AFRL-IF-RS-TR-2006-157
Final Technical Report
May 2006

LEARNING AND REPAIR TECHNIQUES FOR SELF-HEALING SYSTEMS

Massachusetts Institute of Technology, CSAIL

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. S477

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2006-157 has been reviewed and is approved for publication

APPROVED: /s/

ALAN J. AKINS
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Tech Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MAY 2006	Final Jun 2004 – Dec 2005

4. TITLE AND SUBTITLE
LEARNING AND REPAIR TECHNIQUES FOR SELF-HEALING SYSTEMS

5. FUNDING NUMBERS
C - FA8750-04-2-0254
PE - 62301E
PR - S477
TA - SR
WU - SP

6. AUTHOR(S)
Martin Rinard, Michael Ernst

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology, CSAIL
32 Vassar Street
Cambridge Massachusetts 02139

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency AFRL/IFGA
3701 North Fairfax Drive 525 Brooks Road
Arlington Virginia 22203-1714 Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2006-157

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Alan J. Akins/IFGA/Alan.Akins@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Techniques were investigated for enabling systems to recover from data structure corruption errors. These errors could occur, for example, because of software errors or because of attacks. The technique developed contains three components. The first component observes the execution of training runs of the program to learn key data structure consistency constraints. The second component takes the learned data structure consistency constraints and examines production runs to detect violations of the constraints. The third component updates the corrupted data structures to eliminate the violations. The goal is not necessarily to restore the data structures to the state in which a (hypothetical) correct program would have left it, although in some cases our system may do this. The goal is instead to deliver repaired data structures that satisfy the basic consistency assumptions of the program, enabling the program to continue to operate successfully. A prototype system that contained these three components and the system was applied to two programs, BIND (part of the Internet Domain Name System (DNS)) and FreeCiv (a freely distributed multi-player game). Experience with BIND indicated that the technique can eliminate previously existing undesirable behavior in this program. The technique was applied to FreeCiv in the context of a Red Team experiment. The results of this experiment show that, on this program and for the workload in the Red Team experiment, our system signi cantly out-performed the DARPA Self-Regenerative System metrics: it recognized 80% (not just 10%) of the attacks, and it recovered from 60% (not just 5%) of them.

14. SUBJECT TERMS
Regenerative Systems, Cyber Defense, Automatic Code Generation, Data Restoration, Consistency Constraints, Learned Data Structure

15. NUMBER OF PAGES
23

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1 Summary

We investigated a technique for enabling systems to recover from data structure corruption errors. This technique contains three components. The first component observes the execution of training runs of the program to learn key data structure consistency constraints. The second component takes the learned data structure consistency constraints and examines production runs to detect violations of the constraints. The third component updates the corrupted data structures to eliminate the violations. The goal is not necessarily to restore the data structures to the state in which a (hypothetical) correct program would have left it — although in some cases our system may do this. Our goal is instead to deliver repaired data structures that satisfy the basic consistency assumptions of the program, enabling the program to continue to operate successfully.

We built a prototype system that contained these three components and applied this system to two programs, BIND (part of the Internet Domain Name System (DNS)) and FreeCiv (a freely distributed multi-player game). Our BIND experience indicates that our technique can eliminate previously existing undesirable behavior in this program. We applied our technique to FreeCiv in the context of a Red Team experiment. The results of this experiment show that, on this program and for the workload in the Red Team experiment, our system significantly out-performed the DARPA SRS metrics: it recognized 80% (not just 10%) of the attacks, and it recovered from 60% (not just 5%) of them. We also performed other activities as part of this project.

2 Introduction

Unlike biological organisms, which can respond robustly to damage from internal errors and external attacks by healing the damage and continuing to operate, computer systems have traditionally lacked mechanisms that allow them to detect and repair damage and avoid similar problems in the future.

One of the most important aspects of software systems is the integrity of its data structures, which store the information that the program manipulates. To correctly represent the information that a program manipulates, its information representation must satisfy key consistency constraints. If an attack, software error, or some other anomaly causes the information representation to become inconsistent, the basic assumptions under which the software was developed no longer hold. In this case, the software typically behaves in an unpredictable manner and may even fail catastrophically.

We proposed to apply a new technique, *consistency constraint repair*, to allow the system to heal such information representation damage and continue to execute successfully. A system that uses our technique accepts a specification of key information representation consistency constraints. It then dynamically detects and repairs parts of the information representation that violate these constraints. Our goal is not necessarily to restore the information representation to the state in which a (hypothetical) correct program would have left it — although in some cases our system may do this. Our goal is instead to deliver a repaired information representation that satisfies the basic consistency assumptions of the program,

enabling the program to continue to operate successfully. We proposed a system with the following components:

- **Automatically Learning Consistency Constraints:** Consistency constraint repair requires a specification of the consistency constraints. Previous systems used constraints provided by human developers. We proposed to apply machine learning technology that examines runtime behavior to learn key properties of successful executions. This approach promised to reduce the burden on the developer and help to ensure that the extracted properties are comprehensive, i.e., that they capture those key properties whose satisfaction will ensure that the program is able to continue to operate successfully after repair.

- **Error Detection:** Our consistency constraint repair algorithm contained an algorithm that can examine the information representation to find violations of the consistency constraints. We proposed to use this technology to enable the system to localize the source of errors that cause consistency constraint violations. We anticipated that the system would perform repeated consistency checks, ideally enabling it to find constraint violations quickly.

- **Repair:** Once the system found an error, we proposed to explore a repair strategy that used the automatically learned consistency constraints to repair the data structures so that they satisfied the key consistency constraints. The goal is to enable the program to recover from whatever caused the damage in the first place to continue to execute and successfully serve the needs of its users.

We started the project with a solid foundation of existing techniques and implementations. Specifically, we had: an implementation of our information representation repair technique (this implementation used manually-developed specifications), an implementation of our error localization technique (this implementation also used manually-developed specifications), and an implementation of a technique that learns operational abstractions that can be used in place of manually-developed specifications. We had positive results using each of these techniques in isolation. One of the goals of the research was to apply the techniques, synergistically and in combination, to enable error detection and recovery.

We identified one of the most important aspects of this combination to be the development of a common constraint language that would allow the data structure consistency constraint learning component to interact successfully with the data structure consistency constraint violation detection component and the data structure consistency constraint repair component. We anticipated that, given the solid foundation of existing components, we would be able to obtain meaningful results within the eighteen month time frame of the contract.

We were successful in realizing the combination of the learning, error detection, and repair components. As part of this combination we found that we needed to modify each of the basic components so that they could work more successfully with each other. On the learning side these modifications included the addition of new specification properties to the consistency constraint learning, and augmenting it to

produce output in the format required by the data structure repair tool. We implemented a new inference algorithm that greatly improved performance for programs that have large numbers of variables. We enabled our system to run online (without writing trace files to disk). We enhanced support for C and added support for C++. We added support for Java version 5 and have built a new infrastructure for Java instrumentation. We improved portability to the Windows operating system. On the data structure consistency constraint repair side we integrated support for working with the consistency constraint language. One of the important points of focus was scalability. We have detailed these and other changes in our quarterly reports.

3 Methods, Assumptions, Procedures

We adopted an experimental approach to our research. Early in the project we performed case studies on selected programs to evaluate our techniques. Our approach included running our invariant detection tool (Daikon) on a variety of programs to evaluate its capabilities. We also ran our data structure repair tool on selected programs with either real-world data corruption errors or errors that we obtained by using fault injection. In general, we observed the results we obtained and the general process of obtaining these results and used them to drive further development. We also identified any weaknesses or missing pieces and worked towards remedying the weaknesses and filling in any missing pieces. We also made large parts of our software available for download via the Internet.

During the course of the project we devoted a major effort to integrating and evaluating the various different components. Our integration efforts focused on developing software to connect the different components. Once the software was developed we tested it and updated it as the tests indicated was necessary. We evaluated our techniques by applying them to programs with different kinds of data structure corruption errors. During this process we observed any deficiencies and developed techniques that addressed these deficiencies. We developed the main results of the project on two programs, BIND (part of the Internet Domain Name System (DNS)) and FreeCiv (a freely distributed multi-player game).

The underlying assumption behind this research is that these empirical techniques will generalize to larger classes of programs. We see no way to test these assumptions other than testing our techniques out on programs.

In the later part of the project our activities centered around a Red Team exercise. The general idea behind a Red Team exercise is to set up an adversarial contest between a Red Team (in this case, a group from Raba, Inc.) and a Blue Team (in this case, the MIT group). A White Team officiated the contest.

The Red Team exercise centered around a specific program, FreeCiv. We used our learning tool to automatically learn key consistency constraints by observing its executions on standard test inputs. We used the produced data structure consistency constraints to augment FreeCiv with data structure consistency violation detection and repair. We then developed an interface that enabled the Red Team to corrupt various

parts of the data structures. We interacted with the Red Team to establish the parameters of the Red Team exercise, and performed the exercise.

4 Results and Discussion

We next present our results automatically generating data structure consistency specifications and using these data structure consistency specifications for data structure repair in the BIND and FreeCiv software systems.

4.1 Methodology

For each system, we selected an initial fault-free workload and identified the data structures of interest. We then executed the systems on this workload and used the Kvasir front end (a part of Daikon) to record a trace of the execution. We used Daikon to automatically extract a consistency specification for the data structures of interest from this trace. We manually reviewed this consistency specification and (for our two systems) found nothing we wanted to change.

We next used the data structure repair tool to compile the consistency specifications into C code that detects and repairs any inconsistencies in the data structures, then augment the programs with the repair code. Finally, we tested the ability of the resulting data structure repair algorithm to enable the system to recover from data structure corruptions. For BIND, we used a workload that exercised known data structure corruption errors. For FreeCiv, the Red Team used fault injection to corrupt the data structures. We then observed the continued execution of the program after the resulting repair. Note that the entire process of obtaining and enforcing the data structure consistency specification, with the exception of the specification review, is entirely automated.

For FreeCiv we had previously developed a manual specification. We evaluate the automatically generated manual specification, in part, by comparing it to this manual specification. Our evaluation focuses on the coverage of the properties in the automatically generated specification and the difficulty of developing the manual specification from scratch compared with reviewing the automatically generated specification.

4.2 BIND

The Domain Name System (DNS) is an Internet service responsible most notably for translating human-readable computer names (such as www.mit.edu) into numeric IP addresses (such as 18.7.22.83). BIND (http://www.isc.org/sw/bind/) is an open-source software suite which includes the most commonly used DNS server on the Internet. Because of BIND's ubiquity on the Internet, it is a frequent target of security attacks, and a number of serious flaws have been found in it over its decades of use. The most recent major revision of BIND, version 9, is an almost complete rewrite, intended among other changes to be more secure.

BIND's basic operation is straightforward: it listens for DNS requests on a network socket and sends reply packets containing information from the DNS database. Each Internet domain (such as .uk, .google.com, or .csail.mit.edu) has one or more "authoritative" servers that provide information about hosts (computers) in that domain and point to its sub-domains; in addition most networks have "caching" servers which handle requests from clients (such as desktop computers), communicate with authoritative servers, and retain results for a limited time period so that repeated requests can be processed more efficiently. Currently most DNS traffic on the Internet does not use any form of strong authentication, but an extended version of the DNS protocol, known as DNSSEC, allows authoritative information to be cryptographically signed by a domain's owner.

4.2.1 BIND Errors

The BIND developers maintain a list of security-critical bugs (http://www.isc.org/sw/bind/bind-security.php). Many earlier BIND security bugs were classic buffer overruns. Existing tools can detect and correct such problems [30, 12, 27], which have also become less common in recent BIND versions, perhaps because of more careful coding practices and auditing.

Our evaluation considers attacks involving higher-level data structure changes, which represent a greater proportion of recent vulnerabilities.[1] We selected two previously-discovered (and -corrected) problems: the "negative caching bug" (section 4.2.2) and the "NSEC validation bug" (section 4.2.3). Both of these represent denial-of-service vulnerabilities: a malicious user interacting with BIND could prevent the server from handling legitimate requests. The bugs existed in historical versions of BIND; to simplify our experiments, we reproduced them by introducing the same defects into the most recent version of BIND, 9.3.1.

4.2.2 Negative Caching Error

An authoritative DNS server can return either positive results (for instance, www.mit.edu exists and its address is 18.7.22.83) or negative ones (for instance, no host qqq.mit.edu exists). Both positive and negative results may be cached, and both positive and negative replies contain a field (called the TTL, or "time-to-live") indicating for how long they should be cached. Versions of BIND 8 prior to 8.4.3 contained a bug in the way they cached some negative results. When a caching server received domain information in a reply from an authoritative server, it performed several checks on the consistency of the data: for instance, the server from which the data originated should be the authoritative server for the domain the results refer to, and the results should pertain to the domain as the original query. If these checks fail, the results are considered illegitimate, and discarded. Because of a logic error in vulnerable versions of BIND, however, a packet that had been determined to be illegitimate was sometimes still added to the cache of negative information.

[1] Code-level techniques such as ours probably are not effective in addressing protocol errors — fundamental algorithmic or design errors — which are another category of common problems.

To exploit this bug, an attacker who controls an authoritative server for some domain modifies it so that when replying to requests, it returns negative results about some unrelated domain. For instance, if the attacker controls attacker.com, a query to that domain's server for the address of mailserver.attacker.com might yield a reply saying that "www.mit.edu does not exist". The attacker then gets a vulnerable caching server to make a request to the malicious server, for instance by sending an email with a "From" address at attacker.com to a mail server that uses the vulnerable caching DNS server. The caching server will incorrectly retain the negative information, so that any attempts by other users of the caching server to access the target host (www.mit.edu in the example) will fail until the information expires from the cache. Normally, negative results have a small TTL, and so would expire quickly, but in this attack the TTL for the incorrect reply is chosen by the attacker, and can be arbitrarily long.

Obtaining Specifications: We selected components of the message data structure of interest. Daikon then observed the execution of BIND as it responded to several dozen queries, mostly for domain names that did not exist. We instructed Daikon to observe the dns_ncache_add() function, which runs every time a query for a non-existent address arrives. This function adds an entry for that address to the negative cache.

The resulting specifications included bounds on the time-to-live (TTL) field of a message that is added to the BIND negative cache.

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

The exact numbers may differ from the ones shown (900 seconds, which is 15 minutes), depending on conditions in the server's environment. In particular, different authoritative servers will supply different (valid) TTL values.

Effect of Repair: Without repair, we verified that an attacker can set an arbitrarily large time-to-live on a (bogus) negative reply; as a result, the attacked DNS server will retain (and propagate) the bogus negative reply for days, weeks, or longer.

When our tool's repair code is active, the effects of the negative caching bug are ameliorated. The repair code detects that the hostile server's response has an excessive TTL value; it repairs the data structure holding the TTL, setting the TTL to the inferred upper bound of 900 seconds. Though the malicious reply is still cached, its TTL is limited to the maximum TTL obtained by observing legitimate data (such as 15 minutes), rather than the very long TTL chosen by the attacker. After this period expires, the incorrect information is flushed from the cache, and the DNS server again operates properly.

4.2.3 NSEC Validation Error

The NSEC validation error, reported in early 2005, is part of the validation that BIND can perform when processing authoritative replies. Under the DNSSEC security extensions, each piece of data returned by a server may be accompanied by a

cryptographic signature, which can be checked to verify its authenticity. A particular complication in DNSSEC concerns negative results: since there is normally no record corresponding to a negative result, there is no obvious object to be signed. To allow the authentication of negative results, DNSSEC introduces a new kind of record, of type "NSEC", to record negative information. For each name that exists in a domain, there is an NSEC record that lists the next name in the domain, in a cyclic alphabetical order, as well as which kinds of data exist for the name. When a request for a nonexistent name is received, a DNSSEC-compliant server can send the NSEC record for the alphabetically closest previous existing name, and its signature, to convince a recipient that no such data exists; similarly an NSEC record can prove that while a name exists, no data of the requested type is available.

BIND version 9.3.0 contained a bug in the code to check such signed negative responses. Normally, a secure negative reply DNS packet would contain a section with four records: an NSEC record verifying the nonexistence of the requested record, an SOA record indicating that the server that generated the reply data is the legitimate authority for the domain in question, and two RRSIG signature records containing signatures for the aforementioned other records. The DNS protocol places no requirements or significance on the order of the four records, but by convention BIND and other servers usually use the order SOA, RRSIG, NSEC, RRSIG.

The code in BIND 9.3.0 processes the records in the order in which they appear in the reply packet. At one point, it performs a check that is meant to determine whether the record currently being examined is an NSEC record, but because of a coding error, the check instead succeeds whenever an NSEC record has been seen so far in the entire section. If a malicious server sends the records in an unconventional order, such as NSEC, RRSIG, SOA, RRSIG, and the NSEC record fails to verify, the vulnerable server will attempt to perform NSEC verification on an SOA record: the code that performs this verification checks the type tag on the record, sees that it is unexpected, and triggers an internal assertion failure that causes the server to immediately terminate. The server will of course then be unable to respond to any requests until it is restarted (e.g., manually).

Obtaining Specifications: We selected components of the nsecset and rdataset data structures to be of interest. Daikon then observed executions of a BIND server that was communicating with a second BIND server. The server being observed was configured to cache DNSSEC entries. The second server was configured to provide authoritative DNSSEC data for a domain. The observed server was configured to forward requests the second server, and to consider the second server's public key valid for authentication. Our testing made various queries of the first server, and we instructed Daikon to observe two functions nsecnoexistnodata() and isc_rdatalist_first().

The function nsecnoexistnodata() checks a record set containing an NSEC record to determine whether that record correctly authenticates the nonexistence of a name, or the nonexistence of some particular data type at that name. In this case, the record set is implemented as an "rdatalist" data structure. The isc_rdatalist_first() function initializes an iterator on an rdatalist to point to the first record in the list, or returns an error condition if the list is empty.

The inferred specifications were as follows.

```
../../../lib/dns/validator.c.nsecnoexistnodata():::ENTER
nsecset != null
nsecset.type = = 47

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

Two properties are relevant for the repair. The first, nsecset.type = = 47, indicates that type of the record set passed to nsecnoexistnodata() must always be 47, the value which denotes an NSEC record. A second property, (rdataset.private1.rdata.head != null) = => (rdataset.type = = rdataset.private1.rdata.head.type), indicates that if an "rdatalist" record set contains a record, that record's type must be the same as the type of the record set overall.

Effect of Repair: Without repair, we verified that an attacker can crash the BIND server by sending records in an unexpected but legal order in a reply packet. This results in a complete denial of service until the BIND server is restarted.

When our tool's repair code is active, the NSEC validation bug is rendered harmless. In the nsecnoexistnodata() function, the repair code detects that the record field type is unexpected, and changes it to 47. Then, at the isc__rdatalist_first() function, the repair code detects that rdataset.private1.rdata.head.type ≠ rdataset.type, and so it sets the rdataset.private1.rdata.head field to null — in other words, it removes the offending record. Existing code in BIND then sees that the record set is empty, so that verification cannot continue. BIND rejects the packet as invalid, and continues normal operation without failing.

4.3 FreeCiv

FreeCiv is a freely distributed, multiplayer, client-server strategy game (http://www.freeciv.org/). It contains a total of 93,612 lines of code of which the FreeCiv server uses 78,555 lines. This server maintains a map of the game world. Each tile in this map has a terrain value chosen from a set of legal terrain values. Additionally, cities may be placed on the tiles.

4.3.1 Obtaining Specifications

We identified the civmap, tile, and map_positions data structures to be of interest. Within these data structures Daikon observed all primitive data type fields (int, char, etc.). It did not observe pointer fields — the data structure repair algorithm

automatically protects against basic pointer corruption by enforcing the constraint that pointer fields must either be null or point to a valid region of memory.

Daikon observed runs of FreeCiv in an automatic execution mode in which several computer-generated players play against each other. We presented it with runs with a variety of parameter settings; these settings include values such as the number of players, the random seeds, the size of the game map, the percentage of various types of terrain, the map generation algorithm, and the time when the game ended. Running the game with a variety of parameters avoided the overspecialization that could otherwise result if Daikon only observed runs with fixed parameter values.

A review of the generated specifications revealed that there was still some overspecialization in the automatic specification generation process. But all of the overspecialized properties happened to be filtered out by a component of the repair algorithm generator that discarded properties that might cause the repair algorithm to loop forever if enforced. We therefore used the automatically generated specifications without change.

4.3.2 Comparison with Manually-Generated Specifications

In previous work, we manually developed specifications for the FreeCiv program. We found several advantages to automatically inferring specifications. The first advantage is that automatically inferring the specification took less effort. We developed a test suite containing 11 different game configurations, then ran FreeCiv to generate execution traces for each of these test suites, and then used Daikon to automatically infer these invariants. We then reviewed the invariants, using the properties as a foundation from which to build enough of an understanding of the program's operation to verify that the properties were not overly specialized to the test suite.

To manually develop specifications, we had to understand how FreeCiv manipulated the data structures and then write the appropriate invariants. In our experience manually developing the specification required significantly more effort than obtaining the specification automatically via Daikon. Note that the original FreeCiv specification was written by the developer of the repair system; we imagine that manually developing specifications would be more difficult for novice users and, therefore, they would find even greater benefit.

The second advantage is that the inferred specification had significantly more invariants than the original manually-developed specification: the inferred specification contains 21 constraints while the original manually-developed specification only contains 6 constraints. Many of these constraints were missing because we were simply unaware of them.

Corruptions	Crashes out of 100 executions		
	Original	With repair	Crashes averted
50	92	12	80
100	95	41	54
150	98	42	56
200	97	62	35
250	97	56	41
300	99	85	14
350	100	80	20
400	98	83	15
450	100	89	11
500	99	82	17

Figure 1: Number of executions that crash, after a given number of data structure corruptions are simultaneously applied. The table indicates how often the original program crashed, and how often the program crashed if augmented with data structure repair.

The third advantage is that inferred specifications may be more likely to be correct. Automatically inferring specifications eliminates the errors that a developer may make when developing a specification. For example, a developer may forget to write a constraint or be unaware of a constraint.

However, automatic inference of specifications has limitations. The inferred specifications are limited to the invariants that Daikon supports. For example, the inferred FreeCiv specification is missing an invariant that states that a city is referenced by at most one tile and an invariant that ensures that cities are not placed on tiles with ocean terrain values. Daikon also requires the application to have a test suite. However, we expect that in most cases that a developer will have a pre-existing test suite that could be used. Finally, this technique does not eliminate all manual effort. The developer may still need to manually review the specifications to ensure that the invariants are not overspecialized.

4.3.3 A Fault Injection Experiment

To support our fault injection experiments, we developed a fault injection API that allowed an attacker to easily corrupt fields of interest. We then used this API to explore the ability of our technique to enable FreeCiv to recover from data structure corruptions. We used as our measure of success the number of program crashes that the repair system was able to avert. We measured this quantity by adding the corruption API both to the original program and also to a version of the program that had been instrumented with repair code. We then applied the same corruptions to both programs and counted the number of times that each one crashed.

Figure 1 reports the results. The first column gives the number of memory locations that the fault injection corrupted. The second column gives the number of times the original FreeCiv program crashed out of 100 executions. The third column gives the number of times the repair-augmented FreeCiv program crashed out of 100 executions. And the final column gives the number of crashes that data structure repair averted (the difference between the second and third columns) out of 100 executions. When 50 memory locations are simultaneously corrupted, the original FreeCiv program crashes 92% of the time. By contrast a version of FreeCiv augmented with data structure repair crashed only 12 times, because the repair code detected and corrected the corruptions. Therefore, with 50 corrupted memory locations data structure repair averted 80 crashes. When 100 fields are simultaneously corrupted, the original FreeCiv program crashed 95 times, compared to 41 times for the augmented version of FreeCiv.

In general, both the original version and the version with repair incur more crashes as the number of corrupted memory locations increases. Note, however, the fragility of the original version — it almost always crashes regardless of the number of corrupted memory locations. Repair can avert a substantial number of these crashes.

4.3.4 Red Team Activities

We also participated in a Red Team activity in which a team of engineers attempted to use the corruption API to cause FreeCiv to fail. This activity involved a Blue Team (the authors) and an outside three-person Red Team whose responsibility it was to attack the system provided by the Blue Team. The Red Team was given complete information about the system they were attacking, including the following:

- All tools used by the Blue Team (in source and binary form), including Daikon and the repair compiler, and manuals/instructions for their use.
- FreeCiv source code (the version that we are using),
- The FreeCiv test cases used to obtain the data structure consistency specifications.
- The data structure consistency specification. This is the output of Daikon, and is the input to the repair compiler. It indicates exactly what properties the instrumented version of the program will check and will attempt to re-establish if found to be false.
- The corruption API. This API lets the Red Team directly modify the contents of memory, by specifying a variable/field and a new value for it. The API also permits examining data structures and logging repair tool actions, permitting understanding and confirmation of the system's behavior.

The Red Team was given the various materials and documents between 3 months and 1 week in advance. They were on-site at MIT for three days. Note that by examining the specification and the tool documentation and source code, and by running the system, the Red Team could determine which corruptions our system would detect and what actions it would take.

We used failures of the program as our measure of success. The Red Team performed many attacks, and an attack was said to succeed if it crashed the program, and to fail

if it did not crash the program. All parties knew which attacks crashed the original, unprotected program, so the Red Team focused on those.

The Red Team used the provided corruption API (and other mechanisms, see below) to inject corruptions into the data structures of the running FreeCiv program. The Red Team devised the corruptions using techniques including intuition, examination of the specification and the repair tool, and random generation.

Our repair system detected 80% of the data structure corruptions that were introduced by the Red Team and took successful corrective action (repaired the data structure sufficiently for the program to continue without crashing) in 75% of those cases. Examples of corruptions that were successful repaired included random corruptions, wholesale replacement of certain data structures, attempts to violate specific properties that the Red Team had seen in the inferred specification, and setting data structures to null.

The Red Team was unable to induce a non-terminating repair — that is a data structure corruption such that in repairing the structure, the system enters an endless loop of repairs. The Red Team was also unable to mount an additive attack, in which a sequence of repairs (in response to a sequence of corruptions) were made that satisfied the specification but which degraded system behavior to the point of a later failure.

The Red Team's main successes in defeating the repair system (and crashing FreeCiv) fell into four main categories. The first was a simultaneous corruption of both the x and y map sizes — although the data structure repair algorithm was able to use the size of the allocated map data structure to detect a corruption, it was unable to come up with valid x and y values that, when multiplied, would produce that size. The second Red Team success involved corruptions that triggered assertion violations in the target program. In retrospect, it may have been possible to avoid these kinds of failures by simply disabling assertions. The third kind of Red Team successes involved many simultaneous corruptions that simply overwhelmed the program with an enormous amount of lost information. The last class of Red Team successes involved the use of the GDB debugger to corrupt arbitrary memory locations far (in the execution stream) from the point in the execution that checked and repaired the data structures. The effect was either a failure before the repair code was encountered, or — as in two of the other three types of Red Team success — the propagation of corruption into so many data structures that successful recovery was not possible.

4.4 Discussion

Our two programs exemplify very different ways in which automatically generated data structure consistency specifications in combination with data structure repair can affect the execution of the program. In BIND, our techniques ameliorated or even eliminated the effects of security attacks. Intriguingly, one of these attacks is arguably not even a data structure corruption attack — it simply injects an undesirable (but arguably not inconsistent) value into a data structure. By keeping this value within observed bounds, our technique ameliorates the negative impact of this value and may keep the program's behavior closer to its anticipated behavior.

FreeCiv illustrates that our technique can help systems recover from surprisingly extensive damage, in particular much more extensive damage than is likely to result from any single data structure corruption error. We were surprised at this result (we anticipated that, even with repair, the system would be much more brittle than it turned out to be) and consider it to be an encouraging indication of the effectiveness of our technique.

One issue that came up is that the repair algorithm statically generates a repair strategy that is guaranteed to terminate no matter what data structure exists at run time. Essentially, it ensures this by considering all possible interactions among the data structure properties and determining whether there are any possible dependence cycles between repair actions for the properties. This static guarantee is useful, but it means that if the specification is very rich (there are many properties or the properties are very detailed), then they are likely to interact in many ways. As a result, the algorithm may state that it cannot statically guarantee termination (even if termination could be guaranteed dynamically or always occurred in practice). While our current repair system deals with this issue, in part, by discarding properties that might lead to an infinite repair loop, it might be worthwhile to consider alternate approaches that abandon the termination guarantee in return for handling more properties.

5 Other Activities

In addition to the focus on data structure consistency constraint learning and enforcement, we performed a variety of other activities under this contract. These include investigating techniques for mode selection based on introspection into the program's behavior. It takes effect when a program is underperforming (likely due to an unexpected environment) and automatically chooses a program modality. We investigated applications of various kinds of failure-oblivious computing to enhance the effectiveness of our repair techniques. Failure-oblivious computing is a suite of techniques that are designed to keep programs executing through otherwise potentially fatal errors. We also worked on software transactions, which are another way of promoting data structure consistency. We also worked on an evaluation of our updgrade selection activities. We also presented a demo of our techniques at DARPATECH.

6 Related Work

We survey related work in invariant inference, software error detection [5, 8, 18, 4], traditional error recovery, manual data structure repair, and databases.

6.1 Invariant Inference

Our technique uses dynamic (runtime) analysis to extract semantic properties of the program's computation. This choice is arbitrary; for example, one could alternately

perform a static analysis (such as abstract interpretation [6]) to obtain semantic properties.

We use the Daikon dynamic invariant detector to generate runtime properties [13]. Its outputs are likely program properties, each a mathematical description of observed relationships among values that the program computes. Together, these properties form an *operational abstraction* that, like a formal specification, contains preconditions, postconditions, and object invariants.

Daikon detects properties at specific program points such as procedure entries and exits; each program point is treated independently. The invariant detector is provided with a trace that contains, for each execution of a program point, the values of all variables in scope at that point.

For scalar variables *x*, *y*, and *z*, and computed constants *a*, *b*, and *c*, some examples of checked properties are: equality with a constant ($x = a$) or a small set of constants ($x \in \{a,b,c\}$), lying in a range ($a \leq x \leq b$), non-zero, modulus ($x \equiv a \pmod{b}$), linear relationships ($z = ax + by + c$), ordering ($x \leq y$), and functions ($y = fn(x)$). Properties involving a sequence variable (such as an array or linked list) include minimum and maximum sequence values, lexicographical ordering, element ordering, properties holding for all elements in the sequence, and membership ($x \in y$). Given two sequences, some example checked properties are elementwise linear relationship, lexicographic comparison, and subsequence relationship. Finally, Daikon can detect The properties are sound over the observed executions but are not guaranteed to be implications such as “if $p \neq null$ then p.value > x” and disjunctions such as “
p.value > limit or p.left ε mytree”.

true in general. In particular, different properties are true over faulty and non-faulty runs. The Daikon invariant detector uses a generate-and-check algorithm to postulate properties over program variables and other quantities, to check these properties against runtime values, and then to report those that are never falsified. Daikon uses additional static and dynamic analysis to further improve the output [14].

6.2 Traditional Error Recovery

Reboot potentially augmented with checkpointing is a traditional approach to error recovery. Database systems use a combination of logging and replay to avoid the state loss normally associated with rolling back to a previous checkpoint [15]. Transactions support consistent atomic operations by discarding partial updates if the transaction fails before committing. There has recently been renewed interest in applying many of these classical techniques in new computational environments such as Internet services [26] and in extending these techniques to reboot a minimal set of components rather than the complete system [1].

6.3 Manual Data Structure Repair

The Lucent 5ESS telephone switch [19, 17, 22, 16] and IBM MVS operating system [25] use inconsistency detection and repair to recover from software failures. The software in both of these systems contains a set of manually coded procedures that periodically inspect their data structures to find and repair inconsistencies. The reported results indicate an order of magnitude increase in the reliability of the system [15].

6.4 Constraint Programming

Researchers have incorporated constraint mechanisms into programming languages. One such system is Kaleidoscope [23]. Kaleidoscope allows the developer to specify constraints that the system should maintain. The developer is intended to write programs using a hybrid of imperative style programming and constraints where appropriate. Kaleidoscope does not include any analog of our model-based approach, as a result it can be very difficult if not impossible to express constraints on recursive data structures or other heap structures containing multiple elements. Another example of a constraint maintenance system as a programming abstraction is Alphonse [20]. Rule based programming [24, 7] is a related technique in which the developer defines a test condition and an action to take in response.

6.5 Integrity Maintenance in Databases

Database researchers have developed integrity management systems that enforce database consistency constraints. These systems typically operate at the level of the tuples and relations in the database, not the lower-level data structures that the database uses to implement this abstraction. One approach is to provide a system that assists the developer in creating a set of production rules that maintain the integrity of a database [3]. This approach has been extended to enable the system to automatically generate both the triggering components and the repair actions [2]. Researchers have also developed a database repair system that enforces Horn clause constraints and schema constraints (which can constrain a relation to be a function) [29]. Our system supports a broader class of constraints — logical formulas instead of Horn clauses. It also supports constraints which relate the value of a field to an expression involving the size of a set or the size of an image of an object under a relation. Finally, it uses partition information to improve the precision of the termination analysis, enabling the verification of termination for a wider class of constraint systems.

6.6 Specification-Based Repair

In our previous research, we have developed a specification-based repair system that uses *external constraints* to explicitly translate the model repairs to the concrete data structures [10, 11, 9]. One disadvantage of this approach in comparison with the

approach presented in this report is a potential lack of repair effectiveness — there is no guarantee that the external constraints correctly implement the model repairs, and therefore no guarantee that the concrete data structures will be consistent after repair. Another disadvantage is that it requires the programmer to provide the specifications, as opposed to the technique presented in this report, which obtains the specifications automatically.

6.7 File Systems

Some journaling or log-structured file systems are always consistent on the disk, eliminating the possibility of file system corruption caused by a system crash [21, 28]. Data structure repair remains valuable even for these systems in that it can enable the system to recover from file system corruption caused by other sources such as software errors or disk hardware damage.

References

[1] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS-VIII*, pages 110–115, May 2001.

[2] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems*, 19(3), September 1994.

[3] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Very Large Data Bases*, pages 566–577, 1990.

[4] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the SIGPLAN '02 Conference on Programming Languages Design and Implementation*, 2002.

[5] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on the Principles of Programming Languages*, pages 238–252, 1977.

[7] D. Litman and A. Mishra and P. Patel-Schneider. Modeling dynamic collections of interdependent objects using path-based rules. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1997.

[8] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the SIGPLAN '02 Conference on Programming Languages Design and Implementation*, 2002.

[9] B. Demsky and M. Rinard. Automatic data structure repair for self-healing systems. In *First Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[10] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2003.

[11] B. Demsky and M. Rinard. Static specification analysis for termination of specification-based data structure repair. *In International Symposium on Software Reliability Engineering*, November 2003.

[12] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (PLDI), 2003.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. 27(2):1–25, Feb. 2001.

[14] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. pages 449–458, 2000.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[16] T. Griffin, H. Trickey, and C. Tuckey. Generating update constraints from PRL5.0 specifications. In *Preliminary report presented at AT&T Database Day*, September 1992.

[17] N. Gupta, L. Jagadeesan, E. Koutsofios, and D. Weiss. Auditdraw: Generating audits the FAST way. In *Proceedings of the 19th International Conference on Software Engineering*, 1997.

[18] S. Hallem, B. Chelf, Y. Xie, and D. E. r. A system and language for building system-specific, static analyses. In *Proceedings of the SIGPLAN '02 Conference on Programming Languages Design and Implementation*, 2002.

[19] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.

[20] R. Hoover. Incremental computation as a programming abstraction. In *Proceedings of the SIGPLAN '92 Conference on Programming Languages Design and Implementation*, 1992.

[21] M. K. Johnson. Whitepaper: Red Hat's new journaling file system: ext3. http://www.redhat.com/support/wpapers/redhat/ext3/index.html, 2001.

[22] D. A. Ladd and J. C. Ramming. Two application languages in software production. In *Proceedings of the 1994* USENIX *Symposium on Very High Level Language*, October 1994.

[23] G. Lopez. *The Design and Implementation of Kaleidoscope, A Constraint Imperative Programming Language.* PhD thesis, University of Washington, April 1997.

[24] A. Mishra, J. Ros, A. Singhal, G. Weiss, D. Litman, P. . Patel-Schneider, D. Dvorak, and J. Crawford. R++: Using rules in object-oriented designs. *In Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, July 1996.

[25] S. Mourad and D. Andrews. On the reliability of the IBM MVS/XA operating system. *Transactions on Software Engineering,* September 1987.

[26] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, March 15, 2002.

[27] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation,* 2004.

[28] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *In Symposium on Operating Systems Principles,* Oct. 1991.

[29] S. D. Urban and L. M. Delcambre. Constraint analysis: A design process for specifying operations on objects. *IEEE Transactions on Knowledge and Data Engineering,* 2(4), December 1990.

[30] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. *In Proceedings of the Year 2000 Network and Distributed System Security Symposium,* 2000.