On Undecidability Aspects of Resilient Computations and Implications to Exascale

Nagi Rao

(Nageswara S. V. Rao)

Oak Ridge National Laboratory

7th Workshop on

Resiliency in High Performance Computing in Clusters, Clouds, and Grids (Resilience2014)

August 25, 2014, Porto, Portugal

At 20th International European Conference on Parallel and Distributed Computing (Euro-Par)

Research Sponsored by

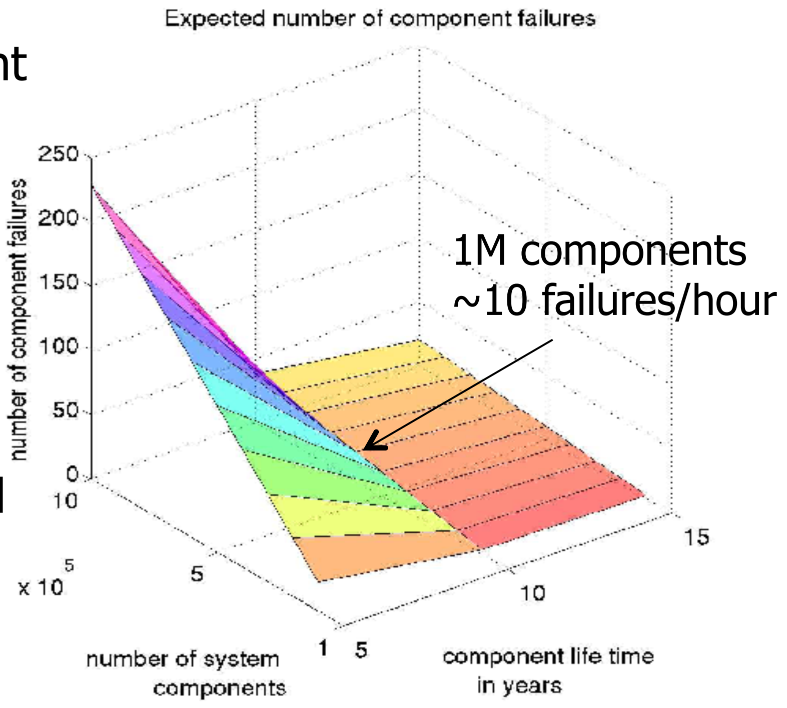
Applied Mathematics Program, U.S. Department of Energy

Outline

1. Introduction
2. Context of Undecidability
3. Complex Faults in Exascale Systems
4. Decision Problems on Exascale Computations
5. Undecidability of Resilient Computations
6. Resilient Computations and Halting Problem
7. Proofs of Resilient Computations
8. Conclusions

Inherent Failures in Exascale Computing Systems

- Exascale computing systems are expected to have millions of processor cores and other components.
  - components with expected life-span of ten years
    - ~100k hours/component = 10 failures/hour among 1M components
  - codes that run for a few hours likely experience failures of several components.
- Failure rates limit the effectiveness of current check-point/recovery methods:
  - Recovery times could be hours for Exascale systems
  - transient silent errors may lead to erroneous computations
- Failures will be integral part of Exascale computations – must be explicitly accounted
  - code outputs must be quantified with confidence estimates
    - specific to system failure profile
    - justifiable by measurements



Related Areas: Resilient Computations

- Foundational works:
  - von Neumann studied (in 1950s) mathematical aspects of achieving reliable computations over systems with unreliable components
  - subsequent reliability improvements in computing systems, perhaps, led to such studies not being extensively continued
- Deployed systems: computing systems in satellites
  - deployed over past decades - enhanced with Software-Implemented Hardware Fault Tolerance (SIHFT) methods to counteract errors due to radiation in space environments.

But, Exascale computations present new challenges:

- sheer size and system complexity makes dynamic profiling of the failures and robustness complicated
- computation becomes inherently probabilistic:
  - for most applications, 100% guarantee of robustness against failures in not possible
  - requires confidence measures for code outputs – running to completion is not sufficient

Faults and Computations

Fault-Free Computing System: Program P with input x is executed error free produces output $\phi_P(x) = \Phi(x, P)$

$\uparrow$ universal function or Turing Machine

P is also represented by its Godel's number $e = \# P$

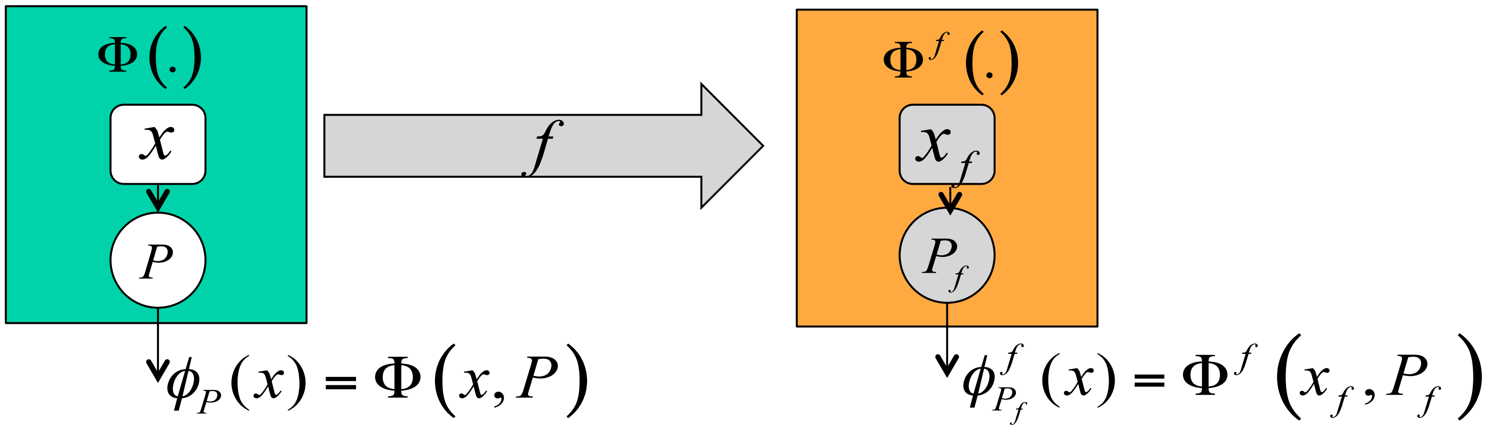
Fault: Program may crash or produce incorrect output or loop forever

Simple fault models: specified by function f

data error: x replaced by x_f

program or code error: P replaced by P_f

execution error: $\Phi(.)$ replaced by $\Phi^f(.)$



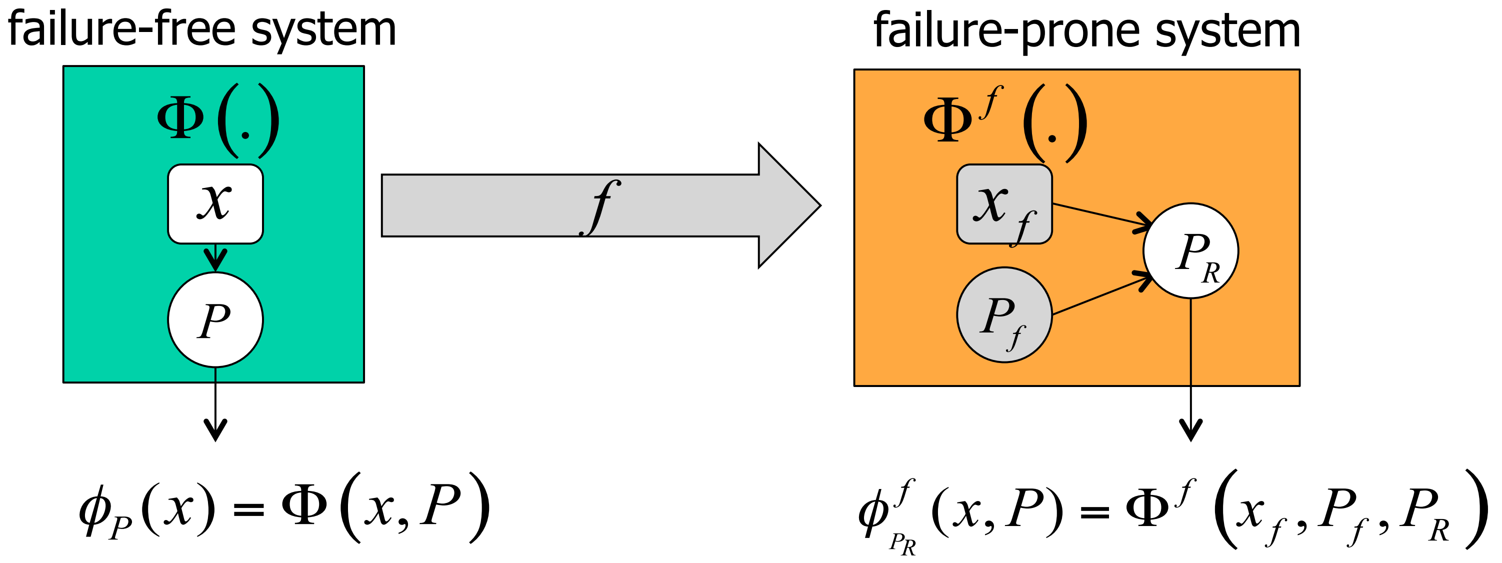
Simplified models: specified by separate functions $f = (f_D, f_P, f_E)$

$f_D : x \rightarrow x_f \qquad f_P : P \rightarrow P_f \qquad f_E : \Phi \rightarrow \Phi_f$

Faults may be more complicated: stochastic, correlated, multi-valued

Faults and Resilient Computations

Resilient Version of Program P: Another program P_R on fault-prone system with same output and halting properties as P on failure-free system



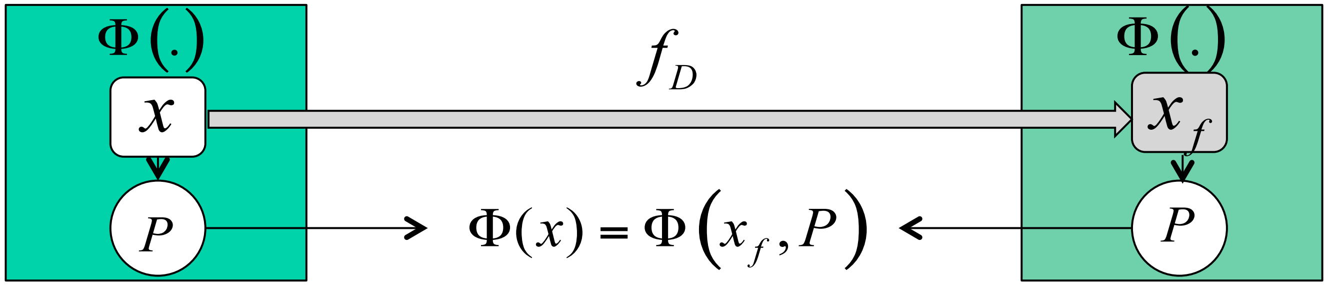
Requirement: Resilient version P_R of the program P is required to produce on the failure-prone system $\Phi^f(.)$ same output as original program on the failure-free systems: that is,

$$\phi_P(x) = \phi_{P_R}^f(x, P)$$

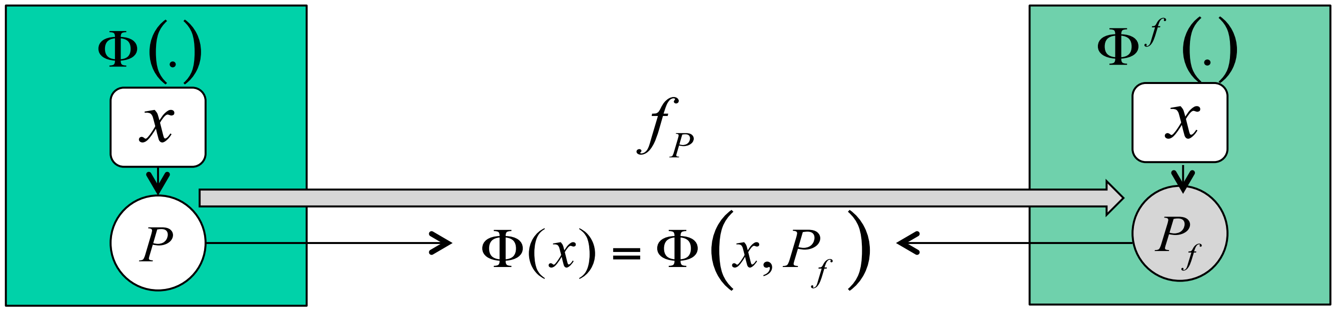
$$\Phi(x, P) = \Phi^f(x_f, P_f, P_R)$$

Individual Data, Program and Executional Faults

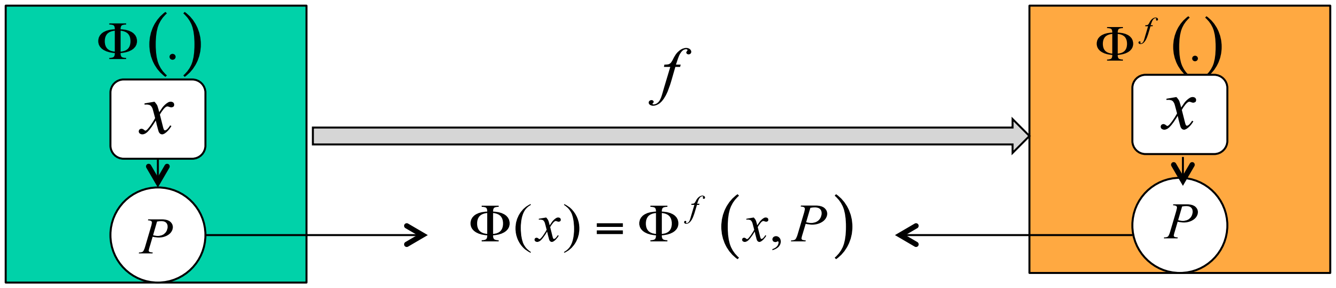
(a) Data faults: $f_D : x \to x_f$



(b) Program faults: $f_P : P \to P_f$



(c) Executional Faults: $f_E : \Phi \to \Phi_f$



Context of Undecidability

Turing's Undecidability or Uncomputability: Non-existence of algorithms for certain class of computing problems

- These are "harder" than NP-hard problems
- Examples: halting problem, loop detection problem, equivalence of context-free grammar, virus detection problems
- **Resilient Computations**: Certain resilient computational problems turn out to be undecidable

Godel's Undecidability: Non-existence of mathematical proofs for true statements within an axiomatic system such as arithmetic system with multiplication

- Arithmetic systems with addition and multiplication operations have statements that are true but cannot be proved as theorems
- **Resilient Computations**: Certain resilience properties of computer codes cannot be proved

These two results are closely related: algorithmic information theory of Gregory Chaitin

Resilient computations: certain faults may embody information beyond the capacity of axiomatic system or finite computations

- not be compensated purely by mathematical or computing means
- but, may be handled by other means, such as fault monitors

Complex Faults in Exascale Systems

1. **Code Corruption**: Executables may be corrupted and lead to infinite loops
   - condition $i < N$ may be changed to $i > 0$
   - go to statements changed to be self-referential
2. **Errors in Parameters and Variables**: infinite loops can be created by corrupted variable values
   - changing boundary values of loop variables
3. **ALU Circuit Errors**: arithmetic and logic execution errors can create infinite loops
4. **Program Counter Errors**: loading errors in program counters will lead computations directed to unexpected locations

Composite Errors: Hot-spots in server cabinets lead to overheating – that effects multiple parts of the system

Language L and Turing machines: codes and data both appear as strings

memory errors effect them both

Resilient Computations

Under Programming Language L

Program P is consists of instructions

Converted to numerical code $y = \#(P)$ using Godel numbering

$\phi_y(x)$ is output of program

Universality Theorem: there exists a universal function $\Phi(.)$ such that

$$\Phi(x, y) = \phi_y(x)$$

Resilient version: y_R is resilient version of y

Functions computed on fault-prone machine: with input x

original non-resilient program: $g(x)$

resilient-version: $g_R(x)$

Set of original programs: A

Examples: non-linear solvers, climate codes, …

All their resilient versions: $R = \left\{ y_R \mid y \in A \right\}$

index set:

$$R_R = \left\{ t \in N \mid \phi_t \in R \right\}$$

Undecidability of Verification of Resilient Computations

Resilience Property is not verifiable computationally:

index-set $R_R = \{t \in N \mid \phi_t \in R\}$ is not computable – **Theorem 1**

Proof outline: By contradiction – assume it is computable

$$h(t,x) = \begin{cases} g(x) & if \quad t \in R_R \\ g_R(x) & if \quad t \notin R_R \end{cases}$$

If R_R is computable, then the following function is partially computable

$$h(t,x) = 1_{R_R}(t) g(x) + \left[1 - 1_{R_R}(t)\right] g_R(x)$$

where $1_S(.)$ is the indicator function of set S such that

$$1_S(x) = \begin{cases} 1 & if \quad x \in S \\ 0 & if \quad x \notin S \end{cases}$$

By recursion theorem, there exists a program *e* such that

$$\phi_e(x) = h(e,x) = \begin{cases} g(x) & if \quad \phi_e \in R \\ g_R(x) & if \quad \phi_e \notin R \end{cases}$$

Undecidability of Verification of Resilient Computations

Proof outline continued: by contradiction

By universality theorem, there exists a function $\Phi(.,.)$ such that

$$\Phi(x, e) = \phi_e(x)$$

Question: which set does *e* that computes $h(e, x)$ belongs to?

Answer is neither as shown below

Case A: it is resilient code $e \in R_R$

implies on error-prone system: by definition of *h(.)*

$$\phi_e(x) = h(e, x) = g(x)$$

but its function is same as original non-resilient $g(x)$

Case B: it is non-resilient code $e \notin R_R$

implies on error-prone system: by definition of *h(.)*

$$\phi_e(x) = h(e, x) = g_R(x)$$

but its function is same as resilient version

Resilient Computations and Halting Problem

Halting Problem: HALT(x, y) is true if and only if y halts on input x

- it is well-known to be undecidable

Loop Detection Problem: NO-LOOP(x, y) is true if and only if y does not loop indefinitely on input x

Fault Function: $f(x, y) = (x_f, y_f)$

replace both data and code by faulty versions

- it is only a special abstracted case of faults

Resilience Versions: RESILIFY(x, y, f) is true if and only if there exists a program P_f for failure-prone system that:

takes input x_f and produces the same output as y with input x on failure-free system

Uncomputability of RESILIFY(x, y, f)

Loop Creating Faults:

RESILIFY(x, y, f) is not a computable predicate – Theorem 2
Proof outline: Equivalent to RESILIFY(x_f, y_f, I) where I is the Identity function that creates no faults

input and code are changed to x_f and y_f instantaneously

It is equivalent to NO-LOOP(x_f, y_f), which is non-computable:

Consider the program:

```
[P]: if  NO-LOOP( $x, x$ ) go to  P
         else return NO
```

index of the code $y_0 = \#P$

NO-LOOP(x, y_0) is true if and only if NO-LOOP(x, x) is false

Using $x = y_0$ we have contradiction

NO-LOOP(y_0, y_0) is true if and only if NO-LOOP(y_0, y_0) is false

Resilient Computations on Turing Machines

A Turing Machine consists of:

- infinite tape of cells – read/write head that process them one at a time, and moves left or right or no motion
- Finite set of states that control the computation

TM *M* starts with input *w* on the tape, and goes through state transitions and writes output on to the tape

Same computation is done by Universal TM (UTM) by specifying both *M* and *w* as input on tape

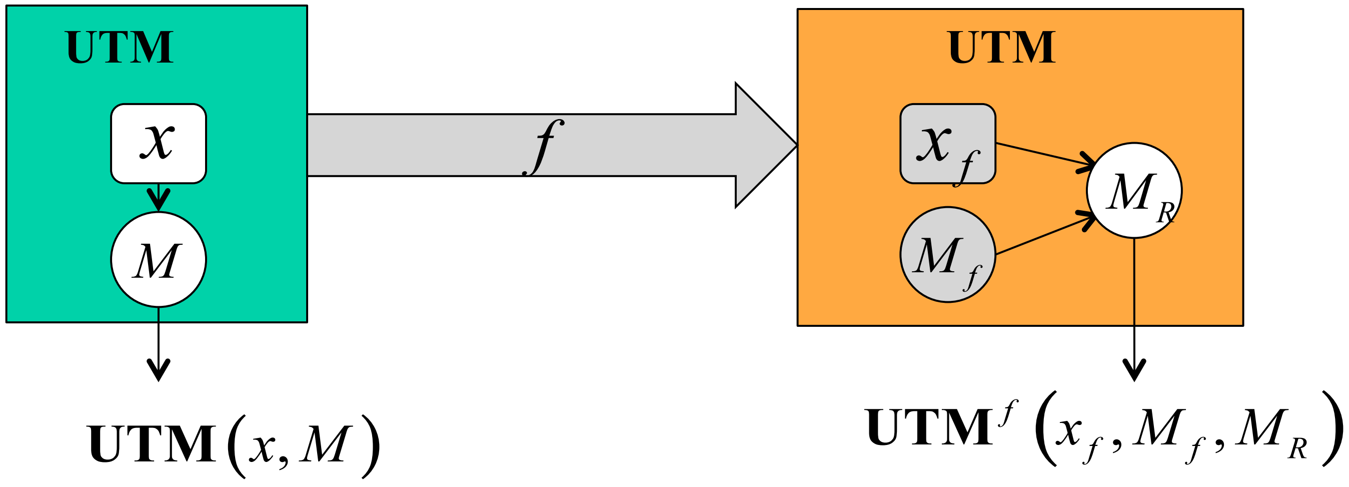
Faults are more "concretely" visualized electro-mechanically on TM:

- **memory faults**: contents of cell may be corrupted
- **code errors**: states may be corrupted
  - more easily see if code is written on tape and executed by UTM
- **execution errors**:
  - state transition faults – analogous to program counter errors
  - read/write head errors – analogous to memory to register transfer errors

Failure-prone version of M is denoted by M_f under fault function f

Faults and Resilient Computations on TM

Resilient Version of TM M: Another TM M_R on fault-prone system with same output and halting properties as M on failure-free system



Requirement: Resilient version M_R of TM M is required to produce under the failure conditions the same output as the original failure-free TM:

$$\mathbf{UTM}(x, M) = \mathbf{UTM}^f(x_f, M_f, M_R)$$

Resilient Computations on Turing Machines

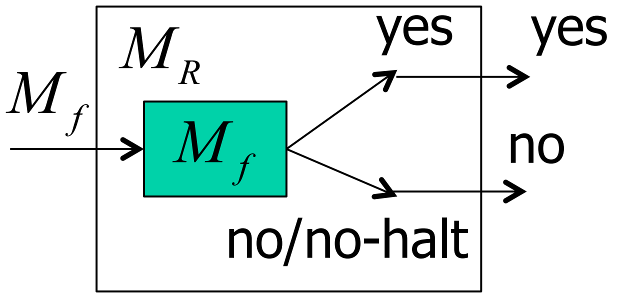
Resilient Turing Machine: M_R operates under fault function f

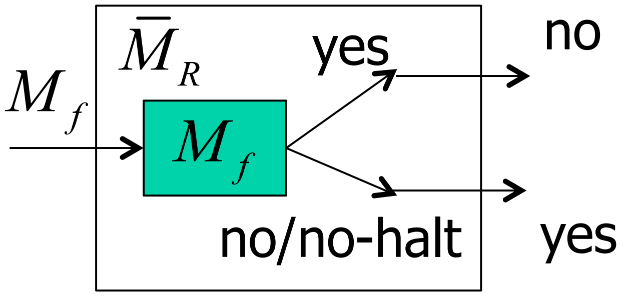
- RTM takes as input TM M and its input w
- halts and produces same output as M when executed by failure-free UTM with input w

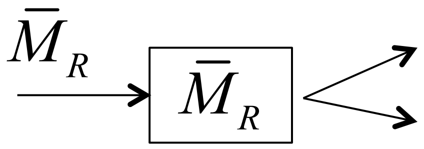
Resilient Turing Machines do not exist under executional and data failures – Theorem 3

Proof Outline: If M_R exists: it detects when M_f does not halt, and interject and produces failure-free output of M

Halting problem:







Yes => $\bar{M}_R$ outputs no or does not halt

No => $\bar{M}_R$ outputs yes

Difficulties of Mathematical of Resilience Proofs

Simple Computing Example: Compute $G(x,a) = x^a$

Computations are limited to multiplications, additions and these functions – all are integer computations

Computing Errors: limit magnitude of operand to $x_f < x$

– zeroing of higher-order bits in registers: it computes $G(x_f, a)$

Potential Solution: program that makes up the difference by computing and adding the "rest" of the value such that:

$$G(x,a) = G(x_f, a) + G(y,a)$$

Difficulty: Proof of this resilience property requires us to contradict Fermat's last theorem that claims no solution exists for $a > 2$:

$$x^a = x_f^{\ a} + y^a$$

Proof has remained open for 300 years until 1995 by Andrew Wilks

Illustration of difficulties in proving a rather "simple" resilience properties

Beyond Halting Problem: Relativized Computations

Foundational Question: Does solving the Halting problem give us complete resilience?

Short answer: most likely NO – certain complex faults may remain that still lead to undecidable problems

Relativized Complexity Classes: Model Halting Problem solution as an oracle to abstract away the complexity:

> Relativized Rice Theorem establishes that assessing general resilience properties of computations remains undecidable under the Halting Problem Oracle.

Resilient Computations:

- Halting problems address classes of problems related to creation of infinite loops
- Ensuring that outputs of resilient versions are identical to output of original codes poses additional challenges somewhat analogous to checking equality of context-free grammars:
  - yet another class of undecidable problems

Conclusions

Addressed computational aspects of resilient computations under broad class of faults

Resilient computations present significant computational challenges:

(a) asserting resiliency of computations is non-computable

(b) mathematical proofs of resilience of algorithms are undecidable

These problems are not solvable in general form by computations and mathematical proofs alone: but,

- resilient computations can be designed for specific classes
- additional fault detection methods could make some problems computable

In general, these results motivate: deeper investigations of fault classes and resilient computations customized for them with complementary information

Future Work:

- These results are only a very first step
- Deeper study of fault classes in needed
  - impact on computations and resilience mechanisms
- Study of hierarchy of undecidable problems and corresponding faults (vice versa):
  - is there a taxonomy of faults that reflects this hierarchy
    - seems reasonable to expect: some fault are harder to protect against

Thank you