

# Learning and Repair Techniques for Self-Healing Systems: Successes and Next Steps

Michael Ernst and Martin Rinard  
MIT CSAIL

Self-Regenerative Systems PI Meeting  
December 14, 2005

# Outline

Overview

SRS program metrics

Red team evaluation

Case study: BIND DNS server

Next steps

Conclusion

# Technical objective and approach

Goal: Prevent user-visible data structure corruption errors

- crashes
- security breaches
- improper behavior

Approach:

- Automatically **learn** consistency properties
- Automatically **detect** violations
- Automatically **repair** corrupted data structures

# Improvements to current practice

Existing practice (only for the most critical applications)

- **Manually** specify consistency properties
- Automatically detect violations
- **Manually** repair corrupted data structures

The manual steps are error-prone, tedious, and rare in practice

We reduce costs in human time, not computer time

- Automate previously impractical techniques

# Achievements

Self-learning and self-healing systems

- Observe their own behavior
- Preserve that behavior in the future

Applications:

- Repair errors, preventing negative consequences
- Diagnose errors (early warning of root causes)
- Many other uses for the learned specifications
  - Download: <http://pag.csail.mit.edu/daikon>

Result: increased system robustness

- Red team evaluation
- Case studies

# Outline

Overview

**SRS program metrics**

Red team evaluation

Case study: BIND DNS server

Next steps

Conclusion

# Program metrics

Goal:

- Diagnose 10% of root causes
- Automatically correct 5%

Evaluation 1: single data structure corruptions (9% crash program)

- Corruptions of protected fields; not all fields are protected
- Diagnosed 58% of corruptions (attempted repair)
- Automatically corrected 99% of crashes (with repair, only .1% crash the program)

Evaluation 2: Multiple simultaneous corruptions

- 10 corruptions: reduced crashes from 55% to 9%
- 20 corruptions: reduced crashes from 79% to 33%
- 30 corruptions: reduced crashes from 90% to 61%
- 40 corruptions: reduced crashes from 98% to 75%

Evaluation 3: Red Team

- Multiple data structure corruptions
- Hand-crafted attacks that crash the program
- Results: Blue Team corrected 4 out of 6

# Impediments to dramatic improvement

## Learning:

- Learning rich, complex properties can be slow  
Solution: Algorithmic improvements, focus on a critical subsystem (key data structures), perform mostly ahead of time
- Learning can produce too much output (overfitting)  
Solution: Use better test suites, stronger statistical tests

## Repair:

- Cannot guarantee repair of rich specifications  
Solution: Compute strategy dynamically (not statically)
- Cannot correct wide-scale data structure destruction  
Solution: Add more redundancy (e.g., checkpoints)
- Model construction is slow; repair only at single program point  
Solution: Check concrete data structures, generalize implementation

A complete solution must include other techniques

- Many errors cause data structure corruption, but other techniques may be more appropriate
- “Root cause” may be outside the program’s purview

# Outline

Overview

SRS program metrics

**Red team evaluation**

Case study: BIND DNS server

Next steps

Conclusion

# Target program: Freeciv strategy game

Client-server implementation (we focused on the server)

Available from `freeciv.org`

Key data structure: `map`

- All fields except pointers to two arrays
- Also some fields in `tile` array elements
- No city fields

# Evaluation process

1. Insert corruption API into code (for evaluation)
2. Choose test cases (manual, largely arbitrary)
3. Create constraints **automatically**
4. Create repair code **automatically**
5. Perform corruptions (manual or random) on **original** and **instrumented** versions of program, compare results
  - both fail
  - neither fails
  - only one fails

# Results

Corrected most data structure corruptions

Most frequent causes of failures to correct:

- Bug in correcting multiple fields
  - Fixed by end of day
- Multiplication:  $xsize \times ysize == size(tile\_array)$ 
  - If both  $xsize$  and  $ysize$  are corrupted, it is hard to know the proper values
- Assertion violation in original code
  - Due to data structures not protected by repair

Data structure checking and repair should occur frequently

# Large-scale corruptions

Difficult to repair large-scale corruptions

- Change many fields
- Eliminate entire pointer-based data structures

Reproducing the lost information requires **redundancy**

- Often unavailable or difficult to expose
- Setting an invalid pointer to null is a simpler (and often successful) repair

# Red team evaluation conclusions

The technique worked on the given fields

- It would be more compelling to cover all data structures

Learned specification was as good as a manually written one

Learning & repair alone is not a complete security solution

- Effectively solves one set of important problems
- Does not solve all problems

Pointed out several avenues for further work

- Key: more constraints for more of the program
- Several minor issues were identified

# Outline

Overview

SRS program metrics

Red team evaluation

**Case study: BIND DNS server**

Next steps

Conclusion

# BIND: a DNS server

DNS = Domain Name System

- Translates “`www.darpa.mil`” to “`192.5.18.102`”

BIND implementation

- Most common DNS server on the Internet
- A frequent target of security attacks
- <http://www.isc.org/sw/bind/>

Operation

- Listens for DNS requests, sends reply packets
- Reply can be positive or negative (“no such host”)
- Caches information from other DNS servers

# Why BIND?

Fixing BIND security flaws was a challenge from Lee Badger  
It would not have been our choice:

- We have no experience with this (complex) software
- Technique was untested on these kinds of security flaws
  - Different flavor from most data structure corruption

Pleasant surprise: **successful results!**

- Automatically learn constraints
- Automatically detect and mitigate real security vulnerabilities

# BIND security bugs

We used two real (previously discovered) security bugs

- Negative caching bug: denial of service due to incorrect cache entry
- NSEC validation bug: denial of service by crashing BIND
- (Not buffer overruns: other tools are adequate, so such errors are becoming much less common)
- (Our tools were *not* tuned for these particular bugs)

Methodology

1. Run machine learner on some sample executions
  - Rather than using many executions, we used very few executions and restricted the learner to specific fields
2. Run repair compiler on resulting specifications, link output into BIND
3. Run both original and instrumented versions of BIND, compare the behavior

All steps are automatic (except workarounds to minor tool bugs)

# Negative Caching Bug

DNS server results:

- positive: “www.darpa.mil exists with address 192.5.18.102”
- negative: “qqq.darpa.mil does not exist”

Results from other DNS servers are cached

- Results are checked before being inserted in the cache
- Due to a bug, some negative results were cached

Exploit:

1. Attacker is authoritative server for bad-guy.com
2. Attacker causes victim (BIND) to make a request about bad-guy.com
3. Attacker replies “www.darpa.mil does not exist; cache this result for 1 month”
4. Victim caches reply, has no access to www.darpa.mil

# Learning

Sample executions: queries for non-existent domain names  
Function `dns_ncache_add()` adds negative cache entries.  
Its specification bounds the cache time-to-live (TTL) to less than 15 minutes:

```
message.sections[2].head.list.head.ttl <= 900  
message.sections[2].head.list.head.ttl >= 29
```

# Repair

Without repair: the attack succeeds

- Bogus negative reply is cached and propagated

With repair: effects of negative caching bug are ameliorated

- Repair code detects excessive TTL value and reduces it to 15 minutes
- Malicious reply is still cached, but duration of attack is greatly reduced
- After 15 minutes, incorrect information is flushed from the cache

# NSEC Validation Bug

DNSSEC security extensions

- Add cryptographic signature to DNS replies

Bug in code to check negative replies

- Assumes fields in a given order

Exploit:

1. Attacker is authoritative server for `bad-guy.com`
2. Attacker causes victim (BIND) to make a request about `bad-guy.com`
3. Attacker sends NSEC record with fields in a different (legal) order
4. Assertion failure in victim (BIND) code
5. Victim crashes, provides no service to users

# Learning

Sample executions: queries for non-existent domain names

1. Function `nsecnoexistnodata()` validates a negative NSEC record list.

The argument has type 47 (NSEC record):

```
nsecset != null
nsecset.type == 47
nsecset.private1 != null
nsecset.private1.rdata.head != null
```

2. Function `isc__rdatalist_first()` function iterates over a list.

Each record's type is that of the list:

```
rdataset != null
rdataset.type >= 0
rdataset.private1 != null
(rdataset.private1.rdata.head != null)
==> (rdataset.type == rdataset.private1.rdata.head.type)
```

# Repair

Without repair: the attack succeeds

- Attacker crashes the BIND server with records in legal but unexpected order in a reply packet
- Denial of service until the BIND server is re-started

With repair: NSEC validation bug is rendered harmless

1. Repair code detects unexpected record field type, changes it to 47
2. Repair code detects non-matching type in list, sets list to null (i.e., it removes the offending record)
3. Existing code in BIND skips over the empty record set
4. BIND rejects the packet and continues normal operation

# Outline

Overview

SRS program metrics

Red team evaluation

Case study: BIND DNS server

**Next steps**

Conclusion

# Topics for future work

Dealing with large-scale destruction

Preventing incorrect repairs

Scaling

- To many properties
- To complex properties
- Identifying key data structures

Avoiding overfitting in machine learning

Tool improvements

More experiments

# Large-scale destruction

Suppose that an attacker erases all of memory

- Or some substantial part of it

Can you reconstruct it in its previous state?

For large-scale corruptions

- Regenerate from scratch
- Requires **extra redundancy**

Example: Record recent good states (checkpoints, backups)

- Roll back (in part or whole) when error discovered
- Adds redundancy

Similar to existing recovery strategies

- Automatically apply protection to data structures

# Incorrect behavior after the repair

Data structure is repaired to a consistent state

- Not necessarily to the *same* state it would have been in

We have not observed this to be a problem in practice

- Corruptions tend to be small, and the repair tends to be small, so the result is the original or a nearby state
- “Correct” behavior is rarely specified or even known
- The learning system is fairly effective
- Future studies could determine how important this is

# Incorrect, but acceptable, behavior

**Partial** functionality may be better than **none**

A file is corrupted on disk

- With repair: recover part of the file
- Without repair: entire file is lost

An air traffic control system receives corrupted input

- With repair: a human must correct destination airport
- Without repair: system crashes, no air traffic control

If **no** functionality is better than **some**:

- Do not apply data structure repair
- Use heavyweight manual approaches, such as formal verification

# Preventing incorrect repairs

Focus on repairable/important data structures

Extra redundancy: checkpoints, checksums, etc.

User specifies mission-critical aspects of behavior

- Adds to learned specification
- Repair system respects these constraints

Improve the learned specification

- New learning algorithms
- Filter the learned constraints (manually or otherwise)
- Make incorrect repairs less likely

Repair system can ask for guidance

Flag results from repaired data for extra scrutiny

# Scaling

Techniques work well for modest sets of constraints

Apply learning and repair to part of a program

- Need research to identify **key data structures**

# Scaling machine learning

More variables (richer properties)  $\Rightarrow$  more time

More program points (quicker repair)  $\Rightarrow$  more time

Solutions:

- Static and dynamic analysis to reduce dynamic cost
  - Static specification inference
  - Comparability analysis
- New learning algorithms
  - online algorithms
- This is an ahead-of-time step

# Scaling data structure repair

Statically generated repair strategy works for any corruption

If no static strategy exists, algorithm creates no repair

- Even if any, or most, corruptions could be repaired
- More constraints  $\Rightarrow$  harder to make the static guarantee

Solutions:

- Select repair dynamically, based on actual corruption
  - Moves work from compile time to run time
- Better heuristics for what repair to choose

Other scaling needs:

- Extend repair to more constraints
- Check properties on concrete data structures instead of models
  - Saves model construction step
  - Not advantageous if model is used many times

# Avoiding overfitting

Overfitting: learning too many constraints

- Many constraints  $\Rightarrow$  repair algorithm fails
- Undesirable constraints can cause undesirable repairs
- Problem is correlated with expressiveness of learned constraints

Simplest solution: better test suites

- Need research on automatic test suite generation
- Scaling the learning algorithm helps process bigger suites

Enhance statistical tests

# Tool improvements

## Learning tool

- Improvements to statistical tests

## Repair tool

- Handle specifications for multiple program points
- Infer where to insert repair code

# Outline

Overview

SRS program metrics

Red team evaluation

Case study: BIND DNS server

Next steps

**Conclusion**

# Conclusion

Learning and repair of data structure constraints  
is a successful strategy for self-healing systems  
Solves realistic and important problems

- Real bugs in BIND, AbiWord, x86 emulator, ...
- Errors inspired by real bugs in CTAS, file system, ...
- Attacks in Freeciv

Repaired behavior is desirable in many real scenarios

- Can be constrained by user

Achieved goals for SRS program metrics

- Fault seeding and red team approaches

Learning and repair approach eliminates most manual steps

Future work: assurance and scaling