Inference and Enforcement of Data Structure Consistency Specifications

Brian Demsky*, Michael D. Ernst, Philip J. Guo,
Stephen McCamant, Jeff H. Perkins, and Martin Rinard

MIT and *UC Irvine

Data structure corruption and repair

- Corrupt data structures often cause crashes or other unacceptable execution
- Repair returns structures to a consistent state, allowing continued execution
- When applicable, can significantly improve availability

When is repair appropriate?

- An inconsistency is discovered: abort, continue, or repair and continue?
- Repair is appropriate when:
  - Downtime/reboot unacceptable, separable computations
  - E.g, persistent data, CTAS
- Repair is inappropriate when:
  - Reboot easy and fast, imperfect answer worse than none
  - E.g, most short-running batch processes

Automation and scalability

- Before: repair required a hand-written consistency specification
- This work: generate specification automatically
- Evaluation: realistic programs, but human still specifies which structures are important

Outline

Introduction

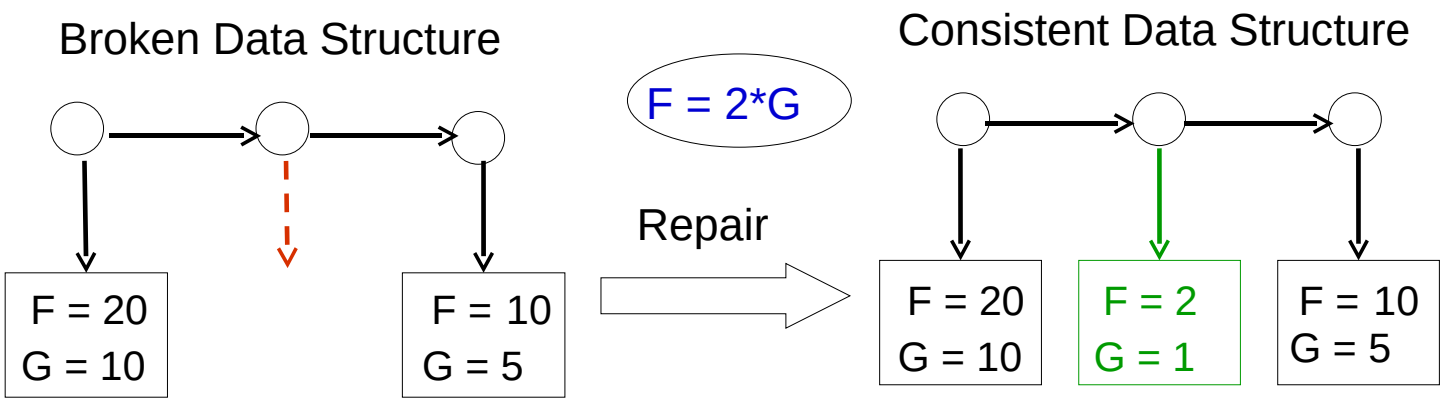
Repair and inference tools

Network attack case study

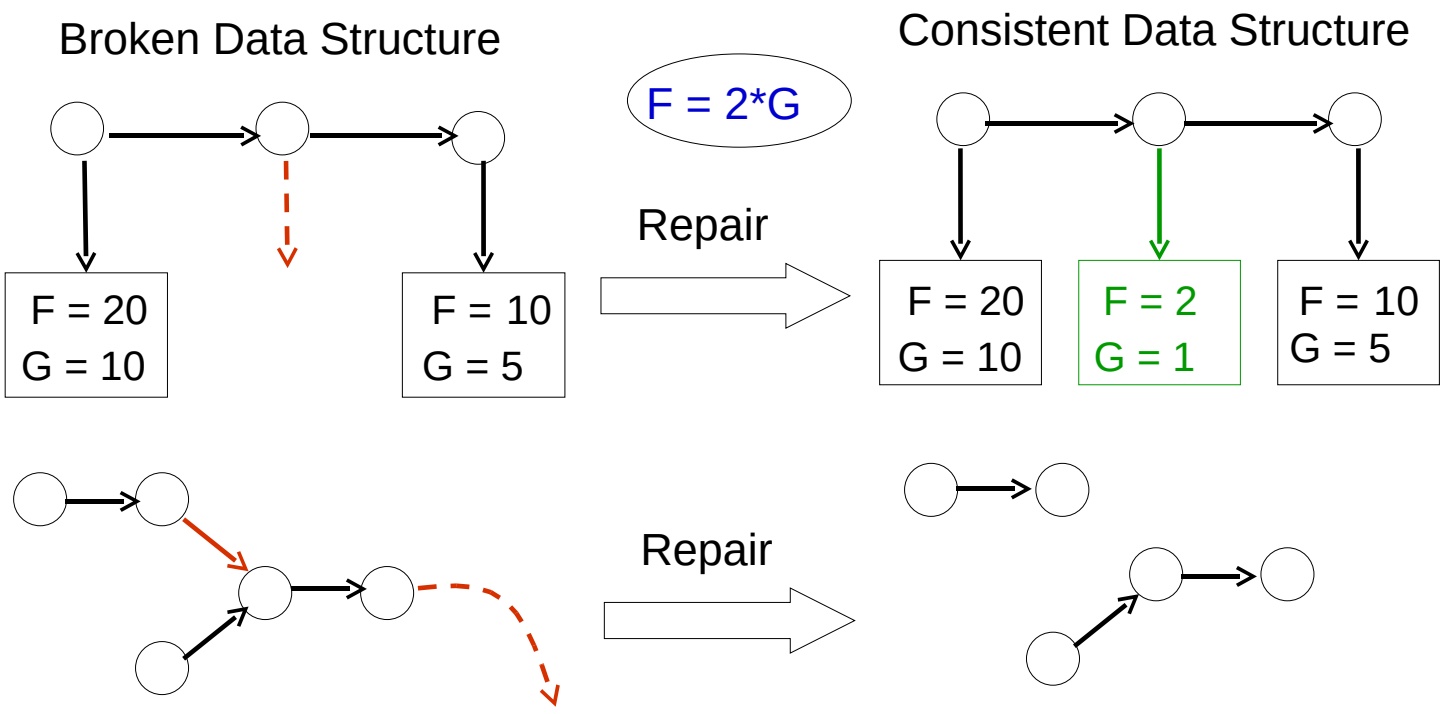
Red Team case study

Conclusion

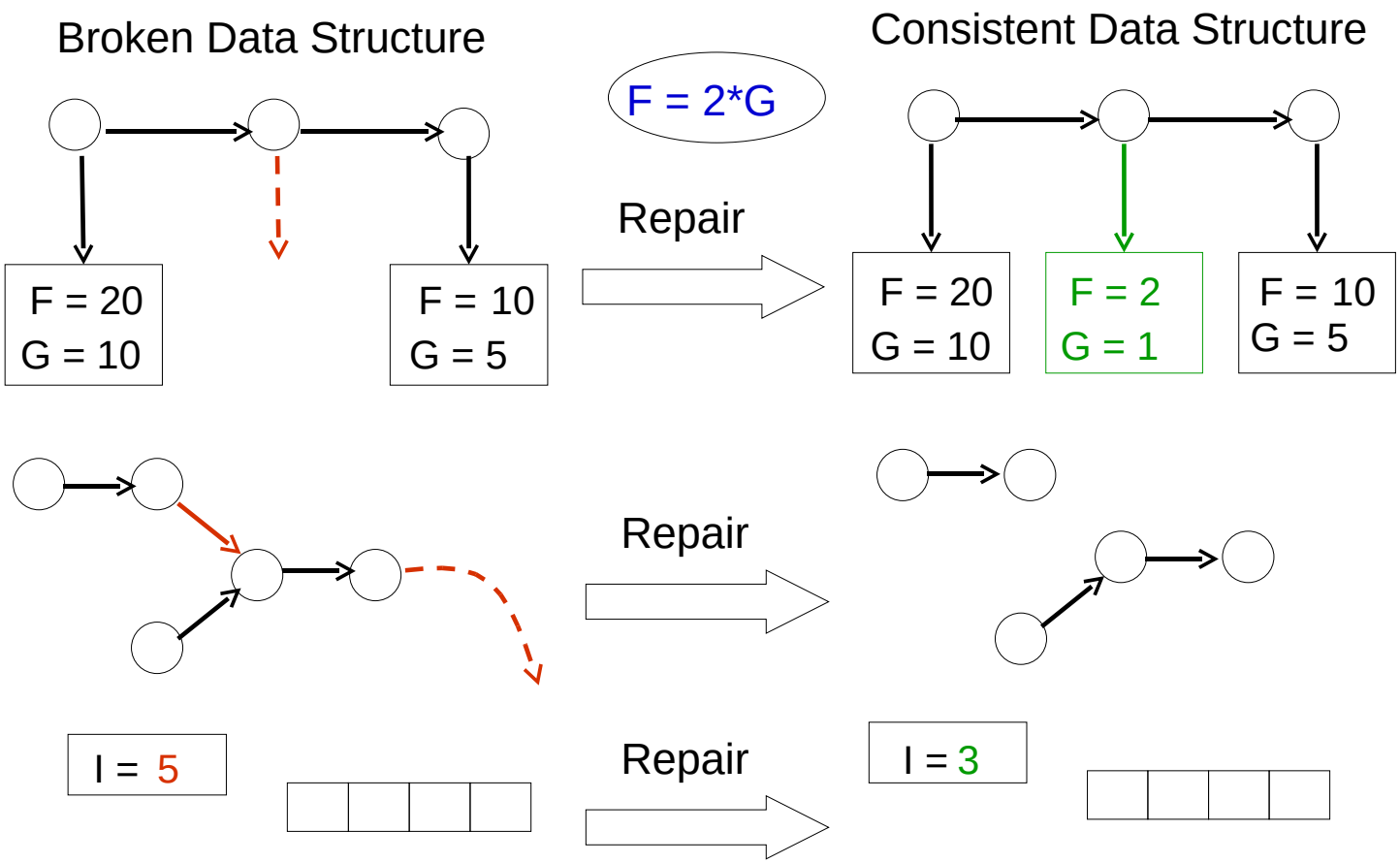
Repair actions



Repair actions



Repair actions



Automatic repair algorithm

1. Evaluate consistency specification
2. Select violated basic propositions
3. Update data structure to satisfy propositions
4. Repeat until all constraints satisfied
   - A static analysis ensures no infinite repair loops possible

Dynamic invariant detection

- Infer spec by examining runtime values, using Daikon 

- Algorithm:
  - Conjecture all properties from a large grammar
  - At each dynamic program point, discard falsified properties
  - Result: all properties satisfied by test suite

Scaling program tracing

- Enable operation on larger C programs than before
- Implemented new value tracing front-end
- Based on dynamic translation (Valgrind) and debugging information (Fjalar)

Specification translation

- Daikon output describes concrete data values
- Repair tool spec has two parts:
  - Mapping from concrete structures to abstract sets and relations
  - Properties about abstraction
- Must translate specification:
  - Expressions $\longrightarrow$ sets
  - Fields $\longrightarrow$ relations

Outline

Introduction

Repair and inference tools

Network attack case study

Red Team case study

Conclusion

DNS and BIND

- The DNS translates between domain names (`www.mit.edu`) and IP addresses (`18.7.22.83`)
- BIND is the most commonly-used DNS server
- We examined two historic DoS vulnerabilities related to data structures

Negative cache bug

- DNS servers cache both positive and negative replies
- Before version 8.4.3, BIND cached even illegitimate negative replies
- E.g., `ns.evil.com` says `www.mit.edu` does not exist

Negative cache TTL

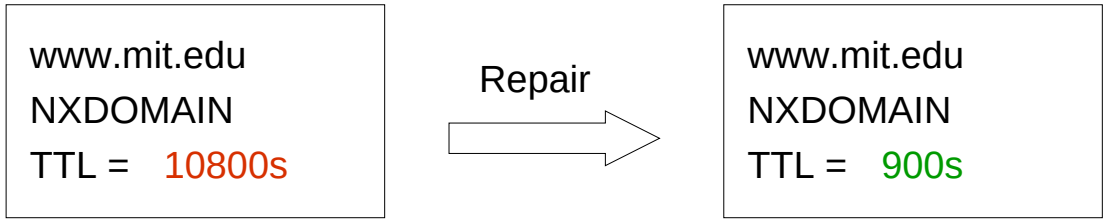
- Reply packet contains a time-to-live (TTL) saying how long to keep
- Attacker can choose how long attack bogus information will be retained
- Target host inaccessible for attacker-chosen interval

Negative cache TTL spec

- Observed normal operation of caching server
- In legitimate packets, negative reply TTL $\leq$ 15 minutes
- `message.sections[2].head.list.head.ttl <= 900`

Negative cache TTL repair

- Repair clamps attacker-supplied TTL to 900 second maximum



- Duration of DoS greatly reduced

NSEC verification bug

- NSEC record authenticates non-existence of a host
- BIND 9.3.0 contained a bug when verifying replies with an unusual record order: passed wrong record set to function
- Attacker can cause internal assertion failure (crash)

NSEC verification spec

- Validation function only takes NSEC record sets

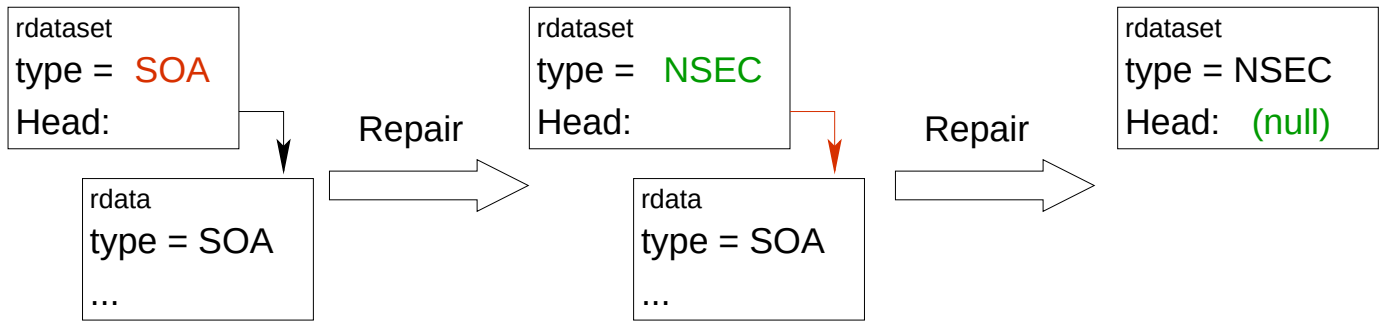
  `nsecset.type == 47`

- NSEC record sets only contain NSEC records

```
(rdataset.private1.rdata.head != null) =>
(rdataset.type ==
  rdataset.private1.rdata.head.type)
```

NSEC verification repair

- Adjusts set type to NSEC
- Element type doesn't match set type, record removed



- Malicious reply discarded, no crash

Outline

Introduction

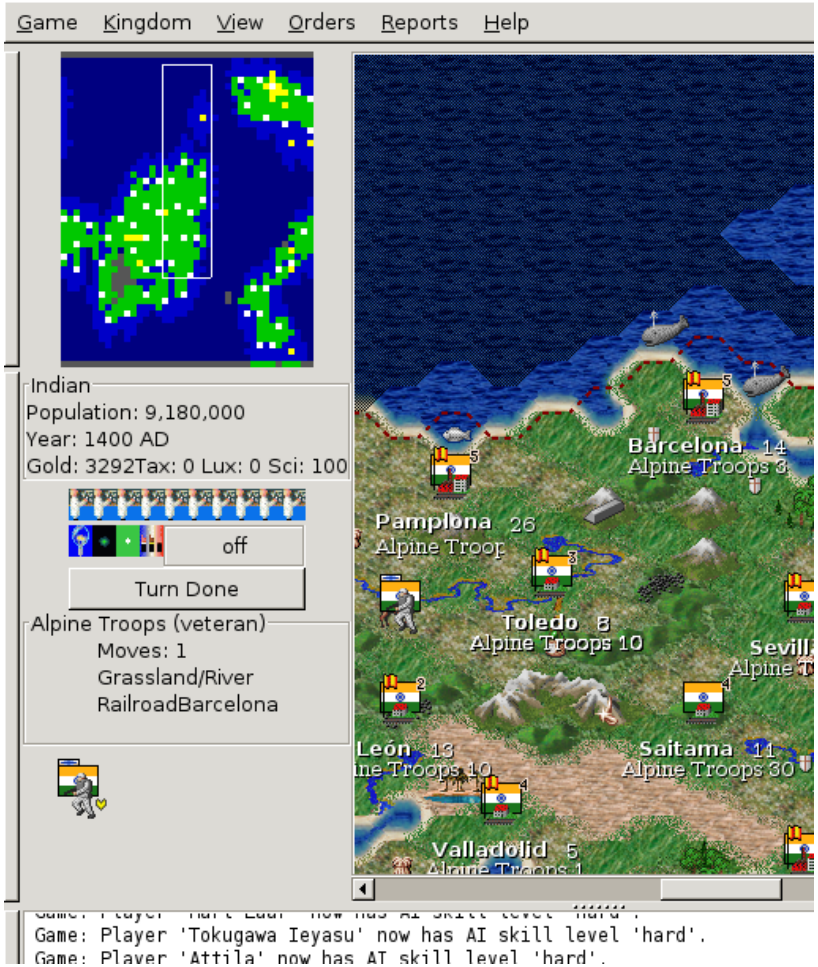
Repair and inference tools

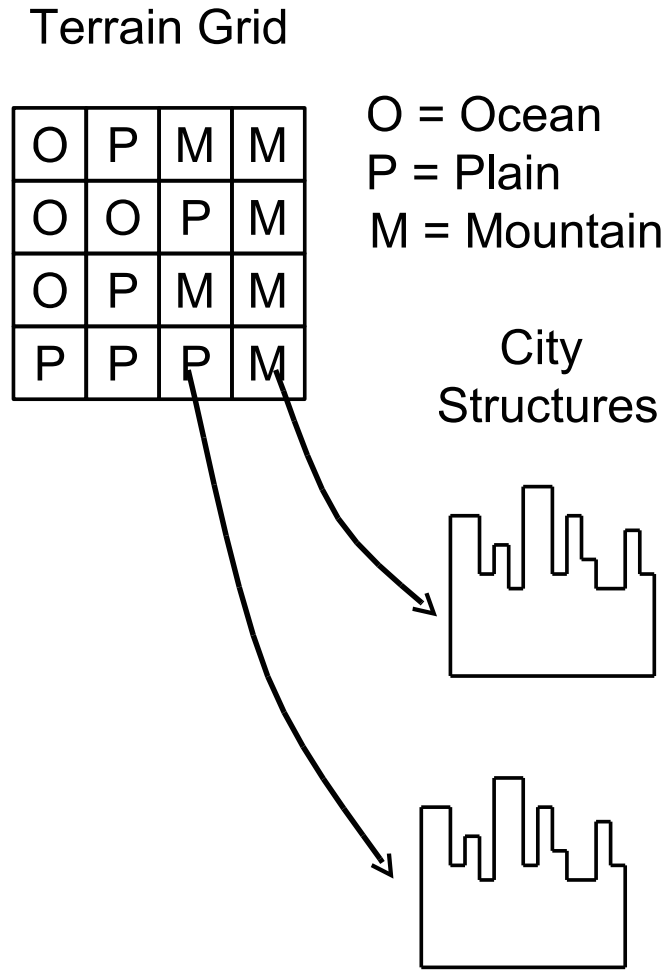
Network attack case study

Red Team case study

Conclusion

Freeciv


Game Kingdom View Orders Reports Help
Indian
Population: 9,180,000
Year: 1400 AD
Gold: 3292Tax: 0 Lux: 0 Sci: 100
off
Turn Done
Alpine Troops (veteran)
Moves: 1
Grassland/River
RailroadBarcelona
Barcelona 14
Alpine Troops 3
Pamplona 26
Alpine Troop
Toledo 8
Alpine Troops 10
León 13
Saitama 14
Alpine Troops 30
Valladolid 5
Game: Player 'Tokugawa Ieyasu' now has AI skill level 'hard'.


Terrain Grid
O P M M
O O P M
O P M M
P P P M
O = Ocean
P = Plain
M = Mountain
City
Structures

Comparison to manual spec: effort

- Test suite required
  - `11` configurations for auto-play mode
- Less detailed program knowledge needed
- Total time: hours versus days

Comparison to manual spec: results

- Automatic spec has $3.5\times$ more constraints:

  $\texttt{map.num\_contintents} \geq 2$

  $\forall \texttt{i}.\ \texttt{i.continent} \leq \texttt{map.num\_contintents}$

- Some manual facts not in grammar:

  $\forall \texttt{c}.\ \texttt{sizeof}(\texttt{c.CITYMAP}^{-1}) = 1$

  $\forall \texttt{c}.\ \neg(\texttt{c.CITYMAP}^{-1}\texttt{.terrain} = 7)$

Comparison to manual spec: quality

- Possibility of human error eliminated
- On examination:
  - No overfitted constraints
  - Subjective evaluation just as good

Red Team exercise

- Outside Red Team evaluated effectiveness against malicious corruptions
- Red Team given complete source code and specifications
- Attempted to crash the system
- Corruption API allows arbitrary changes to protected data structures

Malicious fault results

- Our system:
  - Detected 80% of corruptions
  - Successfully corrected 75% of those
- The Red Team was unable to:
  - Induce non-terminating repair
  - Mount an additive attack combining many corruptions in sequence

Failures of repair

- Corruptions triggering failure before repair code reached
  - E.g., assertion failure
- Corruptions destroying all copies of some information
  - E.g., can recover width or height of board, but not both, given $\texttt{map.xsize} \cdot \texttt{map.ysize} = \texttt{sizeof}(\texttt{grid})$

Outline

Introduction

Repair and inference tools

Network attack case study

Red Team case study

Conclusion

Discussion

- Inferred specs not guaranteed to be appropriate
  - Still easier to check than to create from scratch
- Absence of crash $\neq$ correct execution
  - In case studies, observed behavior was acceptable
- Repair actions could disrupt other unwritten invariants
  - So could existing code operating on corrupt data

Limitations and future work

- Manually specified data structures of interest
  - → Identify critical data structures automatically
  - → Further improve scalability of inference and repair tools
- Static termination analysis restricts possible repairs
  - → Make choices at repair-time instead

Summary

- Automatically-generated constraints:
  - Less programmer effort required
  - Good enough for repair task
- Repair is effective against:
  - Real attacks on critical network servers
  - Attacks chosen by an adversary aware of the repair strategy

Questions?

Translation examples

- Variables $\longrightarrow$ singleton sets
- Arrays translated into a relation between index values and elements
  - $\forall$`i.<i, array[i]>` $\in$ `Array`
- Abstract sets cannot contain `null`
  - "`p != null`" $\longrightarrow$ "`p != null => p` $\in$ `Pset`" and "`sizeof(Pset) > 0`"

Effects of random corruption