FastFlip then combines the analysis results for each section using Chisel [36], an SDC propagation analysis, to calculate the end-to-end SDC characteristics of the program. Lastly, FastFlip uses these results to find optimal instructions to protect against SDCs.

We demonstrate the FastFlip approach on the blocked LU decomposition implementation from the Splash-3 benchmark suite [213] for a sample 16×16 input matrix with an 8×8 block size. Minotaur [180] has shown this input configuration to be sufficient for 100% program counter coverage. We assume that a single bitflip occurs during the program execution at an error site chosen uniformly at random from the program's correct dynamic trace within an architectural register.

Per-section analysis. FastFlip first runs Approxilyzer on each section s of the full program execution T. Approxilyzer determines the effect of injecting a bitflip into each bit in each register in each dynamic instruction in s on the output of s, and stores these outcomes for use in later steps. For SDC outcomes, FastFlip also stores the SDC magnitude, which is the maximum absolute error among all output elements of s.

Besides the SDC caused by bitflips, sections can also propagate and amplify SDCs caused by bitflips in previous sections. For example, FastFlip calculates that if the input to the first section in the second iteration of the outermost loop ($s_{2,1}$) has a pre-existing SDC of magnitude $\Delta(i_{s_{2,1}})$, and no bitflips occur within $s_{2,1}$, then the SDC in the output of $s_{2,1}$ will be at most $3.2\Delta(i_{s_{2,1}})$.

Under the single bitflip error model, a section s may either propagate an existing SDC from a previous section, or the bitflip may occur within s, but not both. Therefore, we can write the *total SDC specification* for the output of s as the sum of the propagated SDC and the SDC due to a bitflip within s. For example, using the results from Approxilyzer and the sensitivity analysis, FastFlip calculates the following upper bound on the effective SDC in the output of $s_{2,1}$ ($\Delta(o_{s_{2,1}})$) as a function of the SDC in the input of $s_{2,1}$ ($\Delta(i_{s_{2,1}})$) and the SDC potentially introduced by a bitflip in $s_{2,1}$ ($\varphi_{s_{2,1}}$):

$$\Delta(o_{s_{2,1}}) \leq 3.2\Delta(i_{s_{2,1}}) + \varphi_{s_{2,1}} \tag{5.2}$$

Calculating end-to-end SDC specifications. FastFlip provides the total SDC specifications for all sections to Chisel, along with a specification of how data flows between sections. Using this information, Chisel calculates the *end-to-end SDC propagation specification* for the full LU decomposition computation:

$$\Delta(o_{fin}) \leq 4174.8\varphi_{s_{1,1}} + 434.3\varphi_{s_{1,2}} + 28.8\varphi_{s_{1,3}} + 3.2\varphi_{s_{1,4}} + \varphi_{s_{2,1}} + \varphi_{s_{2,2}} + \varphi_{s_{2,3}} + \varphi_{s_{2,4}} \tag{5.3}$$

where $\varphi_{s_{x,y}}$ represents the SDC potentially introduced into the output of section y in iteration x by a bitflip in that section ($s_{x,y}$). The coefficient of each $\varphi_{s_{x,y}}$ represents the *total* amplification of an SDC introduced by a bitflip in $s_{x,y}$ by sections downstream of $s_{x,y}$. Under the single bitflip error model, only one of the $\varphi_{s_{x,y}}$ variables can be nonzero at a time. FastFlip uses this end-to-end SDC propagation specification to propagate SDCs caused by bitflips in each section up to the final output.

Selecting instructions to protect. FastFlip adapts the value and cost model from [68] to select a set of optimal instructions to protect against SDCs. FastFlip associates each

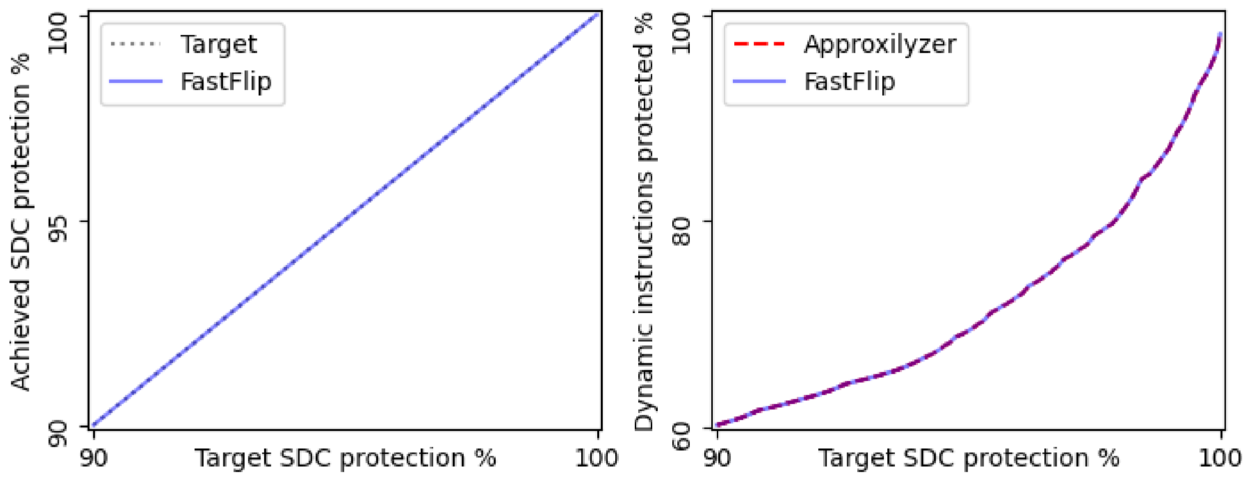


Figure 5.1: Protection value (left) and cost (right) comparison. Note the strong overlap within each plot.

static instruction *pc* in the program with a *value* of protecting it; this is the number of SDC-causing bitflips that can occur at *pc*. Similarly, FastFlip associates each static instruction *pc* with a *cost* of protection; this is the number of dynamic instances of *pc* in the program execution. We assume that the value and cost of protecting a set of instructions is the sum of the value and cost of protecting each instruction in the set. Given a target total SDC protection value, FastFlip aims to select a subset of instructions that meet this target while also minimizing the total protection cost. This is a 0–1 knapsack optimization problem, which FastFlip solves via the standard dynamic programming approach.

Comparison and target adjustment. We compare FastFlip's results to those of a baseline Approxilyzer-only approach. This baseline approach performs an error injection analysis of the whole program at once and uses the results to determine which instructions to protect. Using the results of the baseline analysis, we can calculate FastFlip's *achieved* value, which is the value of protecting FastFlip's selection of instructions according to the error injection outcomes of the baseline analysis. If FastFlip's achieved value is below the target value, FastFlip adjusts the target upwards, so that its selection of instructions to protect for this adjusted target will successfully achieve the original target value.

5.2.1 Results

Value. The left plot in Figure 5.1 shows the value of protecting FastFlip's selection of instructions against SDCs. The X-Axis shows the target value over the range $[90\%, 100\%]$.

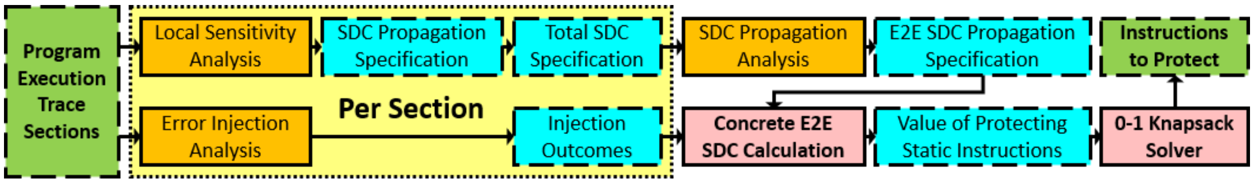
The Y-Axis shows the achieved value. The solid blue line shows FastFlip's achieved protection value (after target adjustment), which overlaps the dotted black line showing the target value. FastFlip successfully achieves (or slightly overshoots) the target protection value for the entire range of targets. Even without target adjustment, FastFlip undershoots the target by less than 0.1%.

Cost. The right plot in Figure 5.1 compares the cost of protecting FastFlip's selection of instructions against the cost of protecting the Approxilyzer baseline's selection. The red dashed line and solid blue line show the cost using Approxilyzer and FastFlip's results, respectively; the two lines visually overlap. The maximum excess of cost of FastFlip over Approxilyzer is below 0.1%, even after target adjustment.

Modifications. FastFlip enables *compositional analysis* by splitting the full program Approxilyzer analysis into multiple sections. We demonstrate the benefits of compositional analysis by performing both analyses on two modified versions of this program. The *small* modification uses a specialized version of section 4 of the program which reduces the number of bounds checks when the matrix size is a multiple of the block size (as is the case for our input). The *large* modification replaces the first section with a lookup table. Unlike the Approxilyzer-only approach, which must inject errors in the full execution of the modified program, FastFlip only needs to inject errors in the modified section of the program, saving considerable time. FastFlip also reuses the adjusted targets that it found for the original version of the program. FastFlip continues to achieve the original target values with these adjusted targets, and the excess of cost of FastFlip over Approxilyzer stays below 0.3%.

Analysis time. FastFlip requires 694 core-hours to analyze the original version of the program, compared to 602 core-hours for Approxilyzer. This slowdown is due to Approxilyzer's ability to prune injections across multiple sections of the computation by forming large equivalence classes. FastFlip cannot prune injections to a similar extent, as it analyzes each section independently. However, FastFlip saves a significant amount of time when subsequently analyzing the modified versions of the program. For analyzing the program with the *small* modification, FastFlip requires 80 core-hours as opposed to 625 core-hours for Approxilyzer. Similarly, for analyzing the program with the *large* modification, FastFlip requires 94 core-hours as opposed to 441 core-hours for Approxilyzer. This shows that FastFlip is useful when analyzing programs that evolve over time, because it saves analysis time with each modification.

FastFlip enables efficient target adjustment by simultaneously running the Approxilyzer



Program Execution Trace Sections
Local Sensitivity Analysis
SDC Propagation Specification
Total SDC Specification
Per Section
Error Injection Analysis
Injection Outcomes
SDC Propagation Analysis
E2E SDC Propagation Specification
Instructions to Protect
Concrete E2E SDC Calculation
Value of Protecting Static Instructions
0-1 Knapsack Solver

Figure 5.2: The FastFlip approach. Green boxes with dashed outlines and bold text are initial inputs and final results. Blue boxes with dashed outlines and normal text are intermediate results. Pink boxes with solid outlines and bold text are steps executed by FastFlip. Orange boxes with solid outlines and normal text are steps executed by the sub-analyses. The dotted box shows the portion of the approach that is applied to each section.

baseline analysis while performing its own analysis. For the original version of the program, the overhead of this simultaneous approach is less than 1% of the FastFlip-only analysis time. Since FastFlip reuses the adjusted targets for the modified versions, it does not need the simultaneous approach for those versions.

5.3 THE FASTFLIP APPROACH

Figure 5.2 visualizes the FastFlip approach. First, FastFlip performs two sub-analyses on each program section s (a function call or one execution of a code block or loop nest marked by the developer) in the full program execution T:

- FastFlip uses a supported error injection analysis[4] to determine the effect of each possible error in s and stores the outcome.
- FastFlip uses a local sensitivity analysis to obtain an SDC propagation specification for s, and converts it into a total SDC specification for s.

Second, FastFlip runs a supported SDC propagation analysis[4] over T, using each section's total SDC specification, to obtain the end-to-end SDC propagation specification for T. Third, FastFlip calculates concrete end-to-end SDC magnitudes to find the probability of an SDC-Bad outcome associated with each static instruction; this corresponds to the value of protecting said static instruction. Finally, FastFlip selects a set of instructions to protect with SDC detection mechanisms that minimizes the total cost of protection, while also ensuring that the total value of the protection is above a developer-defined threshold.

[4] We describe the characteristics of supported error injection and SDC propagation analyses in Section 5.3.10.

5.3.1 Preliminaries

As in previous works [53, 54], FastFlip assumes that 1) the program's input is SDC-free and 2) exactly one error occurs during the execution of the full program. This error can be a single or multi-bit corruption within *one* dynamic instruction.

Definitions. We use the following symbols:

- T: full program execution
- s: program section (typically a function call or one execution of a code block or loop nest); $s \in T$
- J: set of all error injection sites in T
- J_s: set of all error injection sites in s; $J_s \subseteq J$
- $O_s(j)$: effect of an error injection j on the outputs of s, as calculated by the error injection analysis
- $i_{s,0}, \ldots, i_{s,m}$ and $o_{s,0}, \ldots, o_{s,n}$: inputs and outputs of s
- $i_{T,0}, \ldots, i_{T,m}$ and $o_{T,0}, \ldots, o_{T,n}$: inputs and outputs of T
- $f_{s,k}, f_{T,\lambda}, f_{T,\lambda,s}$: SDC propagation specifications calculated by the local sensitivity analysis, the SDC propagation analysis, and FastFlip respectively
- $\varphi_{s,k}, \varphi_{*,*}, \varphi_{s,*}, \varphi_{\bar{s},*}$: symbolic variables (or sets thereof) for SDCs introduced into section outputs by errors
- $p(j)$: probability that the error occurs at $j \in J$
- $PC(j)$: maps $j \in J$ to the corresponding static instruction
- ε_λ: maximum acceptable SDC for output $o_{T,\lambda}$ of T
- $v(pc)$: value of protecting static instruction at pc
- $c(pc)$: cost of protecting static instruction at pc
- pc_{prot}: set of static instructions selected for protection

Analysis inputs. FastFlip accepts the full program T, its partition into sections s, a specification of how data flows between sections, the probabilities $p(j)$, SDC limits ε_λ, and protection cost $c(pc)$ as inputs. Developers can obtain the dataflow specification using standard compiler analysis passes. Expert developers can also input this data manually.

5.3.2 Error Injection Analysis of Program Sections

FastFlip runs an error injection analysis on each program section $s \in T$ to determine the effect of injected errors on the outputs of s, and stores the outcome. If an injection j causes a detectable outcome, such as a crash, timeout, or any clearly out-of-bounds or misformatted output, then the outcome $O_s(j) =$ *detected*. Otherwise, the outcome $O_s(j) = \{r_0, r_1, \ldots, r_n\}$, where r_k is the magnitude of SDC (as measured by an application-specific metric) caused by the injection j in output $o_{s,k}$ of s. If the injection is masked for an output $o_{s,k}$, then $r_k = 0$.

5.3.3 SDC Propagation Analysis of Program Sections

FastFlip performs a local sensitivity analysis on each program section $s \in T$ to calculate how it amplifies SDCs present within its input. The local sensitivity analysis produces an *SDC propagation specification* for s of the general form:

$$\bigwedge_{k=0}^{n} \Delta(o_{s,k}) \leq f_{s,k}(\Delta(i_{s,0}), \ldots, \Delta(i_{s,m})) \tag{5.4}$$

that is, for each output $o_{s,k}$ of s, the specification provides the SDC bound $\Delta(o_{s,k})$ calculated as a function $f_{s,k}$ of the SDC bounds of the inputs of s.

To convert this SDC propagation specification to a total SDC specification, FastFlip adds symbolic variables $\varphi_{s,k}$ to represent the magnitude of SDC introduced into the output $o_{s,k}$ during the execution of s as a result of an error. Under the single error model, if the input to s already contains an SDC, then the error occurred in a previous program section, hence s cannot introduce additional SDC. Thus, for each output of s, we can simply write the total SDC as the sum of the SDC due to an error in s and the SDC propagated by s from its input to its output. This is the *total SDC specification* for s:

$$\bigwedge_{k} \Delta(o_{s,k}) \leq f_{s,k}(\Delta(i_{s,0}), \ldots, \Delta(i_{s,m})) + \varphi_{s,k} \tag{5.5}$$

5.3.4 End-to-End SDC Propagation Analysis

FastFlip runs an SDC propagation analysis on the full program T. FastFlip provides the analysis with the total SDC specifications from Equation 5.5 for each $s \in T$. The analysis also uses the developer-provided dataflow specification indicating how outputs of one section flow into the inputs of a subsequent section. The SDC propagation analysis uses this information to calculate the relationship between errors that occur anywhere in T to the SDC in the final outputs of T. This is the *end-to-end SDC propagation specification* for T and has the form:

$$\bigwedge_{\lambda=0}^{n} \Delta(o_{T,\lambda}) \leq f_{T,\lambda}(\Delta(i_{T,0}), \ldots, \Delta(i_{T,m}), \varphi_{*,*}) \tag{5.6}$$

where $\varphi_{*,*}$ is the list of the symbolic SDC variables across all sections. Because we assume, as in previous work [54], that the initial inputs to the program are free of SDCs, we can simplify $f_{T,\lambda}$ as follows:

$$\bigwedge_{\lambda} \Delta(o_{T,\lambda}) \leq f_{T,\lambda}(\varphi_{*,*}) \tag{5.7}$$

Next, we create specialized versions of $f_{T,\lambda}$ by noting that, under the single error model, the symbolic SDC variables for only one section can be nonzero at a time:

$$f_{T,\lambda,s}(\varphi_{s,*}) = f_{T,\lambda}(\varphi_{s,*}, \varphi_{\bar{s},*} = \mathbf{0}) \tag{5.8}$$

where $\varphi_{s,*}$ is the list of the symbolic SDC variables for section s and $\varphi_{\bar{s},*}$ is the list of the symbolic SDC variables for all other sections. Finally, we rewrite the end-to-end SDC propagation specification as:

$$j \in J_s \Rightarrow \bigwedge_{\lambda} \Delta(o_{T,\lambda}) \leq f_{T,\lambda,s}(\varphi_{s,*}) \tag{5.9}$$

Equation 5.9 states that, if FastFlip injects an error in section s, then the upper bound on the SDC in output $o_{T,\lambda}$ of T is $f_{T,\lambda,s}(\varphi_{s,*})$, a function of the magnitude of SDC in the outputs of s.

5.3.5 Calculating the Value of Protecting Static Instructions

FastFlip uses the injection outcomes (Section 5.3.2) and Equation 5.9 to answer the following question: *For each static instruction identified by its program counter pc in the full execu-*

tion T, what is the total probability[5] *of error injections that result in SDC-Bad* ($|SDC| > \varepsilon_\lambda$) *for any output $o_{T,\lambda}$ of T?* The value of protecting pc with SDC detection mechanisms is proportional to this total probability.

Algorithm 5.1 Finding the value of protecting a static instruction

```
   Input $T$, $J_s$, $PC(j)$, $O_s(j)$, $\varepsilon_\lambda$, $p(j)$: defined in Section 5.3.1; $f_{T,\lambda,s}$: SDC propagation
   specification from outputs of $s$ to output $o_{T,\lambda}$ of $T$
   Returns $\forall pc.\ v(pc)$: value of protecting the static instruction $pc$
1: $v \leftarrow \{\ \forall pc.\ pc \mapsto 0\ \}$
2: for $s$ in $T$ do
3:     for $j$ in $J_s$ do
4:         $pc \leftarrow PC(j)$
5:         if $O_s(j) \neq detected$ then
6:             if $\exists \lambda.\ f_{T,\lambda,s}(O_s(j)) > \varepsilon_\lambda$ then
7:                 $v(pc) \leftarrow v(pc) + p(j)$
8: $v_{total} = \Sigma_{pc} v(pc)$
9: $\forall pc.\ v(pc) \leftarrow v(pc)/v_{total}$
```

Algorithm 5.1 shows how FastFlip calculates the value $v(pc)$ of protecting a static instruction pc. For each error injection in each section, FastFlip checks if the error results in a detectable outcome. If not, FastFlip uses Equation 5.9 to calculate upper bounds on the SDCs in the outputs of T as a function of the SDCs in the outputs of s. If any SDC is SDC-Bad, FastFlip adds the probability of that error to the value of protecting pc. Lastly, FastFlip rescales the values so that the total value of protecting all static instructions is equal to 1.

5.3.6 Finding an Optimal Set of Instructions to Protect

The value $v(pc)$ of protecting the static instruction pc calculated using Algorithm 5.1 and the corresponding protection cost $c(pc)$ together comprise a value and cost model similar to the one from [68]. FastFlip uses $v(pc)$ and $c(pc)$ as inputs to a 0–1 knapsack optimization problem. FastFlip assumes that the value and cost of protecting a set of instructions is the sum of the value and cost of protecting each instruction in the set. Given a developer-defined target total protection value v_{trgt}, FastFlip solves the knapsack problem via the standard dynamic programming approach to select a set of static instructions pc_{prot} to protect that

[5]This is the probability that the error occurs within pc *and* the outcome is SDC-Bad, as opposed to the conditional probability that the outcome is SDC-Bad when the error occurs in pc.

minimizes the total protection cost:

$$\textit{Minimize} \sum_{pc \in pc_{prot}} c(pc) \quad \textit{such that} \sum_{pc \in pc_{prot}} v(pc) \geq v_{trgt} \tag{5.10}$$

FastFlip then efficiently selects the optimal pc_{prot} for a range of v_{trgt} values. This corresponds to solving the value / cost multi-objective optimization problem using the ϵ-constraint method [214] (i.e., turning one of the objectives into a constraint as we do above) to obtain Pareto-optimal choices for pc_{prot}.

5.3.7 Precision of FastFlip

We observed that there are four major factors that affect the precision of FastFlip:

Inter-section masking. Inter-section masking occurs when an SDC in one section is masked by a downstream section. FastFlip must conservatively assume that SDCs introduced in any section result in SDCs in the final outputs. The frequency of inter-section masking is highly dependent on the application.

Imprecision of component analyses. As FastFlip depends on the results of the error injection analysis and the SDC propagation analysis, any imprecision in these analyses can lead to imprecision in FastFlip. For example, in our evaluation, FastFlip is affected by:

- The error injection analysis's injection pruning heuristics, which make per-instruction error injection practical at the cost of introducing some inaccuracy in the injection outcomes [54, Figure 5]
- The SDC propagation analysis's conservative SDC propagation, which causes it to overestimate the magnitude of SDC at the end of the program

Side effects. FastFlip requires each analyzed program section to be free of side effects. If a section modifies a variable that is used by a downstream section, then that variable must be considered as an output of that section. Even after all such variables are included in the output, the section may still cause additional side effects as a result of errors. Below, we describe two major categories of observed side effects that occur exclusively due to errors, along with the strategies employed by FastFlip to account for them.

First, the error may cause the section to write to a memory location outside the memory region where the section's output is stored (e.g., due to incorrect array index calculation or evaluation of a loop exit condition). As a result, it is possible for the outputs of the section to be correct while data in memory locations adjacent to the outputs is corrupted. This data may be used by downstream sections, leading to a side effect. FastFlip mitigates this effect by considering variables stored in memory locations adjacent to the section outputs to be an output of that section.

Second, the error may corrupt variables that will no longer be used by the current section, but are used in downstream sections (e.g., due to the corruption of a register being popped from the stack at the end of a section). FastFlip mitigates this effect by including variables that will be used by future sections in the output of the current section, even if the current section only reads said variables. FastFlip uses the dataflow specification provided by the developer to find such variables.

While these mitigation strategies eliminate a large majority of side effects introduced as a result of errors, some such side effects can still occur. As a result of these remaining side effects, the outcomes of some of the injections recorded by FastFlip may be incorrect.

Untested error sites. A small number of error sites in the full program may not be included in any program section. For example, if sections are executed multiple times within an outer loop, then the instructions which increment the outer loop counter and restart the outer loop body may be excluded from all program sections. FastFlip conservatively assumes that, if an error occurs at such an excluded error site, then it will *always* produce an SDC-Bad outcome. More rigorously, FastFlip creates an imaginary section $s_\perp$ containing all such untested error sites j and assumes that $\forall j \in J_{s_\perp},\ O_s(j) = \{\infty, \ldots, \infty\}$. This reduces the precision of FastFlip, as the actual number of SDC-Bad outcomes for these untested error sites is often lower.

5.3.8 Adapting FastFlip to Compensate for Loss of Precision

If FastFlip loses precision as a result of the factors described in Section 5.3.7, it can lead to a loss of *utility.* That is, a loss of precision can cause FastFlip to protect against a smaller number of errors that cause SDC-Bad outcomes than expected. Similarly, it can also increase the cost of protecting FastFlip's selection of static instructions beyond the actual minimum cost of protection. FastFlip adaptively adjusts the target value v_{trgt} used in Section 5.3.6 in order to compensate for this loss of utility.

Measuring utility. FastFlip must first measure its loss of utility. FastFlip compares its utility to the utility obtained via a baseline monolithic error injection analysis. The baseline analysis injects errors in the whole program at once and directly determines the outcome of these errors at the end of the program. It then uses these results to selectively protect vulnerable static instructions. FastFlip uses two primary metrics for measuring utility:

First, FastFlip treats the outcome labels of the monolithic error injection analysis as the ground truth and calculates the value of protecting its selection against SDC-Bad outcomes according to these alternate outcome labels. FastFlip refers to the protection value of its selection calculated in this manner as the *achieved value* v_{achv}. FastFlip then compares v_{achv} to the target protection value v_{trgt}. FastFlip calculates the loss of value as $v_{loss} = v_{trgt} - v_{achv}$. Value loss measures the degree to which FastFlip undershoots the target value of protection against SDC-Bad outcomes; a lower value loss is better.

Second, FastFlip calculates the excess cost of FastFlip's selection over the monolithic error injection analysis's selection. Specifically, if the costs associated with protecting the two selections of instructions against SDCs are c_{FF} (for FastFlip) and c_{Mono} (for the monolithic analysis) respectively, the excess cost is $c_{excess} = c_{FF} - c_{Mono}$. Excess cost measures the degree of inefficiency of FastFlip's selection for protecting against SDC-Bad outcomes as compared to the more efficient selection made by the monolithic analysis; a lower excess cost is better.

When analyzing a program, FastFlip can simultaneously run the monolithic error injection analysis for minimal additional analysis time[6]. To do so, FastFlip simultaneously checks the effect of each error in each section both on the outputs of that section, as well as the final outputs. Using these two sets of outcome labels, FastFlip can calculate v_{loss} and c_{excess}.

Adjusting the target value. FastFlip replaces the original target v_{trgt} with an adjusted target v'_{trgt}. Let the achieved value for this adjusted target be v'_{achv}. FastFlip minimizes v'_{trgt} such that $v'_{achv} \geq v_{trgt}$. If $v'_{trgt} > v_{trgt}$, then the cost of protecting FastFlip's selection increases, with larger adjustments leading to larger increases. It is also possible that $v'_{trgt} < v_{trgt}$, in which case the cost decreases instead.

5.3.9 Composability and Incremental Analysis

When developers modify a program section, FastFlip must rerun the error injection and local sensitivity analysis on the modified program section and any downstream sections whose input changes as a result of the modification. FastFlip can reuse the results of these sub-analyses for all other sections. Lastly, FastFlip must recalculate the end-to-end SDC propagation specifications using the SDC propagation analysis. Since running the error injection analysis is the major contributor to FastFlip's runtime, this approach leads to

[6]For our evaluation, we run the analyses separately for proper comparison of their analysis time.

significant speedups as compared to rerunning the error injection analysis on the full modified program, even if FastFlip must re-analyze multiple sections due to modifications that change the inputs of downstream sections.

Algorithm 5.2 Using adjusted target values when programs are modified

```
    Input $P_{adj}$: target adjustment interval; $m_{adj}$: number of modifications since last target
    adjustment; $v_{trgt}$: original target value; $v'_{trgt}$: adjusted target value
    Modifies $m_{adj}$; $v'_{trgt}$
    Returns $pc_{prot}$: selection of instructions to protect
1: if $m_{adj} \geq P_{adj}$ then
2:     $m_{adj} \leftarrow 0$
3:     $\mathit{Outcomes}_{FF}, \mathit{Outcomes}_{Mono} \leftarrow \textsc{FastFlipAndMonolithic}(\mathit{Program}, \mathit{Input})$
4:     $v'_{trgt} \leftarrow \textsc{AdjustTarget}(v_{trgt}, \mathit{Outcomes}_{FF}, \mathit{Outcomes}_{Mono})$
5: else
6:     $m_{adj} \leftarrow m_{adj} + 1$
7:     $\mathit{Outcomes}_{FF} \leftarrow \textsc{FastFlipModifiedOnly}(\mathit{Program}, \mathit{Input})$
8: $pc_{prot} \leftarrow \textsc{Knapsack}(v'_{trgt}, \mathit{Outcomes}_{FF})$
```

Algorithm 5.2 shows how FastFlip uses target value adjustment to compensate for loss of utility (Section 5.3.8) when the program is modified. FastFlip maintains a count of the number of modifications that have occurred since the most recent target adjustment (m_{adj}). If m_{adj} is below a threshold chosen by the developer (P_{adj}), FastFlip executes only its own time saving compositional analysis and uses the existing adjusted target v'_{trgt} to choose a set of static instructions to protect. That is, it is not necessary to always run the monolithic analysis as described in Section 5.3.8 for modified programs. As developers make modifications to the program, v'_{trgt} may no longer provide the expected compensation for loss of utility. For this reason, once $m_{adj} \geq P_{adj}$, FastFlip re-adjusts the target value by performing a fresh analysis of the whole program while simultaneously running the monolithic analysis. Developers can choose P_{adj} to trade off between utility loss and analysis time.

5.3.10 Characteristics of Usable Sub-Analyses

Error injection analyses. The error injection analysis must separately report the outcome for errors in each instruction in the program that the developer may wish to protect (e.g., [53, 54]). Analyses that use sampling and only report overall outcome statistics for the program (e.g., [55, 56]) are incompatible with FastFlip, as they do not provide instruction-specific vulnerability information. The analysis may inject single or multi-bit errors into one dynamic instruction per simulation.

SDC propagation analyses. The SDC propagation analysis must support the same application-specific SDC magnitude metric that the error injection analysis and the sensitivity analysis use to report the SDC magnitude. The analysis must also support the propagation of SDCs whose magnitude is represented by a symbolic variable, to enable the calculation of Equation 5.9. Examples of supported analyses are Chisel [36] and DeepJ [210].

5.4 METHODOLOGY

5.4.1 Choice of Sub-Analyses

Approxilyzer. Approxilyzer [54] is an error injection analysis that focuses on single bitflip errors that occur in CPU registers within in each dynamic instruction in a program execution. It uses various heuristics to form equivalence classes of bitflips that will most likely cause similar outcomes. Approxilyzer injects a bitflip into the program execution for only one pilot from each equivalence class. It then continues the now tainted program execution (with possibly incorrect control flow), and records the outcome of the bitflip. Finally, Approxilyzer applies the outcome of this pilot bitflip to all members of the equivalence class. We specifically use gem5-Approxilyzer [199].

Chisel. Chisel [36] is an SDC propagation analysis that calculates the end-to-end SDC propagation specification functions $f_{T,\lambda,s}$ as affine functions of the symbolic SDC variables[7] $\varphi_{s,*}$ (Equation 5.9). Chisel generates conservative specifications because it assumes that 1) each program section always amplifies input SDCs by the maximum amplification factor for that section for any input, and 2) when input SDCs can propagate through multiple control flow paths, they are always affected by the maximum amplification factor among all these paths, regardless of the actual path that the program execution takes.

5.4.2 Error Model

As we compare FastFlip's results to those of an Approxilyzer-only approach, we use the same error model as Approxilyzer (described below) to ensure a fair comparison. We inject one single-bit transient error per simulation in an architectural general purpose or SSE2 register. We target both source and destination registers in dynamic instructions within the region of interest (a subset of the correct dynamic trace of the program execution for a

[7]We modified Chisel to add support for such symbolic SDC variables.

Table 5.1: List of benchmarks for FastFlip

Benchmark	**Input size**	**Sections**	**$\|J\|$**
BScholes	2 options	4 ($\times 2$)	36.7K
Campipe	32×32	5 ($\times 1$)	72.7M
FFT	256×2	5 ($\times 1$)	9.23M
LU	16×16	4 ($\times 2$)	1.75M
SHA2	32 bytes	3 ($\times 1$)	403K

particular input), and skip instructions without register operands. We do not inject errors in special purpose, status, and control registers (e.g., %rsp, %rbp, and %rflags) as we assume that they always need protection which can be provided by hardware. Similarly, we assume that caches are protected via hardware techniques like ECC. As in previous works (e.g., [55, 215]) we assume that the probability $p(j)$ that the error will occur at any error site j follows a uniform distribution.

5.4.3 SDC Detection Model

SDC detection mechanism. We assume that a compiler pass (e.g., [70, 71]) duplicates the instructions that FastFlip selects for protection and then follows the original and duplicate instructions with a check that ensures that their results match. The duplicated code execution and increased register pressure leads to runtime overhead. However, by allowing the compiler and CPU scheduler to rearrange instructions and by coalescing multiple checks together, the overhead for extensive instruction duplication across the program can be reduced to 29% on average [71]. The overhead for selective duplication of individual instructions is even lower.

Value and cost of SDC detection. We adapt the value and cost model from [68], described below. Since the error model in Section 5.4.2 assumes that errors are uniformly distributed ($p(j)$ is uniform), the value of protecting a static instruction pc is proportional to the number of distinct errors possible in pc that produce an SDC-Bad outcome. The cost $c(pc)$ of protecting pc is proportional to the number of dynamic instances of pc in the program execution trace. Developers can use alternate value models by changing the error probability distribution $p(j)$ and alternate cost models by changing $c(pc)$.

5.4.4 Benchmarks

Table 5.1 shows our benchmarks for FastFlip. Column 2 shows the input size, Column 3 shows the number of static sections (and the number of dynamic instances of those sections), and Column 4 shows the total number of error sites ($|J|$). We briefly describe the benchmarks and their origin below:

- *BScholes*: Black-Scholes stock option analysis benchmark from the PARSEC suite [184]
- *Campipe*: The raw image processing pipeline for the Nikon-D7000 camera, adapted from [216]
- *FFT*: Fast Fourier Transform benchmark from the Splash-3 suite [213]
- *LU*: Blocked matrix decomposition benchmark from the Splash-3 suite [213]
- *SHA2*: The SHA-256 hash function, adapted from [217]

For FFT and LU, the input size is the same as the minimized input size found by Minotaur [180], a technique for reducing injection analysis time by minimizing inputs without sacrificing program counter coverage. For BScholes, we manually reduced the 21 option minimized input found by Minotaur down to 2 options while ensuring that the program counter coverage remained the same. For Campipe, we use a reference 32×32 raw image input provided along with the implementation. For SHA2, we use a common cryptographic key size (256 bits).

5.4.5 Baseline, Comparison, and Experimental Setup

Software and hardware. FastFlip uses gem5-Approxilyzer version 22.1 [199] simulating an x86-64 CPU as the architecture simulator. We performed our experiments on AMD Epyc processors with 94 error injection experiment threads.

Region of interest. We focus on the computational portion of each benchmark and do not analyze I/O, initial setup, or final cleanup code.

SDC magnitude metric. We use the maximum element-wise absolute difference as the SDC magnitude metric for all benchmarks. Specifically, if $o_k[\ell]$ represents the ℓ^{th} element of an output o_k and the modified output due to an injection is $\hat{o}_k$, then the SDC metric is $\max_\ell |o_k[\ell] - \hat{o}_k[\ell]|$.

SDC-Bad threshold. We first analyze all benchmarks assuming that any SDC is SDC-Bad, no matter how small ($\forall\lambda.\ \varepsilon_\lambda = 0$). Next, we relax this requirement by considering SDC magnitudes up to 0.01 to be tolerable, i.e., SDC-Good ($\forall\lambda.\ \varepsilon_\lambda = 0.01$) for all benchmarks except SHA2 (whose applications require the output to be fully precise).

Sensitivity analysis parameters. As we consider the maximum tolerable SDC magnitude ε_λ to be 0.01, we use this as the maximum perturbation during sensitivity analysis. To estimate the Lipschitz constant K, we perform 10^6 random perturbations up to ε_λ. For array inputs, we randomly perturb one, multiple, or all elements.

Comparison metrics. We compare the performance and utility of FastFlip to a baseline monolithic Approxilyzer-only approach. The baseline approach treats the entire program as a single section. For performance, we compare the analysis times of FastFlip and Approxilyzer.

For comparing utility, we compare the selections of instructions to protect made by the two approaches using the value loss and excess cost metrics described in Section 5.3.8. FastFlip always uses target adjustment in our evaluation, except in Section 5.5.4, where we investigate the effects of target adjustment. We compare utility for four target values: $v_{trgt} \in \{0.90, 0.95, 0.99, 1.00\}$, which are target values that correspond to protecting against 90%, 95%, 99%, and 100% of errors that cause SDC-Bad outcomes, respectively.

Error range. While Approxilyzer's use of equivalence classes as described in Section 5.4.1 speeds up analysis, the pilot is not a perfect predictor of the outcomes for the pruned injections (i.e., the rest of the equivalence class). Figure 5 in Approxilyzer [54] shows that, on average, 4% of pruned injections have an outcome that is significantly different from the pilot. Therefore, Approxilyzer's results cannot be considered to be the absolute ground truth for comparison.

To account for the potential discrepancy between the ground truth and Approxilyzer, we calculate an error range around the value of SDC protection according to Approxilyzer. Using the detailed results from Approxilyzer or FastFlip, we can determine, for each error site, 1) whether the outcome is SDC-Bad, or a different outcome, and 2) whether an injection was actually performed at that error site, or the error site was pruned (the outcome for an error at that site was inferred from the outcome for another error site in the same equivalence class). Additionally, each error site can either be protected or unprotected, depending on whether the analysis selected the corresponding static instruction for protection.

Based on these classifying factors, we can divide all error sites into eight categories, and use a variable to represent the *number* of error sites in each of these categories, as shown in

Table 5.2: Variables representing the number of error sites in various categories for error range calculation

	Injected		**Pruned**	
Outcome:	**SDC-Bad**	**Other**	**SDC-Bad**	**Other**
Protected	A	B	C	D
Unprotected	E	F	G	H

Table 5.2. Lastly, let R be the rate of outcome misprediction for pruned error sites. Using the above information, we can calculate the lower and upper bounds on the actual value of protecting the selected instructions:

$$\begin{aligned} v_{\min} &= \frac{A + (1-R)C}{A + (1-R)C + E + G + RH} \\ v_{\max} &= \frac{A + C + RD}{A + C + RD + E + (1-R)G} \end{aligned} \tag{5.11}$$

Then, the value error range is the interval $[v_{\min}, v_{\max}]$. Note that using the above notation, the value calculated in Section 5.3.6 is equal to:

$$v_{calc} = \frac{A + C}{A + C + E + G} \tag{5.12}$$

which always lies within $[v_{\min}, v_{\max}]$.

For FFT, LU, and BScholes, we use the benchmark-specific pilot prediction inaccuracy from Figure 5 in Approxilyzer [54] (3%, 4%, and 10% respectively). For Campipe and SHA2, we consider the average inaccuracy from the same figure (4%).

We calculate FastFlip's error range in this manner around its results for achieved protection value. If, for this error range, $v_{\max} \geq v_{trgt}$, then we consider FastFlip's result to be acceptable, even if v_{achv} is less than v_{trgt}.

Analysis time. For FastFlip, the total time required is the sum of the time required for 1) analyzing each program section with Approxilyzer and the sensitivity analysis, 2) Chisel error propagation analysis, 3) calculating the value of protecting static instructions, and 4) solving the knapsack problem. For modified benchmarks, we do not include the time required to analyze the unmodified sections in the first category. For Approxilyzer, the total time required is the sum of the time required for 1) analyzing the full program and 2) solving the knapsack problem.

Timeouts. FastFlip assumes that if the error causes the runtime of a program section to exceed $5\times$ the nominal runtime, then the execution times out, which is a detected outcome. We use the same timeout rule for Approxilyzer.

5.4.6 Program Modifications

To test the advantages offered by FastFlip's compositional analysis, we modify one or more sections within each benchmark. Then, we re-analyze the full modified program with Approxilyzer, and compare this to FastFlip's re-analysis of only the modified sections followed by SDC propagation.

For each benchmark, we experiment with two types of semantics-preserving modifications. *Small* modifications represent simple modifications that developers or compilers may make while optimizing and maintaining the program. Such modifications of up to 15 lines of code form a majority of open-source commits [218]. For the *large* modifications, we replace a program section with a lookup hashtable. The hashtable stores key-value pairs that map inputs of that section to corresponding outputs. If the modified section finds the current input in this table, it returns the corresponding output. Otherwise, it executes the original section code. Lastly, for the BScholes benchmark, we also experiment with a modification that uses coarse-grained code duplication to detect the presence of SDCs.

Details of *small* modifications. For Campipe and FFT, we store an expression used in multiple locations within the section into a variable to improve code readability. For LU, we introduce a specialized version of a section that reduces the number of necessary bounds checks when it detects that the matrix size is a multiple of the block size (as is the case for our input). For BScholes, we eliminate a redundant floating point operation that occurs in some cases in the cumulative normal distribution function. This change very slightly changes the semantics of the section due to floating point imprecision. For SHA2, we similarly eliminate a redundant shift operation (without changing semantics or making the runtime input-dependent).

5.5 EVALUATION

5.5.1 Similarity of Injection Outcomes of FastFlip and Approxilyzer

Table 5.3 shows how injection outcome statistics compare between FastFlip and the baseline Approxilyzer approach for the benchmarks without modifications. Column 2 shows

Table 5.3: Injection outcome counts for unmodified benchmarks

		Outcome counts		
Benchmark	**Approach**	**Detected**	**SDC**	**Masked**
BScholes	Approxilyzer	6.14K	19.1K	11.5K
	FastFlip	5.87K	19.6K	11.3K
	diff	-274	+503	-229
Campipe	Approxilyzer	29.0M	11.1M	32.6M
	FastFlip	29.8M	18.2M	24.8M
	diff	+749K	+7.09M	-7.84M
FFT	Approxilyzer	3.53M	4.22M	1.48M
	FastFlip	3.53M	4.40M	1.31M
	diff	-7.61K	+178K	-170K
LU	Approxilyzer	878K	809K	65.4K
	FastFlip	879K	815K	56.8K
	diff	+1.43K	+6.50K	-8.57K
SHA2	Approxilyzer	65.2K	318K	19.9K
	FastFlip	64.5K	319K	19.8K
	diff	-698	+798	-100

the approach and Columns 3-5 show the number of detected, SDC, and masked outcomes, respectively. The counts for the modified benchmarks have a similar trend.

The clearest difference in the outcome counts is the lower number of masked outcomes and the higher number of SDC outcomes identified by FastFlip. This is often the result of errors that cause an SDC in the output of a program section, which is then masked by a downstream section. Because FastFlip checks for SDCs the output of each section and conservatively assumes that SDCs in a section's output always propagate to the final output, it cannot identify such inter-section masking. This is especially visible for Campipe, whose last section is a nearest-value lookup table that masks many SDCs from upstream sections.

For LU and BScholes, FastFlip does not analyze a small number of error sites that are related to the iteration of the outer loop containing the analyzed program sections. As described in Section 5.3.7, FastFlip conservatively assumes that errors at these untested error sites always lead to SDC-Bad outcomes. As a result, FastFlip may select such untested instructions for protection even if errors in them do not result in SDCs.

Table 5.4: Comparison of FastFlip and Approxilyzer utility when all SDCs are unacceptable (SDC-Bad). A ✓ indicates that the achieved value is within the value error range of FastFlip.

		$\mathbf{v_{trgt}} = \mathbf{0.90}$		$\mathbf{v_{trgt}} = \mathbf{0.95}$		$\mathbf{v_{trgt}} = \mathbf{0.99}$	
Benchmark	**Modif.**	**Value**	**Cost (diff)**	**Value**	**Cost (diff)**	**Value**	**Cost (diff)**
BScholes	None	0.901✓	0.635 (+0.000)	0.950✓	0.717 (+0.000)	0.990✓	0.827 (+0.000)
	Small	0.899✓	0.634 (+0.003)	0.950✓	0.713 (+0.000)	0.990✓	0.821 (+0.000)
	Large	0.898✓	0.669 (+0.000)	0.949✓	0.753 (+0.000)	0.991✓	0.849 (+0.000)
Campipe	None	0.915✓	0.611 (+0.038)	0.950✓	0.676 (+0.017)	0.991✓	0.807 (+0.024)
	Small	0.924✓	0.611 (+0.060)	0.954✓	0.678 (+0.030)	0.990✓	0.807 (+0.034)
	Large	0.912✓	0.760 (+0.068)	0.961✓	0.819 (+0.043)	0.993✓	0.899 (+0.015)
FFT	None	0.900✓	0.544 (+0.011)	0.950✓	0.629 (+0.002)	0.990✓	0.780 (+0.000)
	Small	0.904✓	0.542 (+0.010)	0.950✓	0.629 (+0.004)	0.990✓	0.781 (+0.002)
	Large	0.900✓	0.492 (+0.001)	0.950✓	0.586 (−0.000)	0.987✓	0.716 (−0.016)
LU	None	0.900✓	0.603 (+0.000)	0.950✓	0.694 (+0.000)	0.990✓	0.873 (+0.000)
	Small	0.901✓	0.606 (+0.002)	0.951✓	0.698 (+0.002)	0.990✓	0.875 (+0.001)
	Large	0.902✓	0.560 (+0.002)	0.951✓	0.640 (+0.003)	0.990✓	0.826 (−0.001)
SHA2	None	0.900✓	0.666 (+0.001)	0.950✓	0.772 (+0.000)	0.990✓	0.908 (+0.001)
	Small	0.900✓	0.665 (+0.000)	0.949✓	0.771 (−0.001)	0.990✓	0.908 (+0.000)
	Large	0.883✓	0.476 (−0.007)	0.943✓	0.551 (−0.003)	0.985✓	0.655 (−0.007)

5.5.2 Utility of FastFlip Compared to That of Approxilyzer

Table 5.4 compares the utility of FastFlip and Approxilyzer for selective protection of instructions against SDCs, using the metrics described in Section 5.3.8. Columns 1-2 show the benchmark name and version, respectively. Subsequent pairs of columns show the utility comparison for the target protection values 0.90, 0.95, and 0.99, respectively. In each column pair, the first column shows FastFlip's achieved protection value. The second column shows the cost of protecting FastFlip's selection, and compares this to the cost of protecting Approxilyzer's selection.

With target adjustment, FastFlip successfully meets all target values for the original, unmodified version of each benchmark. Because FastFlip reuses the adjusted targets for the modified versions as described in Algorithm 5.2, it may not precisely meet the target for those modified versions. The maximum difference between the target and achieved values is 0.017 (1.7%) for SHA2-Large. In all cases, the target value is within FastFlip's achieved value error range (Section 5.4.5).

For most benchmarks, the cost of protecting FastFlip's selection of instructions is at most 0.011 (1.1%) more than the cost of protecting Approxilyzer's selection. The exception is Campipe, for which FastFlip's cost can be higher by as much as 0.068 (6.8%). This is because, unlike the other benchmarks, FastFlip has to aggressively adjust the target values for Campipe in order to compensate for the loss of precision caused by inter-section masking and meet the original target values. Section 5.5.4 describes the consequences of forgoing target adjustment.

Table 5.5: Comparison of FastFlip and Approxilyzer analysis time

		Analysis time (core-hours)		
Benchmark	**Modif.**	**FastFlip**	**Approxilyzer**	**Speedup**
BScholes	None	69 hrs	65 hrs	0.9×
	Small	42 hrs	62 hrs	1.5×
	Large	3 hrs	24 hrs	8.4×
Campipe	None	2459 hrs	2631 hrs	1.1×
	Small	158 hrs	2720 hrs	17.2×
	Large	45 hrs	494 hrs	11.0×
FFT	None	980 hrs	520 hrs	0.5×
	Small	300 hrs	509 hrs	1.7×
	Large	93 hrs	513 hrs	5.5×
LU	None	694 hrs	602 hrs	0.9×
	Small	80 hrs	625 hrs	7.8×
	Large	94 hrs	441 hrs	4.7×
SHA2	None	726 hrs	728 hrs	1.003×
	Small	718 hrs	726 hrs	1.01×
	Large	43 hrs	45 hrs	1.05×

We observed that if we removed the last section of Campipe (which is the primary cause of high inter-section masking in Campipe), FastFlip's target adjustments became less aggressive, and the excess cost of FastFlip decreased to 0.036 (3.6%). This suggests that more precise SDC propagation analyses that also calculate the probability of SDC masking may help to reduce the need for target adjustment.

The geomean cost of protecting FastFlip's selection is 0.601, 0.685, and 0.819 for the target protection values 0.90, 0.95, and 0.99, respectively. This shows that it is possible to protect against 90% bitflips that cause SDCs by protecting on average 60% of all dynamic instructions. Protecting against the remaining SDCs quickly leads to diminishing returns.

5.5.3 Performance of FastFlip Compared to That of Approxilyzer

Table 5.5 compares the analysis time of FastFlip (without simultaneously running Approxilyzer) and Approxilyzer. Columns 1-2 show the benchmark name and version, respectively. Columns 3-4 show the total analysis time for FastFlip and Approxilyzer, respectively. Column 5 shows the speedup of FastFlip over Approxilyzer. We measure analysis time in core-hours as the error injection analysis is highly parallelizable. The actual analysis time is much lower when using a large number of error injection experiment threads.

For FastFlip, over 99% of analysis time is for the error injection analysis. The sensitivity analysis requires less than five minutes. The symbolic SDC propagation analysis and knapsack problem solver each require less than one minute, even for programs and inputs that are much larger than our benchmarks.

The two approaches have similar analysis times for the unmodified (*None*) versions of all benchmarks except FFT. For FFT, Approxilyzer prunes a larger number of injections since operations such as transpose are performed multiple times in different dynamic sections of the program execution. As FastFlip injects errors into each section independently, it cannot similarly prune injections across sections. However, FastFlip is faster when analyzing the modified versions of FFT.

To enable target adjustment, FastFlip simultaneously runs the Approxilyzer analysis as described in Section 5.3.8. We use the methodology from [180, Section 4.7] to confirm that the time required for this simultaneous approach is at most 1% more than the greater of the analysis times of FastFlip and Approxilyzer for the unmodified versions of the benchmarks.

For the modified versions, the speedup depends on how much of the original program the developer replaced. If the modified sections are small with respect to the full program, then FastFlip must re-analyze only that small modified portion of the program, as opposed to Approxilyzer which must re-analyze the full modified program. This leads to the particularly large speedups for Campipe. On the other hand, if the modified sections are a large portion of the full program, then FastFlip must re-analyze large portions of the program, leading to smaller speedups. This leads to the negligible speedups for SHA2, where we modified the most expensive section of the program. As FastFlip reuses the adjusted targets for modified benchmarks, it only needs to use the simultaneous approach for the original version.

These results show that FastFlip can save a significant amount of analysis time when analyzing modified programs. For modern software systems that accumulate multiple small modifications over time, FastFlip can provide ever increasing savings.

5.5.4 Effects of Target Adjustment

Section 5.5.2 presents the results of FastFlip when it uses target adjustment in order to meet the original targets. For most benchmarks, the adjusted targets are within 0.4% of the original targets, so the results with and without target adjustment are similar. For Campipe, FastFlip has to aggressively adjust the target protection value because its characteristics lead to precision loss.

Table 5.6 compares the utility of FastFlip and Approxilyzer for Campipe with and without target adjustment. The format is similar to that of Table 5.4, except that Column 1 indicates whether the results are for the original (non-adjusted) target or the adjusted target. Without target adjustment, FastFlip undershoots the target by as much as 0.052 (5.2%) and

Table 5.6: Comparison of FastFlip and Approxilyzer utility for Campipe with and without target adjustment. A ✓ indicates that the achieved value is within the value error range of FastFlip, while a ✗ indicates the opposite.

Target type	**Modif.**	**Value**	**Cost (diff)**	**Value**	**Cost (diff)**	**Value**	**Cost (diff)**
		$\mathbf{v_{trgt} = 0.90}$		$\mathbf{v_{trgt} = 0.95}$		$\mathbf{v_{trgt} = 0.99}$	
Original	None	0.848✗	0.542 (−0.031)	0.920✓	0.622 (−0.038)	0.977✓	0.751 (−0.032)
	Small	0.879✓	0.543 (−0.008)	0.925✓	0.623 (−0.025)	0.980✓	0.752 (−0.021)
	Large	0.868✗	0.687 (−0.005)	0.925✗	0.776 (+0.001)	0.979✓	0.864 (−0.021)
		$\mathbf{v'_{trgt} = 0.942}$		$\mathbf{v'_{trgt} = 0.972}$		$\mathbf{v'_{trgt} = 0.997}$	
Adjusted	None	0.915✓	0.611 (+0.038)	0.950✓	0.676 (+0.017)	0.991✓	0.807 (+0.024)
	Small	0.924✓	0.611 (+0.060)	0.954✓	0.678 (+0.030)	0.990✓	0.807 (+0.034)
	Large	0.912✓	0.760 (+0.068)	0.961✓	0.819 (+0.043)	0.993✓	0.899 (+0.015)

Table 5.7: Comparison of FastFlip and Approxilyzer utility when SDCs greater than 0.01 are SDC-Bad. A ✓ indicates that the achieved value is within the value error range of FastFlip.

		$\mathbf{v_{trgt} = 0.90}$		$\mathbf{v_{trgt} = 0.95}$		$\mathbf{v_{trgt} = 0.99}$	
Benchmark	**Modif.**	**Value**	**Cost (diff)**	**Value**	**Cost (diff)**	**Value**	**Cost (diff)**
BScholes	None	0.900✓	0.645 (+0.013)	0.951✓	0.727 (+0.013)	0.990✓	0.821 (+0.000)
	Small	0.898✓	0.642 (+0.011)	0.949✓	0.724 (+0.013)	0.990✓	0.821 (+0.000)
	Large	0.892✓	0.681 (+0.012)	0.946✓	0.765 (+0.006)	0.990✓	0.849 (−0.012)
Campipe	None	0.900✓	0.576 (+0.031)	0.951✓	0.674 (+0.032)	0.991✓	0.802 (+0.030)
	Small	0.915✓	0.577 (+0.057)	0.958✓	0.674 (+0.054)	0.992✓	0.802 (+0.045)
	Large	0.903✓	0.694 (+0.024)	0.953✓	0.780 (+0.018)	0.992✓	0.895 (+0.016)
FFT	None	0.900✓	0.563 (+0.002)	0.950✓	0.687 (+0.004)	0.990✓	0.848 (+0.008)
	Small	0.904✓	0.579 (+0.012)	0.947✓	0.684 (−0.006)	0.989✓	0.845 (+0.002)
	Large	0.895✓	0.529 (−0.007)	0.936✓	0.625 (−0.028)	0.979✓	0.774 (−0.041)
LU	None	0.900✓	0.657 (+0.004)	0.950✓	0.787 (+0.000)	0.990✓	0.932 (+0.002)
	Small	0.906✓	0.674 (+0.020)	0.950✓	0.787 (+0.000)	0.990✓	0.932 (+0.000)
	Large	0.903✓	0.638 (+0.009)	0.943✓	0.753 (−0.007)	0.989✓	0.884 (−0.002)

the original target value does not always fall within FastFlip's achieved value error range. These results demonstrate the importance of target adjustment for ensuring that FastFlip meets the original protection targets for all benchmarks. However, aggressive target adjustment also leads to an increase in protection cost of FastFlip's selection of instructions over Approxilyzer's selection.

5.5.5 Comparison of FastFlip and Approxilyzer When Small SDCs are Acceptable

Table 5.7 compares the utility of FastFlip and Approxilyzer when small SDCs (≤ 0.01) are considered acceptable (SDC-Good) and the analyses focus on only protecting against larger SDCs (SDC-Bad). Table 5.7 has the same format as Table 5.4. We exclude SHA2 for this evaluation as its applications require the calculated hashes to be fully precise.

FastFlip successfully meets all target values for all benchmarks. The maximum difference

Table 5.8: Comparison of FastFlip and Approxilyzer utility for protecting against *all* SDC outcomes throughout the program execution. A ✓ indicates that the achieved value is within the value error range of Approxilyzer.

		$\mathbf{v_{trgt} = 0.99}$		$\mathbf{v_{trgt} = 1.00}$	
Benchmark	**Modif.**	**Value**	**Cost (diff)**	**Value**	**Cost (diff)**
BScholes	None	0.990✓	0.827 (+0.000)	1.000✓	0.954 (+0.010)
	Small	0.990✓	0.821 (+0.000)	1.000✓	0.953 (+0.011)
	Large	0.991✓	0.849 (+0.000)	1.000✓	0.958 (+0.024)
Campipe	None	0.991✓	0.807 (+0.024)	1.000✓	0.900 (+0.002)
	Small	0.990✓	0.807 (+0.034)	1.000✓	0.900 (+0.018)
	Large	0.993✓	0.899 (+0.015)	1.000✓	0.940 (+0.000)
FFT	None	0.990✓	0.780 (+0.000)	1.000✓	0.971 (+0.022)
	Small	0.990✓	0.781 (+0.002)	1.000✓	0.971 (+0.021)
	Large	0.987✓	0.716 (−0.016)	1.000✓	0.932 (+0.000)
LU	None	0.990✓	0.873 (+0.000)	1.000✓	0.982 (+0.000)
	Small	0.990✓	0.875 (+0.001)	1.000✓	0.983 (+0.000)
	Large	0.990✓	0.826 (−0.001)	1.000✓	0.932 (+0.000)
SHA2	None	0.990✓	0.908 (+0.001)	1.000✓	0.989 (+0.000)
	Small	0.990✓	0.908 (+0.000)	1.000✓	0.989 (+0.000)
	Large	0.985✓	0.655 (−0.007)	1.000✓	0.723 (+0.014)

between the target and achieved values is 0.014 (1.4%) for FFT-Large. In all cases, the target value is within FastFlip's achieved value error range caused by injection pruning. For most benchmarks, the cost of protecting FastFlip's selection of instructions is at most 0.020 (2%) more than the cost of protecting Approxilyzer's selection. The exception is Campipe, for which FastFlip's protection cost is higher by as much as 0.057 (5.7%) due to aggressive target adjustment.

The geomean cost of protecting FastFlip's selection is 0.619, 0.720, and 0.849 for the target protection values 0.90, 0.95, and 0.99, respectively. FastFlip obtains the results in Table 5.7 at the same time as the results in Table 5.4 for negligible additional analysis time (less than one minute).

5.5.6 Protecting Against *All* SDC Outcomes

Table 5.8 compares the utility of FastFlip and Approxilyzer for protecting against *all* SDC outcomes. The format is similar to that of Table 5.4. Table 5.8 repeats the results for $v_{trgt} = 0.99$ for comparison.

FastFlip achieves a protection value of 1.000 (100%) for $v_{trgt} = 1.00$. It is not necessary (or possible) to adjust this target value. However, for some benchmarks, FastFlip also selects for protection some instructions for which Approxilyzer found only masked or detected outcomes. As a result, the overhead of FastFlip's selection is still slightly higher than that

```
modifiedCode(input) {
  output1 = originalCode(input);
  output2 = originalCode(input);
  assert(output1 == output2);
  return output1;
}
```

Figure 5.3: Coarse-grained SDC detection mechanism

Table 5.9: Effect of adding error detection mechanisms to a static section of BScholes

	Error outcomes	**Analysis time (core-hours)**		
Modif.	**SDC / Total (%)**	**FastFlip**	**Approxilyzer**	**Speedup**
None	2991 / 5536 (54.0%)	69 hrs	65 hrs	0.9×
Error detection	882 / 8960 (9.8%)	21 hrs	71 hrs	3.4×

of Approxilyzer's selection. The geomean cost of protecting FastFlip's selection is 0.819 and 0.936 for the target protection values 0.99 and 1.00, respectively. There is a significant increase in the number of dynamic instructions that must be protected to protect against the last 1% of SDC-causing errors. The results for protecting against all SDC outcomes whose magnitude is greater than 0.01 are similar.

5.5.7 Case Study: Effectiveness of Error Detection Mechanisms

To verify that error detection mechanisms can reduce the likelihood of SDC outcomes as a result of errors, we modified one static section of the BScholes benchmark. The modified section executes the original code twice and compares the output of the two executions. If the outputs do not match, this indicates that an error occurred, and the program raises an exception (a detected outcome), as illustrated by the pseudocode in Figure 5.3. We analyze the effects of errors in the modified section with FastFlip and Approxilyzer to check the effectiveness of such an error detection mechanism.

Table 5.9 shows the results of adding this error detection mechanism to BScholes. Column 2 shows the number of SDC outcomes as a fraction of the total error outcomes. Columns 3-4 show the total analysis time for FastFlip and Approxilyzer, respectively. Column 5 shows the speedup of FastFlip over Approxilyzer. We find that the modification significantly reduces the number and fraction of SDC outcomes occurring due to errors in the modified section; this is despite the increase in code size due to duplication. The remaining SDCs occur in 1) non-duplicated instructions at the start or end of the modified section, or 2) instructions within the first execution of the original code that cause side effects that affect the second execution too. As with other modifications, FastFlip saves time over Approxilyzer because it must only analyze the modified sections of the benchmark.

5.6 RELATED WORK

Error injection-less reliability analyses. ePVF [219] is a dynamic analysis which finds locations where a bitflip will cause a crash, as opposed to an SDC, with $\sim 90\%$ accuracy. TRIDENT [57] uses empirical observations of error propagation in programs to predict the overall SDC probability of a program and the SDC probabilities of individual instructions. Other works [220, 221] use analytical modeling to detect SDCs in a program. Leto [48] is an SMT-based fault tolerance analysis that supports multiple error models, including the single error model. However, it requires precise specifications of program components, which are cumbersome for large programs when errors can occur at any point. While these analyses can be faster than error injection analyses, they are less accurate and may not be able to precisely estimate the magnitude of the output SDC due to an error. FastFlip's compositional nature makes error injection analysis more affordable by amortizing the cost of analyzing evolving programs over time.

Aloe (Chapter 4) statically analyzes programs with potentially imperfect error recovery mechanisms to determine the overall reliability of the program. Unlike FastFlip, Aloe supports error models where multiple errors can simultaneously occur during program execution with varying probabilities. This is a limitation of FastFlip, as the correctness of Equation 5.5 relies on the assumption that only one error occurs during program execution.

Error injection analyses. Error injection analyses operate at different levels of abstraction, including hardware, assembly, and IR [55, 56, 58, 59, 197, 198, 200, 201, 202, 203, 204, 205]. These analyses typically use *sampling* - they select a statistically significant number of error sites at random and only perform error injections at those sites. While this is sufficient for providing overall outcome statistics as we do in Section 5.5.1, we cannot use such results to determine which specific instructions or blocks of instructions are particularly vulnerable to SDCs in order to protect them. However, FastFlip can still use these analyses if they are modified to perform per-instruction error injection like Approxilyzer [54, 199].

Minotaur [180] reduces the size of inputs required to test the reliability of programs when subjected to error injections, while keeping the percentage of static instructions evaluated close to 100% (as compared to the reference input). We use the input sizes proposed by Minotaur for the FFT and LU benchmarks, and manually minimize the Minotaur-proposed BScholes input to 2 options without sacrificing instruction coverage. While Minotaur reduces injection analysis time by reducing input size, the program must still be analyzed in full.

FastFlip complements Minotaur by adding the flexibility of only re-analyzing small sections of the program when developers modify it, further reducing analysis time.

Papadimitriou and Gizopoulos [207] show that injecting errors in various SRAM hardware structures can give different results compared to injecting errors at higher levels of abstraction. AVGI [56] builds on [207] to show that hardware errors manifest in software in different ways, but result in similar distributions of final outcomes across applications. Using this insight, AVGI accelerates hardware-level fault injection for large workloads to provide overall outcome statistics. Santos et al. [222] similarly examine how faults injected at the RTL level affect common GPU instructions, and inject these instruction-level effects into applications to provide overall outcome statistics and identify vulnerable hardware components. However, FastFlip requires the error injection analysis to report outcomes for errors at each possible error site, as opposed to summary statistics (Section 5.3.10). The above analysis techniques that aim to efficiently determine the effect of low-level faults via hybrid fault injection are too slow even for small program sizes when modified to report outcomes in the manner required by FastFlip. If future analyses succeed in providing such detailed outcome information for low-level faults in a scalable manner, FastFlip would be able to use them to improve the accuracy of its analysis.

SDC propagation analyses. SDC propagation analyses either propagate SDCs forward through programs [46, 223], or propagate SDC bounds backwards through programs [36, 45, 86]. While we instantiated FastFlip with the Chisel [36] backwards SDC propagation analysis, the FastFlip approach can instead use any alternate SDC propagation analysis that satisfies the requirements outlined in Section 5.3.10.

Ashraf et al. [206] analyze the propagation of randomly injected faults in MPI applications. Using the injection experiment results, they build a model to estimate the number of memory locations corrupted over time to guide roll-back decisions. Combining FastFlip with [206] is an attractive opportunity for making per-instruction error injection analysis of MPI applications practical.

Mutlu et al. [60] predict the effect of bitflips injected into iterative applications on the final output by analyzing the effects of fault injections on a limited number of iterations. While this potentially gives [60] an advantage over FastFlip for applications that iterate the same operation multiple times, unlike FastFlip, [60] cannot handle applications with multiple sections that perform distinct operations, such as our benchmarks.

Hardware-based selective protection. Researchers have examined the use of selective hardware hardening (e.g., via redundancy or ECC) for improving hardware reliability while limiting the use of additional chip area [224, 225, 226, 227]. These techniques find and replicate only those hardware components that, as a result of transient errors, produce unacceptable outcomes across the range of typical applications that users are expected to run on the hardware. The error model we use for FastFlip's evaluation (Section 5.4.2) also assumes that caches and certain registers are protected within the hardware. FastFlip efficiently provides information which can be used to apply *additional*, software-based selective protection tailored to the needs of specific applications, as opposed to adding further hardware protections irrelevant to other applications.

Software-based selective SDC protection. Unlike crashes, timeouts, or clearly invalid data, SDCs are more difficult to detect by nature. SWIFT [70] uses instruction duplication to detect errors in computational instructions. To reduce overhead, it makes use of instruction reordering by the compiler and the processor. DRIFT [71] further reduces overhead by coalescing the checks of multiple duplicated instructions, which reduces basic block fragmentation. SWIFT and DRIFT *completely* eliminate the possibility of SDCs occurring due to single bitflip errors in the duplicated computational instructions. nZDC [196] provides comparable overhead to SWIFT while also protecting programs from 99.6% of SDCs caused by single bitflip errors during load, store, and control flow instructions.

Shoestring [72] finds and duplicates only particularly vulnerable instructions. Hari et al. [68] propose protecting blocks of instructions with single detectors placed at the end of loops or function calls. These two techniques use the results of error injection analyses to guide *selective* instruction duplication. We consider such techniques to be *clients* of FastFlip. They provide FastFlip with information about the runtime overhead of protecting various instructions or instruction blocks. Our evaluation in fact uses an adapted version of the value and cost model from [68]. In return, FastFlip provides precise information on which instructions should be protected in order to minimize runtime overhead while protecting against a developer-defined fraction of SDC-causing errors. After developers apply these techniques to protect FastFlip's selection of instructions, FastFlip can re-analyze the protected sections to confirm the decrease in SDC vulnerability. For FastFlip, we focused on efficiently handling program modifications in general. We also briefly explore a modification that reduces SDC vulnerability via coarse-grained code duplication in Section 5.5.7. Analyzing sections after applying fine-grained duplication of FastFlip's selection of vulnerable instructions is an interesting topic for future work.

Chapter 6: Conclusion and Future Work

6.1 CONCLUSION

Uncertainty is an unavoidable aspect of modern computations. It is either present within the inputs themselves, or has to be intentionally introduced to solve otherwise intractable problems. These computations are capable of tolerating uncertainty to differing extents. For this reason, researchers have put significant effort into analyzing how uncertainty is introduced into a computation, how it propagates through it, and how it affects the final outputs. Researchers have also put effort into designing various techniques for reducing uncertainty when necessary.

The cost of analyzing the uncertainty in modern computations is further compounded by the fact that they are continuously evolving not just during development, but also after they have been deployed in a production environment. Compositional analyses of uncertainty would help developers by reducing analysis time when incremental changes are made to a computation, as it would only be necessary to re-analyze modified components when changes are made. Unfortunately, little work exists that aims to make uncertainty analysis of computations composable.

In this dissertation, I presented my work that shows that it is possible to analyze uncertainty in computations in a composable manner. First, I described two analyses of critical components of computations. AxProf is a framework for statistical analysis of approximate randomized algorithm implementations that ensures that the implementations satisfy theoretical accuracy constraints. Aloe is a static analysis that determines how recovery mechanisms protect critical sub-computations from excessive uncertainty.

Second, I described two composable analyses of end-to-end computations. GAS focuses on simulations of autonomous vehicle systems and separately analyzes the expensive to simulate perception components. FastFlip accelerates error injection analysis by dividing programs into sections and using an uncertainty propagation analysis to determine the effect of errors in each section on the final outputs of the program. By analyzing computations as separable components, these analyses reduce the cost of analysis over time. Specifically, they reduce the need to re-analyze unmodified components of computations that evolve over time. In doing so, these analyses are often able to remain as precise as monolithic, non-composable analyses of the whole computation.

Inexpensive, composable analysis of uncertainty will enable developers to rapidly test proposed changes to modern computations to determine their response to uncertainty. Uncertainty analysis could become commonplace in regression testing suites for modern computations. The work that I have presented in this dissertation serves as a starting point towards this goal.

6.2 FUTURE WORK

Supporting application-specific recovery mechanisms in Aloe. Aloe supports recovery mechanisms that redo a computation if it detects an error. Aloe can also model other recovery mechanisms as a re-execution of a computation, such as checkpoint-restore and the re-evaluation of eagerly executed code that produced unsatisfactory results.

Some applications can employ application-specific recovery mechanisms. For example, many iterative applications whose output converges to a fixed point can recover from some types of errors by simply increasing the number of iterations performed. For some other applications, it may be less expensive to use an alternative algorithm to correct a slightly erroneous output, than to redo the full computation. Adding support for such custom recovery mechanisms would increase the applicability of Aloe.

Supporting complex autonomous vehicle tasks in GAS. Our GAS benchmarks model vehicle systems that perform essential, narrowly-focused tasks. Specifically, the benchmarks model vehicles that attempt to avoid collisions with other vehicles or landscape features. GAS is capable of accurately modeling vehicles performing such tasks.

The latest autonomous vehicles are capable of performing complex, multi-stage tasks such as navigating across a city. These autonomous vehicle systems have a large and varied state space, including cameras, LIDAR, GPS, etc. as well as complex internal states. Adding support for modeling such vehicle systems in GAS could be made possible using the strategies discussed in Section 3.6.1 and would increase the applicability of GAS.

Testing modifications that implement recovery with FastFlip. FastFlip is designed to be able to efficiently analyze programs after various types of modifications. We demonstrate this by analyzing modifications that improve code readability, remove redundant computation, or completely overhaul program components.

We also briefly explore the effectiveness of a modification that uses coarse-grained code duplication to reduce the likelihood of SDCs. While such a modification does reduce the likelihood of SDCs, it is inefficient as it protects a block instructions with varying vulnerability to SDCs as a single unit. FastFlip's results enable fine-grained instruction duplication of a set of instructions that are especially vulnerable to SDCs. After adding such protections, we should be able to use FastFlip to verify that the protected instructions are no longer vulnerable. This would enable a four step procedure in which developers modify a program section, analyze the modified section with FastFlip to find vulnerable instructions, protect those instructions, and then verify the protections with FastFlip.

Testing phase-aware protection strategies with FastFlip. In our evaluation of FastFlip, we observed that the same static section of code had varying vulnerability to errors depending on the current phase of the end-to-end computation. FastFlip currently selects instructions within a section for protection based on the vulnerability of those instructions over the course of the whole computation.

An interesting extension of FastFlip would be to select different instructions for protection at different phases of the computation. This would be achieved by compiling multiple versions of the same program section and selecting the version to use based on the current phase. This could lead to less expensive protection of the end-to-end computation by targeting instructions for protection based on their phase specific vulnerability to SDCs.

Input-agnostic analysis of programs with FastFlip. If a program executes a static section of code multiple times with different inputs, FastFlip must analyze each dynamic instance of that section separately. Similarly, if the developer decides to test the end-to-end computation with an entirely new input, FastFlip must perform its analysis for every section from scratch.

For some static instructions in the program, the outcome of certain errors in that instruction may be input-agnostic. By analyzing the outcomes for the same error site across multiple inputs, it may be possible to identify such input-agnostic outcomes. FastFlip can then skip analyzing these error sites if the input changes at a later point. The advantages of this approach would be threefold:

- It would speed up FastFlip's analysis of multiple dynamic instances of the same static section by reducing the number of error injections that must be performed for later dynamic sections in the execution.
- It would allow FastFlip to skip some error injections if the developer changes the input to the end-to-end computation.
- It would enable more efficient analysis of modifications that do not preserve semantics by reducing the number of error injections that must be performed for downstream sections whose input changes as a result of the modification.

References

[1] W. Zhuang, X. Chen, J. Tan, and A. Song, “An empirical analysis for evaluating the link quality of robotic sensor networks,” in *International Conference on Wireless Communications and Signal Processing*, 2009.

[2] S. Miura, L.-T. Hsu, F. Chen, and S. Kamijo, “GPS error correction with pseudorange evaluation using three-dimensional maps,” *IEEE Transactions on Intelligent Transportation Systems*, 2015.

[3] M. Segata, R. L. Cigno, R. K. Bhadani, M. Bunting, and J. Sprinkle, “A LiDAR error model for cooperative driving simulations,” in *IEEE Vehicular Networking Conference*, 2018.

[4] S. Borkar, “Designing reliable systems from unreliable components: The challenges of transistor variability and degradation,” *IEEE Micro*, 2005.

[5] N. DeBardeleben, J. Laros, J. T. Daly, S. L. Scott, C. Engelmann, and B. Harrod, “High-end computing resilience: Analysis of issues facing the hec community and path-forward for research and development,” Oak Ridge National Laboratory, Tech. Rep., 2009.

[6] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, and M. Snir, “Toward exascale resilience: 2014 update,” *Supercomputing Frontiers and Innovations: an International Journal*, 2014.

[7] R. Hegde and N. Shanbhag, “Toward achieving energy efficiency in presence of deep submicron noise,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2000.

[8] S. Kose, E. Salman, and E. G. Friedman, “Shielding methodologies in the presence of power/ground noise,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2011.

[9] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo, “Understanding silent data corruptions in a large production CPU population,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[10] V. Pratap, A. Hannun, G. Synnaeve, and R. Collobert, “Star temporal classification: Sequence modeling with partially labeled data,” in *Advances in Neural Information Processing Systems*, 2022.

[11] J. Amores, “Multiple instance classification: Review, taxonomy and comparative study,” *Artificial Intelligence*, 2013.

[12] X. Gong, D. Yuan, and W. Bao, "Understanding partial multi-label learning via mutual information," in *Advances in Neural Information Processing Systems*, 2021.

[13] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the 30th annual ACM symposium on Theory of computing*, 1998.

[14] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Communications of the ACM*, 2008.

[15] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *Analysis of Algorithms*, 2007.

[16] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *European Symposium on Algorithms*, 2002.

[17] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the Count-Min sketch and its applications," *Journal of Algorithms*, 2005.

[18] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, 1985.

[19] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, 1970.

[20] P. Drineas, R. Kannan, and M. W. Mahoney, "Fast Monte Carlo algorithms for matrices i: Approximating matrix multiplication," *SIAM Journal on Computing*, 2006.

[21] N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM review*, 2011.

[22] P. Drineas and M. W. Mahoney, "RandNLA: randomized numerical linear algebra," *Communications of the ACM*, 2016.

[23] H. Shatkay and S. B. Zdonik, "Approximate queries and representations for large data sequences," in *Proceedings of the 12th International Conference on Data Engineering*, 1996.

[24] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[25] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo, "ABS: a system for scalable approximate queries with accuracy guarantees," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014.

[26] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, “Approxhadoop: Bringing approximations to mapreduce frameworks,” in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[27] R. R. Shamshiri, C. Weltzien, I. A. Hameed, I. J. Yule, T. E. Grift, S. K. Balasundram, L. Pitonakova, D. Ahmad, and G. Chowdhary, “Research and development in agricultural robotics: A perspective of digital farming,” *International Journal of Agricultural and Biological Engineering*, 2018.

[28] P. J. Mosterman, D. Escobar Sanabria, E. Bilgin, K. Zhang, and J. Zander, “Automating humanitarian missions with a heterogeneous fleet of vehicles,” *Annual Reviews in Control*, 2014.

[29] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski, “Simulation-based adversarial test generation for autonomous vehicles with machine learning components,” in *IEEE Intelligent Vehicles Symposium*, 2018.

[30] Z. Wang, W. Ren, and Q. Qiu, “LaneNet: Real-time lane detection networks for autonomous driving,” 2018.

[31] C. Dwork, F. McSherry, K. Nissim, and A. Smith, “Calibrating noise to sensitivity in private data analysis,” *Journal of Privacy and Confidentiality*, 2017.

[32] C. Dwork and A. Roth, “The algorithmic foundations of differential privacy,” *Foundations and Trends in Theoretical Computer Science*, 2014.

[33] D. Le Gall, “MPEG: A video compression standard for multimedia applications,” *Communications of the ACM*, 1991.

[34] G. Wallace, “The JPEG still picture compression standard,” *IEEE Transactions on Consumer Electronics*, 1992.

[35] P. Stanley-Marbell and M. Rinard, “Perceived-color approximation transforms for programs that draw,” *IEEE Micro*, 2018.

[36] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, “Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[37] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, “Quality of service profiling,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010.

[38] S. Mazumder, *Chapter 3 - Solution to a System of Linear Algebraic Equations.* Academic Press, 2016.

[39] P. Deuflhard, *Newton Methods for Nonlinear Problems.* Springer Berlin, Heidelberg, 2011.

[40] European Union Aviation Safety Agency, "EASA concept paper: First usable guidance for level 1 machine learning applications," 2021. [Online]. Available: https://www.easa.europa.eu/en/downloads/134357/en

[41] P. Koopman, U. Ferrell, F. Fratrik, and M. Wagner, "A safety standard approach for fully autonomous vehicles," in *Computer Safety, Reliability, and Security: SAFECOMP Workshops*, 2019.

[42] Federal Aviation Administration, "Unmanned aircraft system traffic management (UTM) concept of operations version 2.0," 2022. [Online]. Available: https://www.faa.gov/sites/faa.gov/files/2022-08/UTM_ConOps_v2.pdf

[43] "Road vehicles - functional safety," Website, https://www.iso.org/standard/43464.html.

[44] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013.

[45] V. Fernando, K. Joshi, and S. Misailovic, "Verifying safety and accuracy of approximate parallel programs via canonical sequentialization," in *Proceedings of the ACM on Programming Languages*, no. OOPSLA, 2019.

[46] V. Fernando, K. Joshi, and S. Misailovic, "Diamont: Dynamic monitoring of uncertainty for distributed asynchronous programs," in *International Conference on Runtime Verification*, 2021.

[47] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[48] B. Boston, Z. Gong, and M. Carbin, "Leto: verifying application-specific hardware fault tolerance with programmable execution models," in *Proceedings of the ACM on Programming Languages*, no. OOPSLA, 2018.

[49] M. Carbin and M. Rinard, "Automatically identifying critical input regions and code in applications," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010.

[50] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Monitoring and debugging the quality of results in approximate programs," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[51] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proceedings of the 20th Annual International Conference on Supercomputing*, 2006.

[52] G. Li, S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. Keckler, "Understanding error propagation in deep-learning neural networks (DNN) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[53] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[54] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[55] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," in *IEEE International Conference on Software Quality, Reliability and Security*, 2013.

[56] G. Papadimitriou and D. Gizopoulos, "AVGI: Microarchitecture-driven, fast and accurate vulnerability assessment," in *IEEE International Symposium on High-Performance Computer Architecture*, 2023.

[57] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018.

[58] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017.

[59] J. Calhoun, L. Olson, and M. Snir, "FlipIt: An LLVM based fault injector for HPC," in *European Conference on Parallel Processing Workshops*, 2014.

[60] B. O. Mutlu, G. Kestor, A. Cristal, O. Unsal, and S. Krishnamoorthy, "Ground-truth prediction to accelerate soft-error impact analysis for iterative methods," in *IEEE 26th International Conference on High Performance Computing, Data, and Analytics*, 2019.

[61] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM Transactions on Embedded Computing Systems*, 2013.

[62] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[63] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[64] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[65] 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*, 2011.

[66] V. Fernando, A. Franques, S. Abadal, S. Misailovic, and J. Torrellas, "Replica: A wireless manycore for communication-intensive and approximate data," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[67] S. Sahoo, M. Li, P. Ramchandran, S. Adve, V. Adve, and Y. Zhou, "Using likely program invariants to detect hardware errors," in *IEEE International Conference on Dependable Systems and Networks*, 2008.

[68] S. Hari, S. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012.

[69] G. Lyle, S. Cheny, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "An end-to-end approach for the automatic derivation of application-aware error detectors," in *IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009.

[70] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, 2005.

[71] K. Mitropoulou, V. Porpodas, and M. Cintra, "DRIFT: Decoupled compileR-based Instruction-level Fault-Tolerance," in *Languages and Compilers for Parallel Computing*, 2014.

[72] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[73] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra, "CLEAR: Cross-layer exploration for architecting resilience - combining hardware and software techniques to tolerate soft errors in processor cores," in *53rd ACM/EDAC/IEEE Design Automation Conference*, 2016.

[74] S. Achour and M. Rinard, "Approximate computation with outlier detection in Topaz," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2015.

[75] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[76] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[77] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: Using canary inputs to dynamically steer approximation," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[78] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi, "Phase-aware optimization in approximate computing," in *IEEE/ACM International Symposium on Code Generation and Optimization*, 2017.

[79] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, S. Misailovic, and S. Bagchi, "Videochef: efficient approximation for streaming video processing pipelines," in *USENIX Annual Technical Conference*, 2018.

[80] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[81] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[82] K. Kennedy and J. Allen, *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufman, 2002.

[83] C. S. Păsăreanu, D. Gopinath, and H. Yu, *Compositional Verification for Autonomous Systems with Deep Learning Components.* Springer International Publishing, 2019.

[84] K. Joshi, V. Fernando, and S. Misailovic, "Statistical algorithmic profiling for randomized approximate programs," in *IEEE/ACM 41st International Conference on Software Engineering*, 2019.

[85] K. Joshi, C. Hsieh, S. Mitra, and S. Misailovic, "GAS: Generating fast and accurate surrogate models for autonomous vehicle systems," 2024. [Online]. Available: https://arxiv.org/abs/2208.02232

[86] K. Joshi, V. Fernando, and S. Misailovic, "Aloe: Verifying reliability of approximate programs in the presence of recovery mechanisms," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020.

[87] K. Joshi, R. Singh, T. Bassetto, S. Adve, S. Misailovic, and D. Marinov, "FastFlip: Compositional error injection analysis of programs," 2024. [Online]. Available: https://arxiv.org/abs/2403.13989

[88] J. Misra and D. Gries, "Finding repeated elements," *Science of computer programming*, 1982.

[89] J. Tropp, "An introduction to matrix concentration inequalities," *Foundations and Trends in Machine Learning*, 2015.

[90] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica, "Knowing when you're wrong: building fast and reliable approximate query processing systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014.

[91] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[92] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[93] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan, "Quality configurable reduce-and-rank for energy efficient approximate computing," in *Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[94] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, 1982.

[95] J. Huang, B. Mozafari, and T. F. Wenisch, "Statistical analysis of latency through semantic profiling," in *Proceedings of the 12th European Conference on Computer Systems*, 2017.

[96] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.

[97] E. Coppa, C. Demetrescu, and I. Finocchi, “Input-sensitive profiling,” *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[98] D. Zaparanuks and M. Hauswirth, “Algorithmic profiling,” *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[99] A. Arcuri and L. Briand, “A practical guide for using statistical tests to assess randomized algorithms in software engineering,” in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[100] R. Motwani and P. Raghavan, *Randomized algorithms.* Chapman & Hall/CRC, 2010.

[101] R. A. Fisher, *Statistical methods for research workers.* Oliver and Boyd, 1925.

[102] JorenSix, “TarsosLSH,” https://github.com/JorenSix/TarsosLSH.

[103] A. Bogdanov, L. Trevisan et al., “Average-case complexity,” *Foundations and Trends in Theoretical Computer Science*, 2006.

[104] M. M. Wagner-Menghin, *Binomial Test.* Wiley Online Library, 2014.

[105] H. Cramér, *Mathematical methods of statistics.* Princeton university press, 2016.

[106] A. Wald, “Sequential tests of statistical hypotheses,” *The annals of mathematical statistics*, 1945.

[107] H. L. Younes, M. Kwiatkowska, G. Norman, and D. Parker, “Numerical vs. statistical probabilistic model checking,” *International Journal on Software Tools for Technology Transfer*, 2006.

[108] C. John, W. Cleveland, B. Kleiner, and P. Tukey, “Graphical methods for data analysis,” *Wadsworth, Ohio*, 1983.

[109] S. Mitra, G. Bronevetsky, S. Javagal, and S. Bagchi, “Dealing with the unknown: Resilience to prediction errors,” in *International Conference on Parallel Architecture and Compilation*, 2015.

[110] D. Reshef, Y. Reshef, H. Finucane, S. Grossman, G. McVean, P. Turnbaugh, E. Lander, M. Mitzenmacher, and P. Sabeti, “Detecting novel associations in large data sets,” *Science*, 2011.

[111] E. Jones, T. Oliphant, P. Peterson et al., “SciPy: Open source scientific tools for Python,” 2001. [Online]. Available: https://scipy.org/

[112] tdebatty, “java-LSH,” https://github.com/tdebatty/java-LSH.

[113] mavam, “libbf,” https://mavam.github.io/libbf.

[114] MagnusS, "java-bloomfilter," https://github.com/MagnusS/Java-BloomFilter.

[115] alabid, "Count-Min Sketch," https://github.com/alabid/countminsketch.

[116] AWNystrom, "CountMinSketch," https://github.com/AWNystrom/CountMinSketch.

[117] Yahoo! Inc., "Sketches library from Yahoo," https://datasketches.apache.org/.

[118] ekzhu, "Datasketch," https://github.com/ekzhu/datasketch.

[119] alexpreynolds, "sample," https://github.com/alexpreynolds/sample.

[120] NumericalMax, "Randomized Matrix Product," https://github.com/NumericalMax/Randomized-Matrix-Product.

[121] gnu-user, "mcsc," https://github.com/gnu-user/mcsc-6030-project.

[122] S. Misailović, "Accuracy-aware optimization of approximate programs," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.

[123] E. Cohen, "All-distances sketches, revisited: HIP estimators for massive graphs analysis," *IEEE Transactions on Knowledge and Data Engineering*, 2015.

[124] M. Levandowsky and D. Winter, "Distance between sets," *Nature*, 1971.

[125] C. Curtsinger and E. D. Berger, "Coz: finding code that counts with causal profiling," in *Symposium on Operating Systems Principles*, 2015.

[126] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[127] E. Raman and D. I. August, "Recursive data structure profiling," in *Workshop on Memory system performance*, 2005.

[128] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *IEEE 31st International Conference on Software Engineering*, 2009.

[129] L. Song and S. Lu, "Statistical debugging for real-world performance problems," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[130] A. Zheng, M. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous identification of multiple bugs," in *Proceedings of the 23rd International Conference on Machine Learning*, 2006.

[131] K. Gopinathan and I. Sergey, "Certifying certainty and uncertainty in approximate membership query structures," in *Computer Aided Verification*, 2020.

[132] K. Sen, M. Viswanathan, and G. Agha, "Statistical model checking of black-box probabilistic systems," in *International Conference on Computer Aided Verification*, 2004.

[133] K. Sen, M. Viswanathan, and G. Agha, "On statistical model checking of stochastic systems," in *International Conference on Computer Aided Verification*, 2005.

[134] V. Fernando, K. Joshi, D. Marinov, and S. Misailovic, "Identifying optimal parameters for approximate randomized algorithms," 2019. [Online]. Available: http://approximate.computer/wax2019/papers/fernando.pdf

[135] A. Mahmoud, P. Reckamp, P. Tang, C. Fletcher, and S. Adve, "Approximate checkers," in *Workshop on Approximate Computing*, 2019.

[136] A. Afzal, D. S. Katz, C. L. Goues, and C. S. Timperley, "A study on the challenges of using robotics simulators for testing," 2020. [Online]. Available: https://arxiv.org/abs/2004.07368

[137] A. Afzal, C. L. Goues, M. Hilton, and C. S. Timperley, "A study on challenges of testing robotic systems," in *IEEE 13th International Conference on Software Testing, Validation and Verification*, 2020.

[138] C. Menghi, S. Nejati, L. Briand, and Y. I. Parache, "Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification," in *IEEE/ACM 42nd International Conference on Software Engineering*, 2020.

[139] C. P. Robert, G. Casella, and G. Casella, *Monte Carlo statistical methods.* Springer, 2004.

[140] J. Lin, H. Li, Y. Huang, Z. Huang, and Z. Luo, "Adaptive artificial neural network surrogate model of nonlinear hydraulic adjustable damper for automotive semi-active suspension system," *IEEE Access*, 2020.

[141] D. Xiu, *Numerical Methods for Stochastic Computations: A Spectral Method Approach.* Princeton University Press, 2010.

[142] G. Kewlani, J. Crawford, and K. Iagnemma, "A polynomial chaos approach to the analysis of vehicle dynamics under uncertainty," *Vehicle System Dynamics*, 2012.

[143] I. Sobol, "Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates," *Mathematics and Computers in Simulation*, 2001.

[144] A. N. Sivakumar, S. Modi, M. V. Gasparino, C. Ellis, A. E. B. Velasquez, G. Chowdhary, and S. Gupta, "Learned visual navigation for under-canopy agricultural robots," 2021.

[145] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[146] X. Cheng, B. Khomtchouk, N. Matloff, and P. Mohanty, “Polynomial regression as an alternative to neural nets,” 2019.

[147] R. Pan and H. Rajan, “On decomposing a deep neural network into modules,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[148] O. Ernst, A. Mugler, H.-J. Starkloff, and E. Ullmann, “On the convergence of generalized polynomial chaos expansions,” *Mathematical Modelling and Numerical Analysis*, 2012.

[149] MaybeShewill-CV, “Unofficial implemention of lanenet model for real time lane detection using deep neural network model,” https://github.com/MaybeShewill-CV/lanenet-lane-detection.

[150] P. Du, Z. Huang, T. Liu, T. Ji, K. Xu, Q. Gao, H. Sibai, K. Driggs-Campbell, and S. Mitra, “Online monitoring for safe pedestrian-vehicle interactions,” in *IEEE 23rd International Conference on Intelligent Transportation Systems*, 2020.

[151] K. D. Julian and M. J. Kochenderfer, “Guaranteeing safety for neural network-based aircraft collision avoidance systems,” in *Digital Avionics Systems Conference*, 2019.

[152] J. Feinberg and H. P. Langtangen, “Chaospy: An open source tool for designing methods of uncertainty quantification,” *Journal of Computational Science*, 2015.

[153] K. Konakli and B. Sudret, “Uncertainty quantification in high dimensional spaces with low-rank tensor approximations,” in *1st International Conference on Uncertainty Quantification in Computational Sciences and Engineering*, 2015.

[154] J. Feinberg and H. P. Langtangen, “Truncation scheme - chaospy,” https://chaospy.readthedocs.io/en/master/user_guide/polynomial/truncation_scheme.html.

[155] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, “DryVR: Data-driven verification and compositional reasoning for automotive systems,” in *Computer Aided Verification*, 2017.

[156] S. Jha, V. Raman, D. Sadigh, and S. Seshia, “Safe autonomy under perception uncertainty using chance-constrained temporal logic,” *Journal of Automated Reasoning*, 2018.

[157] R. Calinescu, C. Imrie, R. Mangal, C. Păsăreanu, M. A. Santana, and G. Vázquez, “Discrete-event controller synthesis for autonomous systems with deep-learning perception components,” 2022.

[158] M. Althoff and J. M. Dolan, “Online verification of automated road vehicles using reachability analysis,” *IEEE Transactions on Robotics*, 2014.

[159] P. Musau, N. Hamilton, D. M. Lopez, P. Robinette, and T. T. Johnson, “On using real-time reachability for the safety assurance of machine learning controllers,” in *IEEE International Conference on Assured Autonomy*, 2022.

[160] Y. Yang, R. Kaur, S. Dutta, and I. Lee, “Interpretable detection of distribution shifts in learning enabled cyber-physical systems,” in *ACM/IEEE 13th International Conference on Cyber-Physical Systems*, 2022.

[161] C.-H. Cheng, C.-H. Huang, and H. Yasuoka, “Quantitative projection coverage for testing ML-enabled autonomous systems,” in *Automated Technology for Verification and Analysis*, 2018.

[162] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, “Scenic: A language for scenario specification and scene generation,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[163] R. Li, T. Qin, P. Yang, C.-C. Huang, Y. Sun, and L. Zhang, “Safety analysis of autonomous driving systems based on model learning,” 2022. [Online]. Available: https://arxiv.org/abs/2211.12733

[164] R. Michelmore, M. Wicker, L. Laurenti, L. Cardelli, Y. Gal, and M. Kwiatkowska, “Uncertainty quantification with statistical guarantees in end-to-end autonomous driving control,” in *IEEE International Conference on Robotics and Automation*, 2020.

[165] H.-D. Tran, X. Yang, D. Manzanas Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson, “NNV: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems,” in *Computer Aided Verification, 32nd International Conference*, 2020.

[166] S. Bak, H.-D. Tran, K. Hobbs, and T. T. Johnson, “Improved geometric path enumeration for verifying ReLU neural networks,” in *Computer Aided Verification*, 2020.

[167] H. Converse, A. Filieri, D. Gopinath, and C. S. Păsăreanu, “Probabilistic symbolic analysis of neural networks,” in *IEEE 31st International Symposium on Software Reliability Engineering*, 2020.

[168] H.-D. Tran, S. Choi, H. Okamoto, B. Hoxha, G. Fainekos, and D. Prokhorov, “Quantitative verification for neural networks using ProbStars,” in *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control*, 2023.

[169] S. Bak and H.-D. Tran, “Neural network compression of ACAS Xu early prototype is unsafe: Closed-loop verification through quantized state backreachability,” in *NASA Formal Methods*, 2022.

[170] D. Manzanas Lopez, T. T. Johnson, S. Bak, H.-D. Tran, and K. L. Hobbs, “Evaluation of neural network verification methods for air-to-air collision avoidance,” *Journal of Air Transportation*, 2023.

[171] K. D. Julian, M. J. Kochenderfer, and M. P. Owen, "Deep neural network compression for aircraft collision avoidance systems," *Journal of Guidance, Control, and Dynamics*, 2019.

[172] M. P. Owen and M. J. Kochenderfer, "Dynamic logic selection for unmanned aircraft separation," in *IEEE/AIAA 35th Digital Avionics Systems Conference*, 2016.

[173] S. Ghosh, Y. V. Pant, H. Ravanbakhsh, and S. A. Seshia, "Counterexample-guided synthesis of perception models and control," 2019. [Online]. Available: https://arxiv.org/abs/1911.01523

[174] C. Hsieh, Y. Li, D. Sun, K. Joshi, S. Misailovic, and S. Mitra, "Verifying controllers with vision-based perception using safe approximate abstractions," in *International Conference on Embedded Software*, 2022.

[175] A. Astorga, C. Hsieh, P. Madhusudan, and S. Mitra, "Perception contracts for safety of ML-enabled systems," in *Proceedings of the ACM on Programming Languages*, no. OOPSLA, 2023.

[176] R. P, "European exascale software initiative EESI2 - towards exascale roadmap implementation," *2nd IS-ENES workshop on high-performance computing for climate models*, 2013.

[177] N. Saxena, "Autonomous car is the new driver for resilent computing and design-for-test," in *NASA Electronic Parts and Packaging (NEPP) Program Electronics Technology Workshop*, 2016.

[178] M. Dimitrov and H. Zhou, "Anomaly-based bug prediction, isolation, and validation: An automated approach for software debugging," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[179] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLFIED: Symbolic program-level fault injection and error detection framework," in *IEEE International Conference on Dependable Systems and Networks*, 2008.

[180] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve, "Minotaur: Adapting software testing techniques for hardware errors," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[181] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002.

[182] O. Goldreich, *Introduction to property testing.* Cambridge University Press, 2017.

[183] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *IEEE International Symposium on Workload Characterization*, 2015.

[184] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[185] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A multiplatform benchmark suite for approximate computing," *IEEE Design Test*, 2017.

[186] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," in *The Web Conference*, 1999.

[187] M. Carbin, D. Kim, S. Misailovic, and M. Rinard, "Verified integrity properties for safe approximate program transformations," in *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2013.

[188] M. Carbin, D. Kim, S. Misailovic, and M. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[189] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, "FLEXJAVA: Language support for safe and modular approximate programming," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[190] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[191] J. R. Sklaroff, "Redundancy management technique for space shuttle computers," *IBM Journal of Research and Development*, 1976.

[192] M. R. Lyu, *Software Fault Tolerance.* John Wiley & Sons, 1995.

[193] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[194] M. Soni, A. Pal, and J. S. Miguel, "As-Is approximate computing," *ACM Transactions on Architecture and Code Optimization*, 2022.

[195] H. Jiang, H. Zhang, X. Tang, V. Govindaraj, J. Sampson, M. T. Kandemir, and D. Zhang, "Fluid: A framework for approximate concurrency via controlled dependency relaxation," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021.

[196] M. Didehban and A. Shrivastava, "NZDC: A compiler technique for near zero silent data corruption," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016.

[197] W. Dweik, M. Annavaram, and M. Dubois, "Reliability-aware exceptions: Tolerating intermittent faults in microprocessor array structures," in *Design, Automation and Test in Europe Conference and Exhibition*, 2014.

[198] M.-L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[199] R. Venkatagiri, K. Ahmed, A. Mahmoud, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve, "gem5-Approxilyzer: An open-source tool for application-level soft error analysis," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2019.

[200] J. Li and Q. Tan, "SmartInjector: Exploiting intelligent fault injection for SDC rate analysis," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, 2013.

[201] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *11th European Dependable Computing Conference*, 2015.

[202] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *IEEE International Symposium on Workload Characterization*, 2015.

[203] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[204] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Hamartia: A fast and accurate error injection framework," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, 2018.

[205] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "Online estimation of architectural vulnerability factor for soft errors," in *International Symposium on Computer Architecture*, 2008.

[206] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in HPC applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[207] G. Papadimitriou and D. Gizopoulos, "Demystifying the system vulnerability stack: Transient fault effects across the layers," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, 2021.

[208] D. G. Cacuci and M. Ionescu-Bujor, "A comparative review of sensitivity and uncertainty analysis of large-scale systems - ii: Statistical methods," *Nuclear Science and Engineering*, 2004.

[209] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving programs robust," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[210] J. Laurel, R. Yang, G. Singh, and S. Misailovic, "A dual number abstraction for static analysis of Clarke Jacobians," *Proceedings of the ACM on Programming Languages*, 2022.

[211] P. Agrawal, "Fault tolerance in multiprocessor systems without dedicated redundancy," *IEEE Transactions on Computers*, 1988.

[212] A. Ziv and J. Bruck, "Performance optimization of checkpointing schemes with task duplication," *IEEE Transactions on Computers*, 1997.

[213] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2016.

[214] K. Miettinen, *A Priori Methods.* Springer US, 1998.

[215] G. Li and K. Pattabiraman, "Modeling input-dependent error propagation in programs," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018.

[216] Y. Yao, "CAVA: Camera vision pipeline on gem5-Aladdin," https://github.com/yaoyuannnn/cava.

[217] A. Mosnier, "SHA-2 algorithm implementations," https://github.com/amosnier/sha-2.

[218] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *16th IEEE International Conference on Program Comprehension*, 2008.

[219] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "ePVF: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[220] Q. Lu, G. Li, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Configurable detection of SDC-causing errors in programs," *ACM Transactions on Embedded Computing Systems*, 2017.

[221] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[222] F. F. d. Santos, J. E. R. Condia, L. Carro, M. S. Reorda, and P. Rech, "Revealing GPU vulnerabilities by combining register-transfer and software-level fault injection," in *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2021.

[223] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain<T>: A first-order type for uncertain data," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[224] J. E. R. Condia, P. Rech, F. F. dos Santos, L. Carrot, and M. S. Reorda, "Protecting GPU microarchitectural vulnerabilities via effective selective hardening," in *IEEE 27th International Symposium on On-Line Testing and Robust System Design*, 2021.

[225] F. Libano, B. Wilson, J. Anderson, M. J. Wirthlin, C. Cazzaniga, C. Frost, and P. Rech, "Selective hardening for neural networks in FPGAs," *IEEE Transactions on Nuclear Science*, 2019.

[226] I. Polian and J. P. Hayes, "Selective hardening: Toward cost-effective error tolerance," *IEEE Design & Test of Computers*, 2011.

[227] C. G. Zoellin, H.-J. Wunderlich, I. Polian, and B. Becker, "Selective hardening in early design steps," in *13th European Test Symposium*, 2008.