CCF-1629431
CCF-1703637
CCF-1846354

HR0011-18-C-0122

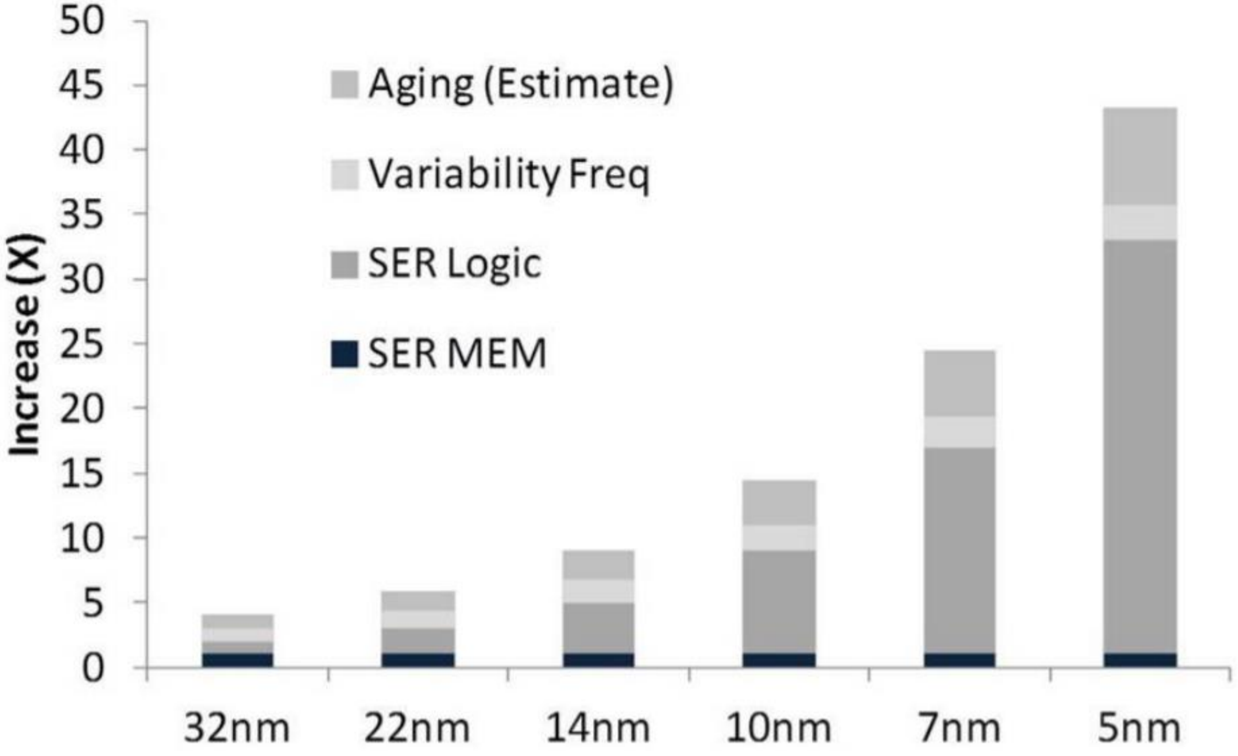
Aloe: Verifying Reliability of Approximate Programs in the Presence of Recovery Mechanisms

Keyur Joshi, Vimuth Fernando, and Sasa Misailovic
University of Illinois at Urbana-Champaign

CGO 2020

ARC UIUC

Unreliable Hardware – Transient Errors



Process size vs. error rate

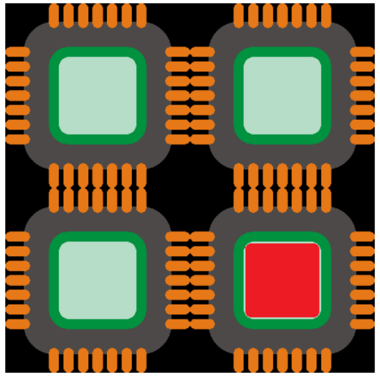
Architects make great efforts to minimize errors

Some errors slip through the cracks – silently corrupt computation results

Image from “Inter-Agency Workshop on HPC Resilience at Extreme Scale”, DoD, ‘12

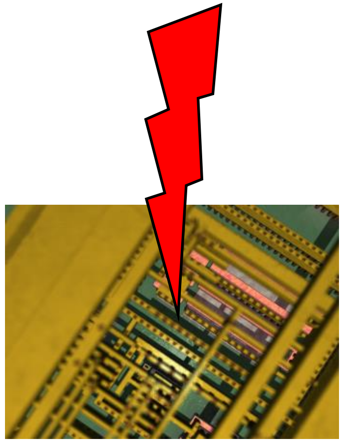
Transient Errors are Everywhere

Big systems
fail due to scale



Heterogeneous systems
have components with
varying reliability

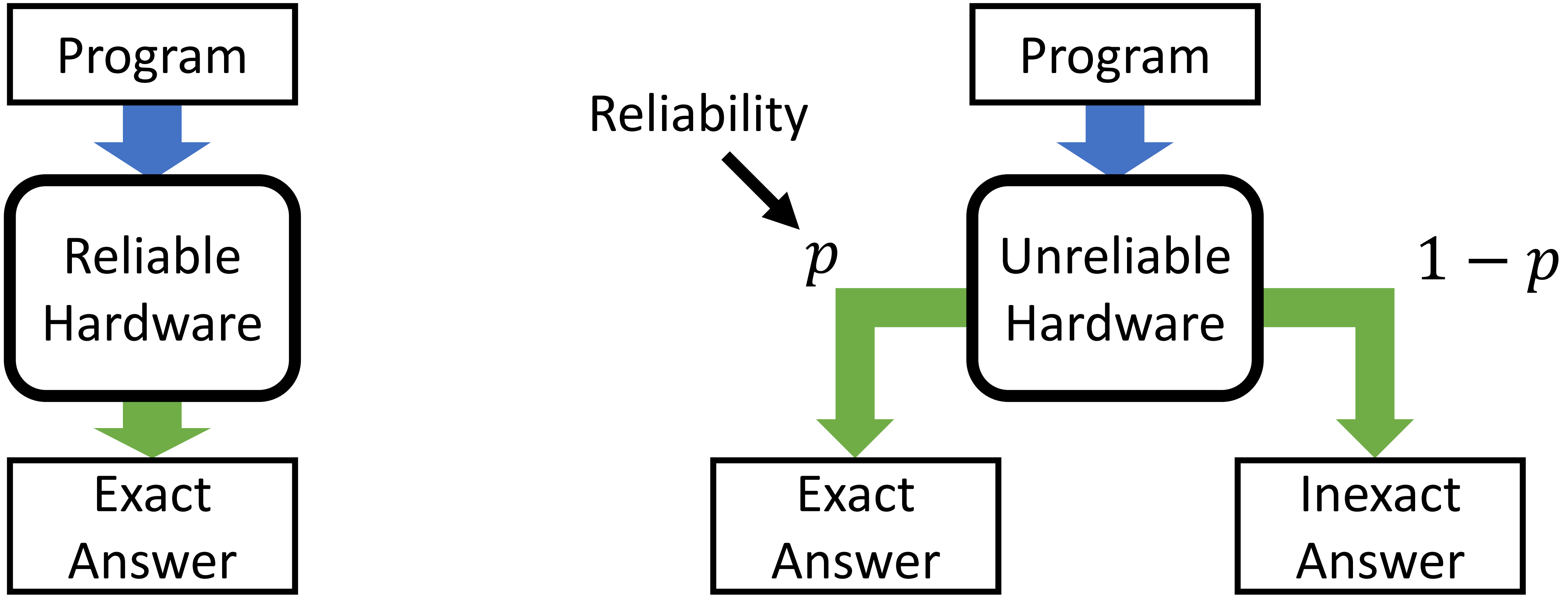
Small systems
fail due to low voltage/power



Rugged environments
radiation, temperature, etc.

Images from Wikipedia and publicdomainvectors.org

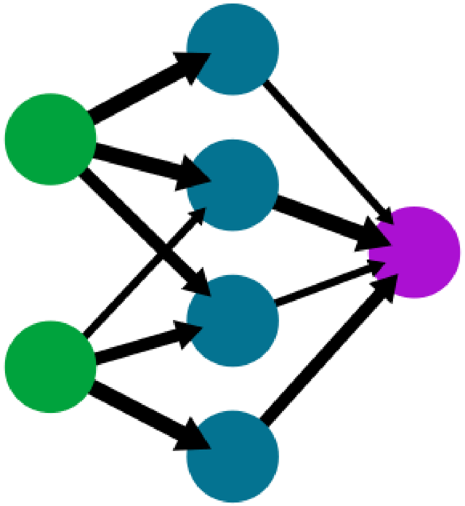
Reliability



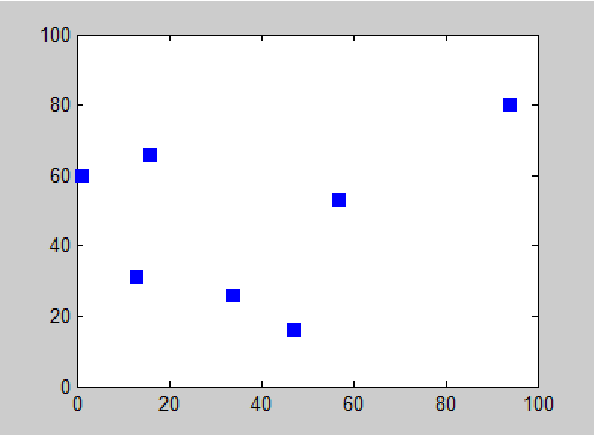
Reliability is the probability of obtaining the *exact* answer

100% Exactness Is **Not** Always Required!

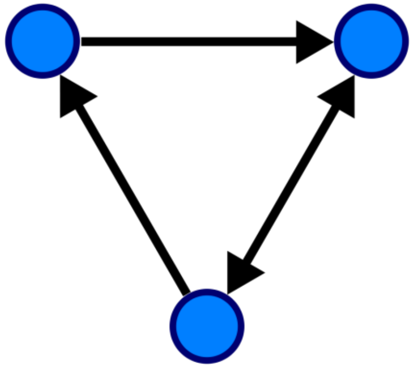
Media Processing



Machine Learning



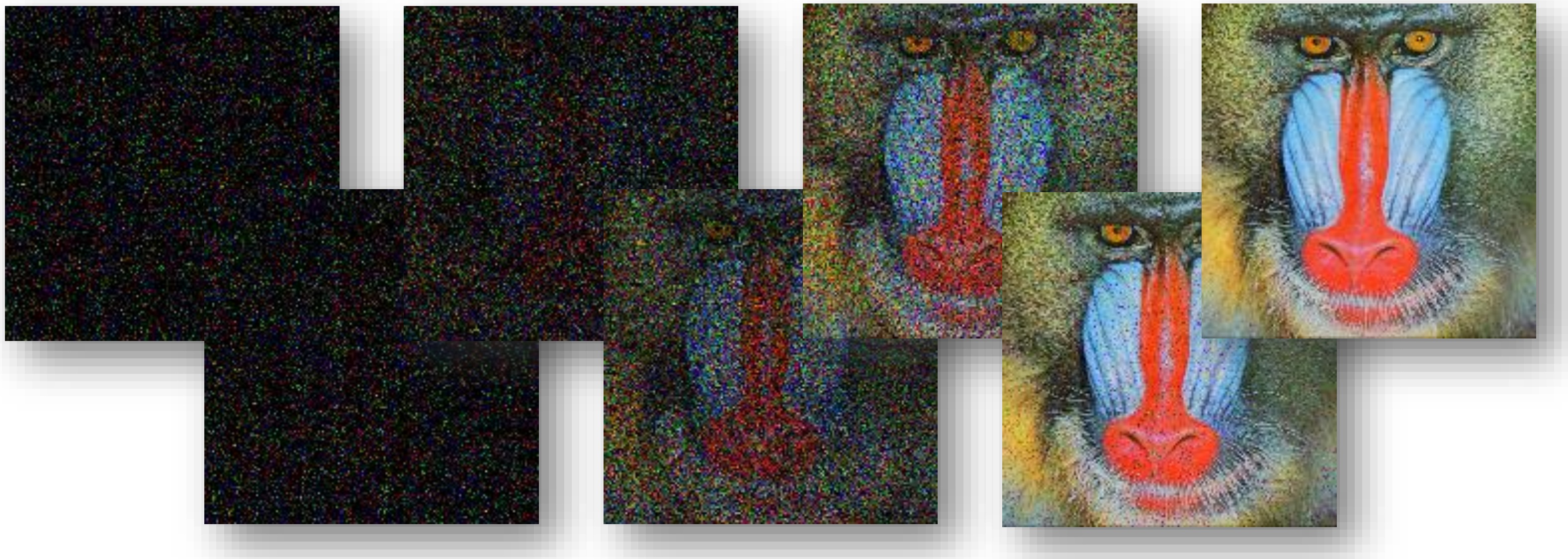
Approximations for NP-Complete Problems



Large-Scale Graph Processing

Images from Wikipedia

But We **Do** Need Quality Control...



% Pixels computed exactly → 20% 40% 60% 80% 90% 99% 99.9%

How do we increase reliability of programs on unreliable hardware?

Lightweight Check and Recover

```
z = x*y
z' = x*y
z==z' ?
```

Code Re-Execution (SWIFT, DRIFT, Shoestring)

```
y = foo(x)
DNN(x,y) ?
```

Anomaly Detection (Topaz, Rumba)

```
y = foo(x)
hw_err_flg ?
```

Hardware Error Flag (Relax)

```
s = SAT(p)
verify(s,p) ?
```

Verification (NP-Complete)

The Try-Check-Recover Mechanism

Some research languages[1,2] expose *Try-Check-Recover mechanisms*:

```
try { solution = SATSolve(problem) }
```
Unreliable code

```
check { satisfies(problem, solution) }
```
Checks for errors

```
recover { solution = SATSolve(problem) }
```
Recovery code

[1]"Relax", M. de Kruijf, S. Nomura, and K. Sankaralingam, ISCA '10

[2]"Topaz", S. Achour and M. Rinard, OOPSLA '15

How do we analyze programs to ensure that they are sufficiently reliable?

Static Reliability Analysis of Programs[1]

Does not contain
try-check-recover

```
output = program(input)
```

Prove:

$$\{\mathcal{R}(\texttt{output}) \geq 0.99 \cdot \mathcal{R}(\texttt{input})\}$$

[1]"Rely", M. Carbin, S. Misailovic, and M. Rinard, OOPSLA '13

How do we do reliability analysis of programs with checks and recovery mechanisms in a formal manner?

Aloe

The first static reliability analysis of programs with recover blocks

Supports recovery blocks that re-execute the `try` computation

Supports arrays, conditionals, and bounded loops

Supports various types of error checkers

Aloe Syntax

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

recovery $\rightarrow$

redo$[n]$	*redo up to n times*
| redo$[\psi]$	*redo on different reliability model*
| S	*other (custom) recovery*

$S \rightarrow$

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

Modelling Unreliable Computations

Aloe <u>models</u> unreliable computations using *probabilistic choice*:

$1 - p$

p

```
var = e_exact [p] e_inexact
```

```
z = x+y [p] rnd() // instruction level[1]
z = foo(x) [p] foo_err(x) // function level[2]
z = 1.0 [p] rnd() // unreliable memory operations[3]
```

[1]"EnerJ", A. Sampson et al., PLDI '11 [2]"Rumba", D. Khudia et al., ISCA '15 [3]"Replica", V. Fernando et al., ASPLOS '19

Hardware Specifications (Example)[1]

	Mild	Medium	Aggressive
DRAM refresh: per-second bit flip probability	10^{-9}	10^{-5}	10^{-3}
Memory power saved	17%	22%	24%
SRAM read upset probability	$10^{-16.7}$	$10^{-7.4}$	10^{-3}
SRAM write failure probability	$10^{-5.59}$	$10^{-4.94}$	10^{-3}
Supply power saved	70%	80%	90%*
`float` mantissa bits	16	8	4
`double` mantissa bits	32	16	8
Energy saved per operation	32%	78%	85%*
Arithmetic timing error probability	10^{-6}	10^{-4}	10^{-2}
Energy saved per operation	12%*	22%	30%

Table 2. Approximation strategies simulated in our evaluation. Numbers marked with * are educated guesses by the authors; the others are taken from the sources described in Section 4.2. Note that all values for the Medium level are taken from the literature.

[1]"EnerJ", A. Sampson et al., PLDI '11

Aloe Reliability Analysis

Aloe's analysis is based on that of Rely[1]

$$\{0.999 \cdot \mathcal{R}(\mathtt{x,y}) \geq 0.99\}$$

Reliability Precondition

```
z = x*y [0.999] rnd();
```

$$\{\mathcal{R}(\mathtt{z}) \geq 0.99\}$$

Reliability Postcondition

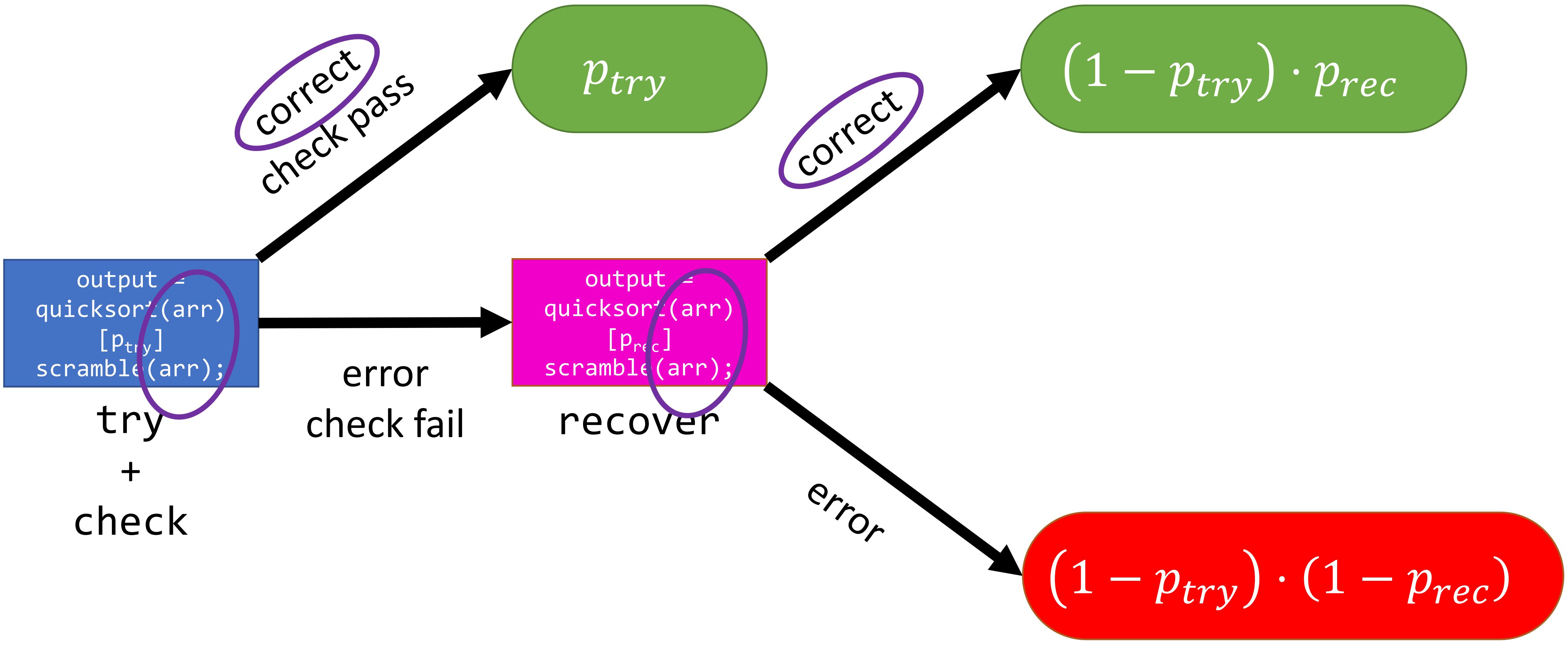
[1] M. Carbin, S. Misailovic, and M. Rinard, OOPSLA '13

Example – Sorting on Unreliable Hardware

```
try {
  output = quicksort(arr) [p_try] scramble(arr);
}
check { sorted(output) }
recover {
  output = quicksort(arr) [p_rec] scramble(arr);
}
```

We want `output` to be correctly sorted with probability $\geq r$

Possible Execution Paths



Aloe Precondition Generation

```
try {
  output = quicksort(arr) [p_try] scramble(arr);
}
check { sorted(output) }
recover {
```

$\{p_{rec} \cdot \mathcal{R}(\text{arr}) \geq r\}$

```
  output = quicksort(arr) [p_rec] scramble(arr);
```

$\{\mathcal{R}(\text{output}) \geq r\}$

```
}
```

$\{\mathcal{R}(\text{output}) \geq r\}$

Detour – Error-Free Rate of `try`

```
try {
   x = y*2 [0.99] rnd();
   z = w+y [0.99] rnd();
} check { f(w,x,y,z) }
```

$$\{0.99 \cdot \mathcal{R}(\mathtt{w},\mathtt{y}) \geq r\}$$

$$\{\mathcal{R}(\mathtt{z}) \geq r\}$$

`check` detects errors in *any* part of `try`

Unreliable computation of x affects the probability that `check` passes!

Aloe separately analyses the probability that `try` executes correctly *in its entirety*

Aloe Precondition Generation

$$\{(p_{try} + (1 - p_{try}) \cdot p_{rec}) \cdot \mathcal{R}(\text{arr}) \geq r\}$$

```
try {
  output = quicksort(arr) [p_try] scramble(arr);
}
check { sorted(output) }
recover {
```

$$\{p_{rec} \cdot \mathcal{R}(\text{arr}) \geq r\}$$

```
  output = quicksort(arr) [p_rec] scramble(arr);
```

$$\{\mathcal{R}(\text{output}) \geq r\}$$

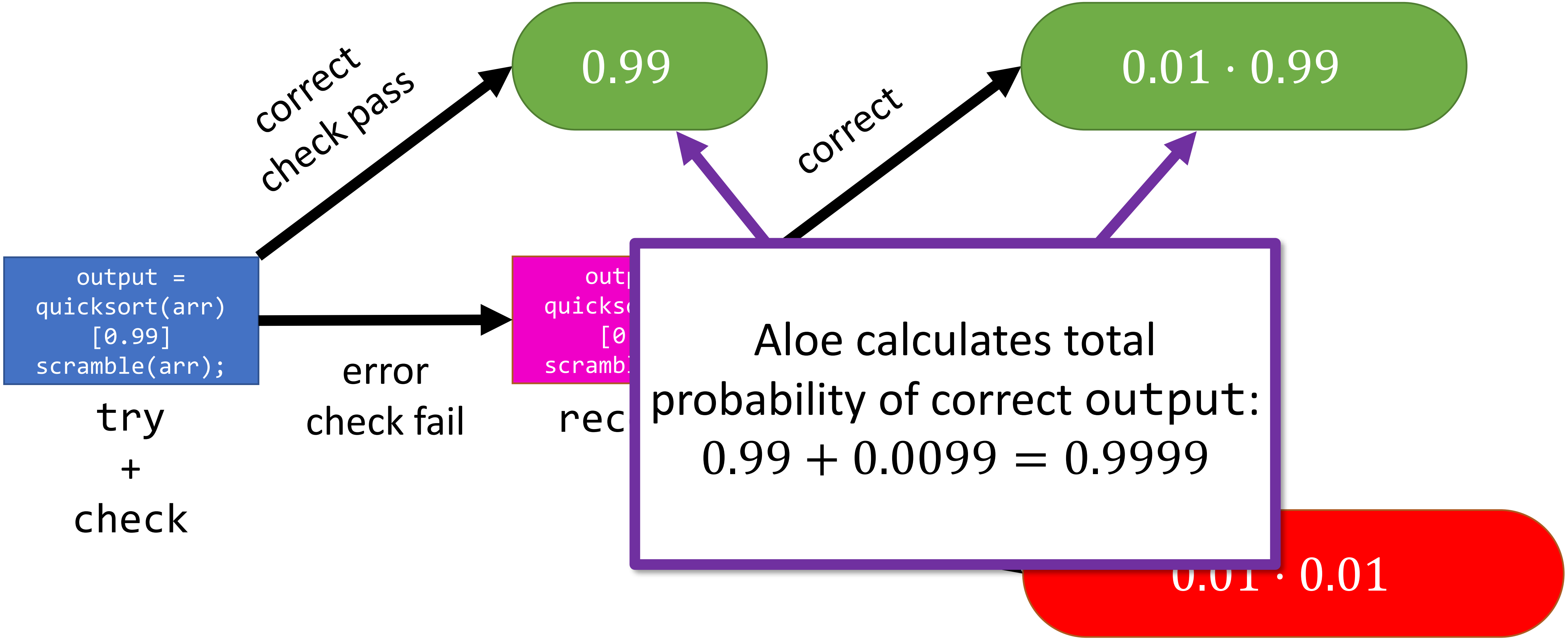
```
}
```

$$\{\mathcal{R}(\text{output}) \geq r\}$$

Error-free rate of `try`:

p_{try}

Possible Execution Paths ($p_{try} = p_{rec} = 0.99$)



Combining Preconditions

```
recover {
```

$$\{0.99 \cdot \mathcal{R}(\mathtt{w,y,z}) \geq r\}$$

$$\{0.99 \cdot \mathcal{R}(\mathtt{w,y}) \geq r \quad \wedge \quad 0.999 \cdot \mathcal{R}(\mathtt{y,z}) \geq r\}$$

```
  if (*) {
    x = y*w [0.99] rnd();
  } else {
    x = y+z [0.999] rnd();
  }
```

$$\{\mathcal{R}(\mathtt{x}) \geq r\}$$

```
}
```

Complex Postconditions

$$\{0.9999 \cdot p_1 \cdot \mathcal{R}(\mathrm{y},\mathrm{z}) \geq r_1 \;\wedge\; p_2 \cdot \mathcal{R}(\mathrm{y}) \geq r_2\}$$

```
try {
   x = y*z [0.99] rnd();
}
check { f(x,y,z) }
recover {
   x = y*z [0.99] rnd();
}
```

Unmodified

$$\{p_1 \cdot \mathcal{R}(\mathrm{x}) \geq r_1 \;\wedge\; p_2 \cdot \mathcal{R}(\mathrm{y}) \geq r_2\}$$

Aloe Assumptions – Re-execution

Aloe expects that `recover` re-executes the code in `try`

The reliability of statements in `try` and `recover` may differ

Why? Impossible to prove using Rely’s logic that `try` and `recover` perform the same computation

If such a proof is already available, then Aloe’s analysis remains valid even for syntactically distinct `try` and `recover`

Aloe Assumptions – Idempotence

Aloe expects that the computation in `try` is *idempotent*

Idempotent – can be run multiple times without changing the correct result

E.g. `x=y+z` ✓ `x=x+z` ✗

Why? Otherwise `try` can modify the result of executing `recover`

Handling Control Flow – Same as in Rely

$$\boxed{RP_\psi(\texttt{if}_\ell\ \ell\ s_1\ s_2, Q) \quad = \quad RP_\psi(s_1, Q) \wedge RP_\psi(s_2, Q)}$$

$$\begin{array}{lcl} RP_\psi(\texttt{while}_\ell\ b : 0\ s, Q) & = & Q \\ RP_\psi(\texttt{while}_\ell\ b : n\ s, Q) & = & RP_\psi(\mathcal{T}(\texttt{if}_{\ell_n}\ b\ \{s\ ;\ \texttt{while}_\ell\ b : (n-1)\ s\}\ \texttt{skip}), Q) \end{array}$$

Rely Precondition Generation for Control Flow

Using If-Then for Recovery Mechanisms

Prior analyses (Rely) expressed recovery mechanisms using if-then statements

```
output = quicksort(list) [p_try] scramble(list);
if ( ! sorted(output) )
{
  output = quicksort(list) [p_rec] scramble(list);
}
```

Using If-Then for Recovery Mechanisms

Rely treats if-then as a nondeterministic choice

Case 1:

output = quicksort(list) [p_{try}] scramble(list);

Case 2:

~~output = quicksort(list) [p_{try}] scramble(list);~~

output = quicksort(list) [p_{rec}] scramble(list);

Using If-Then for Recovery Mechanisms

Rely analyses the reliability of each case separately

Case 1: `output` sorted correctly with probability p_{try}

```
output = quicksort(list) [p_try] scramble(list);
```

Case 2: `output` sorted correctly with probability p_{rec}

~~`output = quicksort(list) [p_try] scramble(list);`~~

```
output = quicksort(list) [p_rec] scramble(list);
```

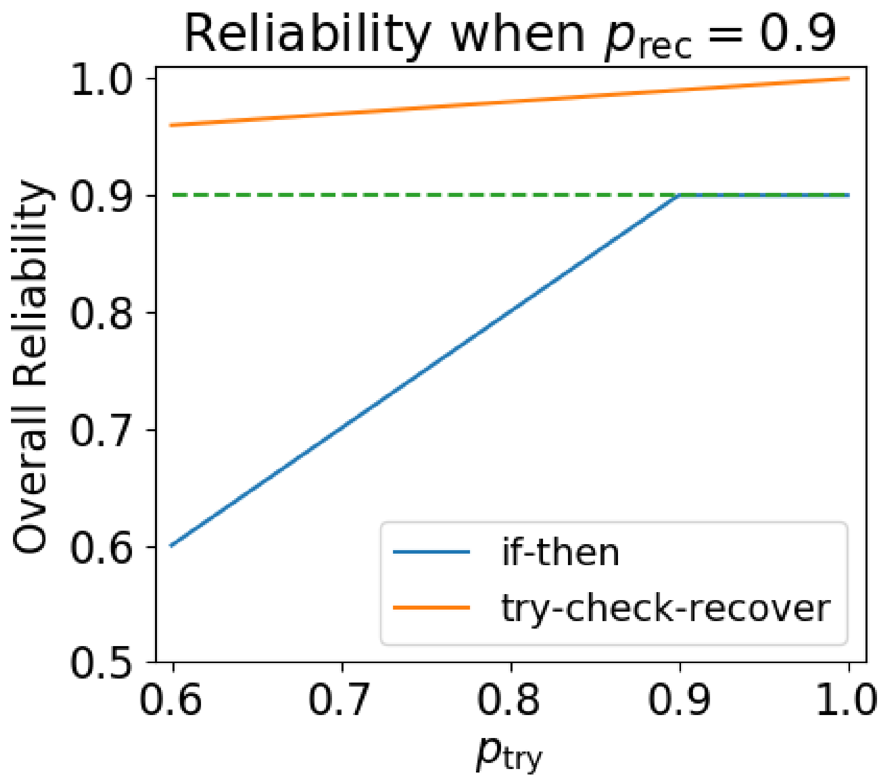
Using If-Then for Recovery Mechanisms

Rely then retains the most conservative case

Overall reliability: $\min(p_{try}, p_{rec})$

Compare to Aloe's calculated reliability using try-check-recover:

$$p_{try} + (1 - p_{try}) \cdot p_{rec}$$



Imperfect Checkers

Many checkers are imperfect – may not precisely detect errors

Code re-execution and comparison

- "SWIFT", G. Reis et al., CGO '05
- "Shoestring", S. Feng et al., ASPLOS '10

May detect nonexistent errors

Error Prediction

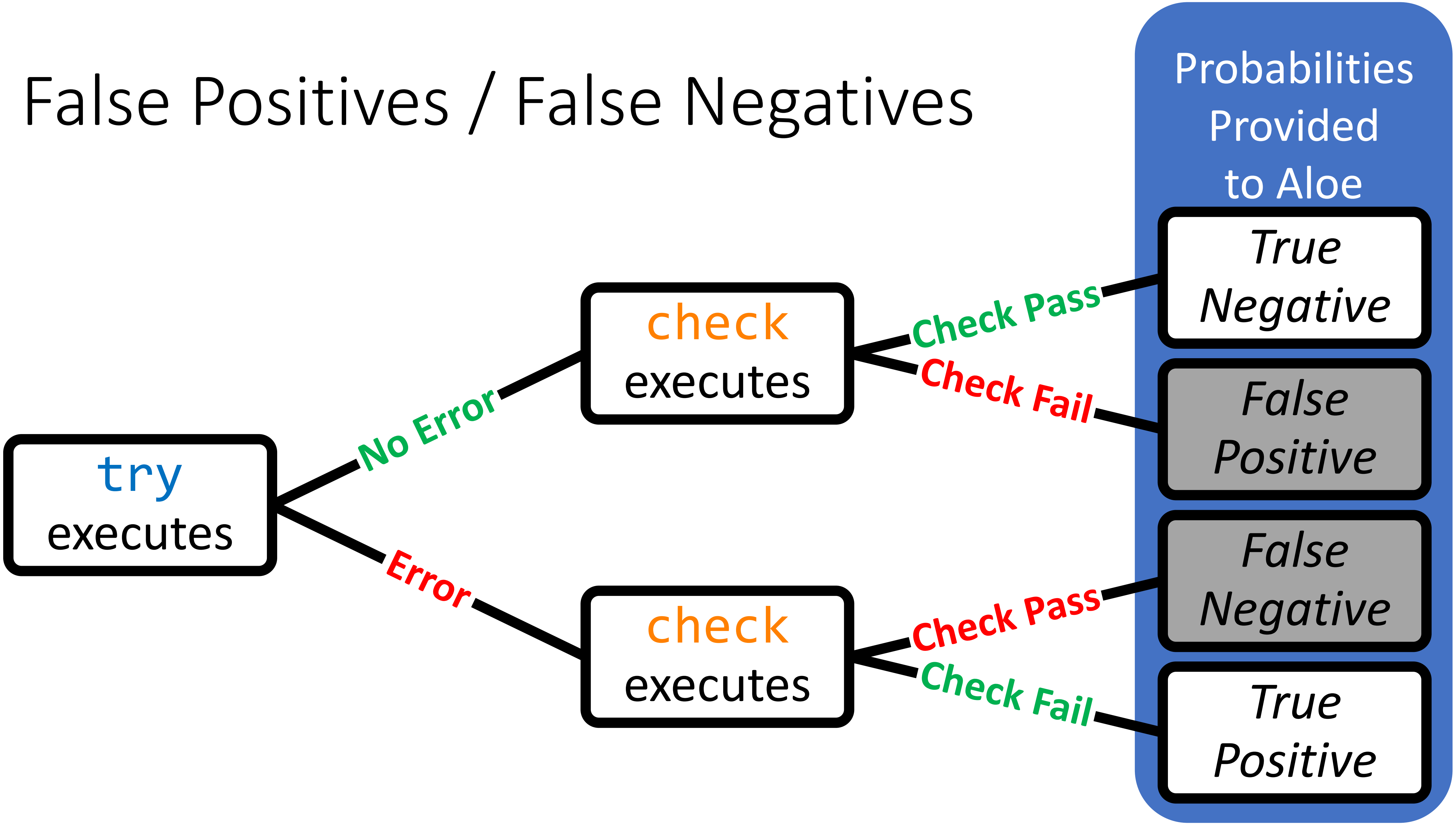
- "Rumba", D. Khudia et al., ISCA '15

Anomaly detection

- "Topaz", S. Achour and M. Rinard, OOPSLA '15

May not detect actual errors, may detect nonexistent errors

False Positives / False Negatives



False Positive / False Negative Rates

For some checkers, these rates can be determined analytically

- E.g. approximate sorted-ness checks provide statistical guarantees

For other checkers, these rates must be determined empirically

- E.g. outlier detection[1], DNNs[2] which require pre-training
- Probabilities of false positives/negatives are estimated from training/testing data
- Aloe's analysis is only valid for similar distribution of input data

[1]"Topaz", S. Achour and M. Rinard, OOPSLA '15

[2]"Approximate Checkers", A. Mahmoud et al., WAX '19

Example – Unreliable Multiplier Hardware

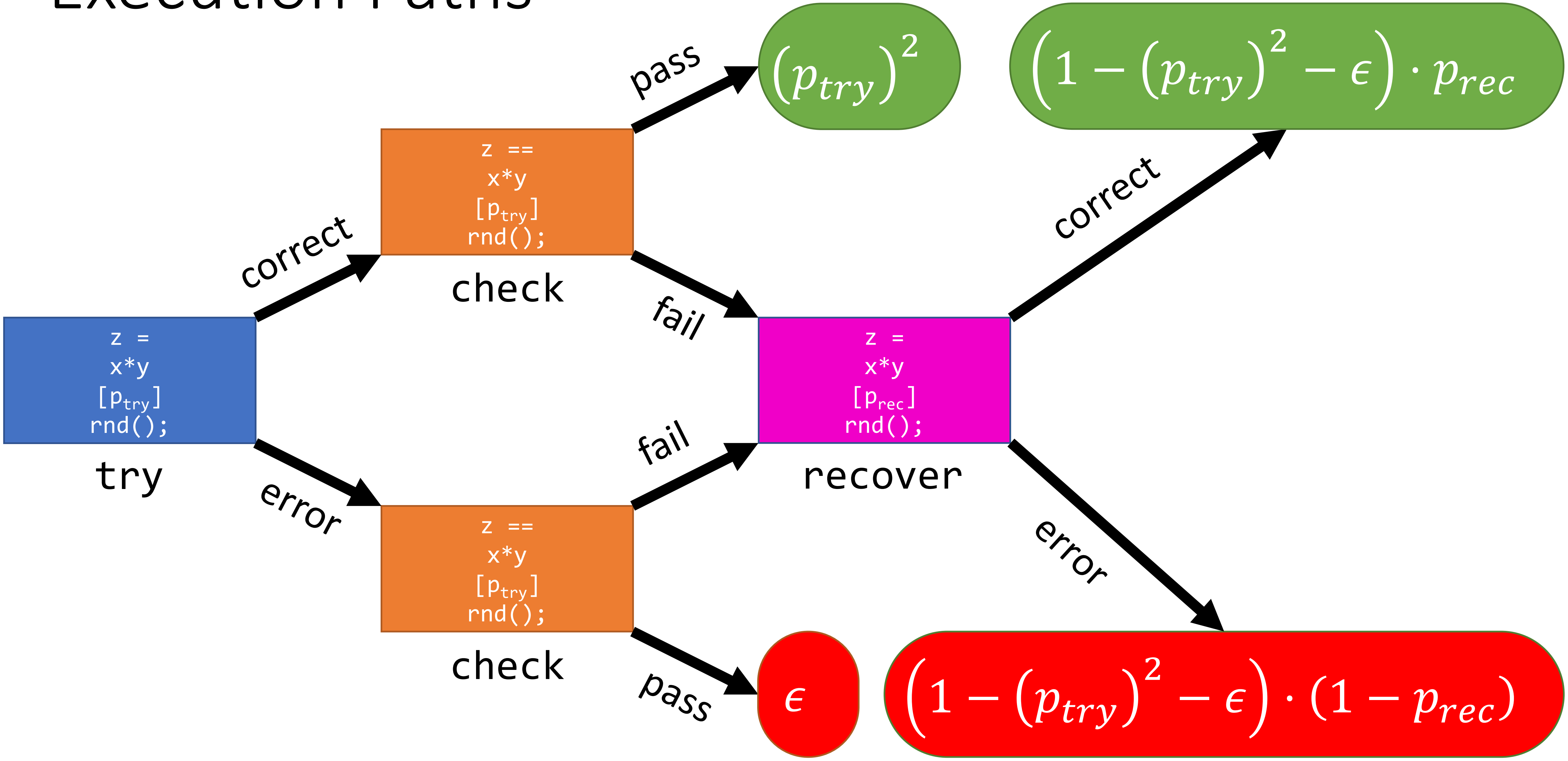
```
try {
   z = x*y [p_try] rnd();
}
check {
   z == (x*y [p_try] rnd());
}
recover {
   z = x*y [p_rec] rnd();
}
```

`try` multiplies x and y in an unreliable manner

`check` re-executes the computation on same hardware

We want z to be exact with probability $\geq r$

Execution Paths



Aloe Precondition Generation

Error-free rate of `try`:

p_{try}

$$\left\{\left(\left(p_{try}\right)^2 + \left(1 - \left(p_{try}\right)^2 - \epsilon\right) \cdot p_{rec}\right) \cdot \mathcal{R}(\mathrm{x},\mathrm{y}) \geq r\right\}$$

```
try {
  z  =  x*y [p_try] rnd();
}
check { z == (x*y [p_try] rnd()); }
recover {
```

$$\{p_{rec} \cdot \mathcal{R}(\mathrm{x},\mathrm{y}) \geq r\}$$

```
  z = x*y [p_rec] rnd();
}
```

$$\{\mathcal{R}(\mathrm{z}) \geq r\}$$

True Negative:

p_{try}

False Positive:

$1 - p_{try}$

False Negative:

$\epsilon \ (\approx 0)$

True Positive:

$1 - \epsilon$

Benchmarks

try-check-recover

Benchmark	End-to-End Computation	Kernel Computation
PageRank	PageRanks of graph nodes	Update PageRank of one node
Scale	Upscale an image	One pixel of upscaled image
Blackscholes	Prices of stock options	Price of one stock option
SSSP	Single Source Shortest Path	One iteration for one node
BFS	Breadth First Search	One search iteration for one node
SOR	Successive Over-Relaxation	One update for one element
Motion	Motion estimation	Similarity calculation for one block
Sobel	Edge detection filter	One pixel of filtered image

Methodology

We model an architecture having multiple available reliability levels[1]

Reliability of arithmetic operations:

`try` – 0.999 [1]

`recover` – 0.9999 [1]

[1] "EnerJ", A. Sampson et al., PLDI '11

Methodology

Perfect checkers: we simulate hardware support for detecting errors[1,2]

Imperfect checkers: we experiment with different false positive/negative rates from Topaz[3]

We compare Aloe's analysis results to Rely

Rely uses if-then instead of try-check-recover

[1]"Relax", M. de Kruijf et al., ISCA '10 [2]"Argus", A. Meixner et al., MICRO '07 [3]S. Achour and M. Rinard, OOPSLA '15

Reliability Calculated by Aloe (Perfect Checker)

	Kernel-level Reliability		End-to-End Reliability		
Benchmark	Aloe	Rely	Aloe	Rely	Aloe Time
PageRank	0.9999	0.9531	≥ 0.99	≈ 0.00	23.33s
Scale	0.9999	0.9891	≥ 0.99	≈ 0.00	10.48s
Blackscholes	0.9999	0.9871	≥ 0.99	≈ 0.00	6.51s
SSSP	0.999999	0.9920	≥ 0.99	≈ 0.00	18.60s
BFS	0.99999	0.9227	≥ 0.99	≈ 0.00	15.22s
SOR	0.99999	0.9950	≥ 0.99	≈ 0.00	21.02s
Motion	0.9999	≈ 0.00	≥ 0.99	≈ 0.00	4.42s
Sobel	0.9999	0.9930	≥ 0.99	≈ 0.00	2.10s

More in the Paper

- error-free rate analysis of `try`
- Several additional examples
- Additional evaluation details
  - Testing setup
  - Unreliable checker and empirical analysis results
- [Appendix] Semantics and Aloe soundness proof

Conclusion

Aloe is the first static analysis of reliability of programs with recovery mechanisms

We analyzed eight kernels and end-to-end benchmarks with recovery mechanisms

Aloe can verify useful reliability bounds for all benchmarks