Inference and enforcement of data structure consistency specifications

Brian Demsky[1], Michael D. Ernst[2], Philip J. Guo[2],
Stephen McCamant[2], Jeff H. Perkins[2], Martin Rinard[2]

[1]University of California at Irvine, Irvine, CA, USA
[2]MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA, USA
bdemsky@uci.edu, {mernst,pgbovine,smcc,jhp,rinard}@mit.edu

ABSTRACT

Corrupt data structures are an important cause of unacceptable program execution. Data structure repair (which eliminates inconsistencies by updating corrupt data structures to conform to consistency constraints) promises to enable many programs to continue to execute acceptably in the face of otherwise fatal data structure corruption errors. A key issue is obtaining an accurate and comprehensive data structure consistency specification.

We present a new technique for obtaining data structure consistency specifications for data structure repair. Instead of requiring the developer to manually generate such specifications, our approach automatically generates candidate data structure consistency properties using the Daikon invariant detection tool. The developer then reviews these properties, potentially rejecting or generalizing overly specific properties to obtain a specification suitable for automatic enforcement via data structure repair.

We have implemented this approach and applied it to three sizable benchmark programs: CTAS (an air-traffic control system), BIND (a widely-used Internet name server) and Freeciv (an interactive game). Our results indicate that (1) automatic constraint generation produces constraints that enable programs to execute successfully through data structure consistency errors, (2) compared to manual specification, automatic generation can produce more comprehensive sets of constraints that cover a larger range of data structure consistency properties, and (3) reviewing the properties is relatively straightforward and requires substantially less programmer effort than manual generation, primarily because it reduces the need to examine the program text to understand its operation and extract the relevant consistency constraints. Moreover, when evaluated by a hostile third party "Red Team" contracted to evaluate the effectiveness of the technique, our data structure inference and enforcement tools successfully prevented several otherwise fatal attacks.

ISSTA'06, July 17–20, 2006, Portland, Maine, USA.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging;
D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Experimentation, Languages, Reliability

Keywords

dynamic invariant detection, data structure repair

1. INTRODUCTION

Data structure consistency is a crucial program property: corrupt data structures can cause a program to behave unacceptably or even fail. Approaches to ensuring data structure consistency have included program testing and monitoring techniques to detect and localize errors (which the developer would then correct) [11, 1], rebooting to clear corrupted volatile data structures [2], dynamic checks to roll back transactions that corrupt data structures [36], and static analysis to verify that the program is free of any errors that may corrupt the data structures [37, 26].

We have recently developed a new approach — data structure repair [13]. Given a specification of data structure consistency properties and a data structure that violates these properties, our data structure repair algorithm updates the data structure so that it satisfies the properties. This technique can enable programs to continue to execute successfully through otherwise fatal data structure corruption errors [13]. It is especially useful for repairing nonvolatile data structures that persist through system restarts (corrupting such data structures can completely disable a system), for enabling systems to remain operational in the face of recurrent problematic inputs that trigger data structure inconsistency errors, and for situations in which it is undesirable or unacceptable to restart the system.

In previous work, the developer must manually generate the data structure consistency specifications [12, 13]. A potential drawback of this approach is that the developer may inadvertently produce incorrect or incomplete specifications. Incorrect specifications may introduce new data structure corruption errors into an otherwise correct program. Incomplete specifications may leave the program vulnerable to otherwise repairable data structure corruption errors.

This paper presents a new technique using the Daikon invariant detection tool to (1) reduce or even eliminate developer effort in constructing data structure consistency specifications, (2) reduce

the possibility that the developer may produce incorrect data structure consistency specifications, and (3) increase the coverage of the data structure consistency specifications. The basic idea is as follows:

- **Dynamic Analysis:** The developer runs the program on a range of test inputs for which the program is known to produce correct results. The Daikon dynamic analysis tool processes the test runs to produce a candidate set of data structure consistency specifications.

- **Review:** The generated data structure consistency specifications are satisfied in all of the sample executions, but may be overly specific to the test inputs. The developer may therefore review the generated specifications to discard or generalize any overly specific properties.

- **Translation:** Our tools translate the generated consistency properties from the Daikon specification language into the repair system's data structure consistency specification language. Whereas Daikon reports properties over the concrete variables and object fields of the program, the data structure repair system works with an abstract model of the state (the abstract model simplifies the repair algorithm). Our tools bridge this abstraction gap.

- **Monitoring and Repair:** Our tools augment the program with automatically generated code that implements our data structure repair algorithm. As the program runs, this code monitors its data structures for consistency property violations, repairing any violations as necessary to keep the program executing acceptably.

We have applied our technique to three software systems: the CTAS air-traffic control system (Section 2), the BIND Domain Name Service (DNS) server (Section 4.2), and the Freeciv interactive game (Section 4.3).

As the most widely used DNS server on the Internet, BIND is the target of many security attacks. Our technique was able to ameliorate or eliminate two known security vulnerabilities in BIND.

Freeciv is an interactive multi-player game. Our evaluation of our technique as applied to Freeciv included an adversarial Red Team exercise in which an outside organization attempted to develop data structure corruption attacks that would cause Freeciv to fail. Our results show that our techniques significantly reduced the vulnerability of Freeciv to these attacks.

The evaluations of BIND and Freeciv both use the automatically generated data structure consistency specifications with no developer modifications. Because of earlier development efforts, we also have a manually generated specification for Freeciv. In comparison with our manually generated specification, the automatically generated specification is more comprehensive (i.e., covers a wider range of important data structure consistency properties) and equally understandable to the developer. For Freeciv, automatically generating the data structure consistency specification decreases the engineering effort required to use data structure repair and increases the coverage of the resulting data structure consistency specifications (and therefore the overall effectiveness of data structure repair).

This paper makes the following contributions:

- **Generation:** It shows how to automatically generate data structure consistency specifications, then use these specifications to enable data structure repair to deliver more robust, resilient programs.

- **Translation:** It shows how to automatically translate specifications from the concrete domain of the program into the abstract domain of the repair tool.

- **Comparison:** It compares the automatically generated consistency specifications with the manually generated specifications. This comparison indicates that the automatically generated properties are more comprehensive and equally understandable.

- **Evaluation:** It presents our experience using unmodified automatically generated specifications to repair data structure corruptions in the BIND and Freeciv software systems. In both cases, the specifications enabled the repair algorithm to keep the systems executing through otherwise fatal corruptions.

The remainder of the paper is structured as follows. Section 2 presents an example that illustrates our technique. Section 3 discusses our automatic consistency constraint generation algorithm. Section 4 presents our experience using the combination of automatic constraint generation and data structure repair to enable applications to survive otherwise fatal data structure corruption errors. Section 5 reviews related work, and Section 6 concludes.

2. EXAMPLE

We next present an example that shows how to apply our technique. The example system is the CTAS air-traffic control system, a set of air-traffic control tools developed at the NASA Ames Research Center [8]. Versions of this system are deployed at air-traffic control centers throughout the United States and are in daily operational use. Among other functionality, CTAS maintains a flight plan for each aircraft in a given airspace. The flight plan data structure contains a total of 38 fields including the `category` field (which denotes the type of flight, such as an arriving or departing flight), the `origin` field (which denotes the originating airport), and the `destination` field (which denotes the destination airport).

The first step in using Daikon to generate specifications for data structure repair is to identify the data structures that are of interest. For CTAS, we identified the flight plan data structure to be of interest. Figure 1 presents the specification that Daikon then generates for this data structure. The specification itself consists of a set of properties. The first property requires arriving flights (i.e., flights with the category 2, 4, or 6) to have the destination airport index set to either 0 or 1. The second property requires all other flights to have the destination airport index set to the special "uninitialized" value of -999999. The remaining properties impose similar constraints on the `origin` field and specify all of the legal values of the `category` field.

2.1 Abstract Model

Daikon generates invariants over concrete variables and object fields. The data structure repair system takes as input data structure consistency constraints expressed in terms of an abstract model of the data structure. Our translation algorithm translates the concrete variables in the Daikon properties into expressions over the model's sets and relations. It does so in three steps: generating an abstract model, defining the relations of the model, and producing constraints over the relations.

Figure 2 presents the generated set and relation declarations for CTAS. In this case, the algorithm abstracts the variable `fp` by the singleton set `Sfp0`. The set declaration `set Sfp0(flightplan);` declares the set `Sfp0` to contain pointers to `flightplan` data structures. Similarly, the algorithm abstracts the field `category` by

```
fp.category one of { 2,4,6 } => fp.destination one of { 0,1 }     arriving flights have a destination
fp.category one of { 1,3,5,7,8,9 } => fp.destination == -999999   non-arriving flights have no destination
fp.category one of { 1,6,7 } => fp.origin one of { 0,1 }          departing flights have an origin
fp.category one of { 2,3,4,5,8,9 } => fp.origin == -999999        non-departing flights have no origin
fp.category one of { 1,2,3,4,5,6,7,8,9 }                          all possible types of flights
```

Figure 1: Partial Daikon dynamic analysis output for the CTAS air traffic controller example.

```
set Sfp0(flightplan);
Rcategory1: Sfp0->int;
Rdestination3: Sfp0->int;
Rorigin5: Sfp0->int;

true => fp in Sfp0;
forall s in Sfp0, true =>
  <s,s.category> in Rcategory1;
forall s in Sfp0, true =>
  <s,s.destination> in Rdestination3;
forall s in Sfp0, true =>
  <s,s.origin> in Rorigin5;
```

Figure 2: Abstract model automatically generated from the CTAS air traffic control code. The set and relation declarations appear at the top of the figure, and the model definition rules appear at the bottom.

the relation `Rcategory1`, the field `destination` by the relation `Rdestination3`, and the field `origin` by the relation `Rorigin5`. The declaration `Rcategory1: Sfp0->int` defines the relation `Rcategory1` to map objects in the set `Sfp0` to integers.

The algorithm then generates model definition rules to define a translation from the concrete data structure to the abstract model. Figure 2 gives the model definition rules for the example. The model definition rules are evaluated to construct the set and relations in the abstract model. A model definition rule consists of optional quantifiers, followed by a guard condition on the concrete data structure (in this example, the guard is always `true`), followed by a set or relation inclusion condition. The model construction process evaluates the model definition rules for each possible binding of the quantifiers. If the guard of the model definition rule is true for a particular binding, then the model process adds the object (or tuple) to the set (or relation) specified in the inclusion condition. The first model definition rule `true => fp in Sfp0` constructs a singleton set `Sfp0` that contains the flight plan object. The remaining model definition rules construct the `Rcategory1`, `Rdestination3`, and `Rorigin5` relations that abstract the `category`, `destination`, and `origin` fields, respectively. These relations map the flight plan object in `Sfp0` to the value of the field that the relation abstracts.

2.2 Model Constraints

Given the abstract model of Section 2.1, the algorithm translates the Daikon invariants into model constraints. The first step of this translation is to translate the Daikon variable names (which are typically the same as variable names in the target program) into equivalent expressions on the sets and relations in the abstract model. For example, the algorithm translates the Daikon variable `fp` into the quantifier variable `fp7` over the singleton set `Sfp0`. It then uses `fp7` to translate the Daikon variable `fp.category` into the expression `fp7.Rcategory1`.

Then the translation algorithm generates a model constraint with the equivalent meaning as the Daikon invariant. In this example, the `one of` Daikon invariant is translated into a disjunction of the different possible equalities, and the implication in Daikon invariant is translated into an equivalent boolean expression. Figure 3 gives the translated model constraints for the example.

```
forall fp7 in Sfp0, (!(fp7.Rcategory1=2 or
  fp7.Rcategory1=4 or fp7.Rcategory1=6)) or
  fp7.Rdestination3=0 or
  fp7.Rdestination3=1;
forall fp0 in Sfp0, (!(fp0.Rcategory1=1 or
  fp0.Rcategory1=3 or fp0.Rcategory1=5 or
  fp0.Rcategory1=7 or fp0.Rcategory1=8 or
  fp0.Rcategory1=9)) or
  fp0.Rdestination3=-999999;
forall fp2 in Sfp0, (!(fp2.Rcategory1=1 or
  fp2.Rcategory1=6 or fp2.Rcategory1=7)) or
  fp2.Rorigin5=0 or fp2.Rorigin5=1;
forall fp5 in Sfp0, (!(fp5.Rcategory1=2 or
  fp5.Rcategory1=3 or fp5.Rcategory1=4 or
  fp5.Rcategory1=5 or fp5.Rcategory1=8 or
  fp5.Rcategory1=9)) or
  fp5.Rorigin5=-999999;
forall fp21 in Sfp0, (fp21.Rcategory1=1 or
  fp21.Rcategory1=2 or fp21.Rcategory1=3 or
  fp21.Rcategory1=4 or fp21.Rcategory1=5 or
  fp21.Rcategory1=6 or fp21.Rcategory1=7 or
  fp21.Rcategory1=8 or fp21.Rcategory1=9);
```

Figure 3: Model constraints corresponding to the 5 properties of Figure 1.

2.3 Monitoring and Repair

Our repair algorithm generator takes as input the abstract model and the model consistency constraints and generates as output an algorithm that automatically monitors the data structures to find and repair any inconsistencies [13]. The actual monitoring and repair can take place either at specified places in the execution or when the program recognizes that it has encountered an error (for example, an addressing error or null dereference error).

We used fault insertion to mimic the effect of errors in the flight plan processing (earlier versions of CTAS contained at least one such error). These errors produce illegal values in the flight plan data structures.

Our workload consisted of a recorded midday radar feed from the Dallas–Ft. Worth center. Without repair, CTAS fails because of an addressing exception (leaving the controller completely without any of the CTAS functionality). With repair, CTAS continues to execute in a largely acceptable state. Specifically, the effect of the repair is to potentially change the origin or destination airport of the aircraft with the faulty flight plan. Even with this change, continued operation is clearly a better alternative than failure. First, one of the primary purposes of the system, visualizing aircraft flow, is left completely unaffected by the repair. Second, only the origin or destination airport of the plane whose flight plan triggered the error is affected — CTAS processes all other aircraft with no errors at all. Finally, augmenting our system to graphically highlight re-

paired values would alert the air-traffic controller to any potentially unreliable data.

Note that in this situation rebooting CTAS is not a viable option. It takes CTAS several minutes to reacquire the flight plans and radar data. Moreover, when CTAS reacquires the problematic flight plan that triggered the data structure corruption error, it will simply fail again before it comes completely up. Without repair, a simple data structure corruption error can render this version of CTAS completely inoperable.

3. SPECIFICATION GENERATION

We have implemented the automatic specification generation tool as a new output format for Daikon. This output format has been distributed as a part of Daikon since Daikon version 3.1.4, released on October 1, 2004.

The primary challenge in generating a specification is to bridge the gap between the abstractions used by the two tools. Daikon generates invariants in terms of concrete program variable names. The repair system uses consistency properties that are specified in terms of an abstract model of the concrete data structure. This model represents data structures in terms of sets of objects and relations between these objects. The consistency specification uses a set of model definition rules to specify a translation between the concrete data structure and the abstract model.

3.1 Generating an Abstraction

Our algorithm constructs a straightforward abstraction: for each Daikon variable the algorithm constructs a set that contains the objects that the Daikon variable refers to, and for each field in the Daikon variable the algorithm constructs a relation that maps the object containing the field to the value or object referenced by the field.

For example, transforming the Daikon expression `m.tiles` requires three steps. First, the algorithm constructs a singleton set `M` that contains the value of `m`. To construct this set, it generates the model definition rule: `true => m in M`. Second, the algorithm constructs the relation `RTiles` to model the field `tiles`. To construct this relation, it generates the model definition rule: `forall m in M, true => <m, m.tile> in RTiles`. Third, the algorithm constructs the set `STiles` to model the values of `m.tiles`. To construct this set, it generates the model rule: `forall m in M, true => m.tile in STiles`.

3.1.1 Local and Global Variables

The model constraint language only allows algebraic constraints on relations. These relations map objects or primitive values to other objects or primitive values. In the case of constraints on local or global variables, there is no object to apply the relation to. In this case, the algorithm constructs a singleton set containing a special value. The value in this singleton set is mapped by relations to the values of local or global variables.

3.1.2 Arrays

The algorithm treats arrays of primitive values specially by constructing a relation to model the array. This relation maps the index of the array to the value of the corresponding element. For example, for the Daikon invariant `x.array[] > 0`, the translation algorithm would construct a relation `Array` that maps a natural number to the corresponding array element. To construct this relation, the algorithm would generate the model definition rule: `forall x in X, for i=0 to x.array.length, true => <i,x.array[i]> in Array`. The translation algorithm would then translate the Daikon invariant into the model constraint `for i=0 to x.array.length, i.Array > 0`.

3.2 Translating Constraints

The model constraint language allows algebraic constraints on relations. To translate most Daikon invariants, the algorithm translates the Daikon variables into the evaluation of a relation on a quantified variable. The algorithm then adds the appropriate quantifier for this variable. For example, the algorithm would translate the invariant `m.tiles.elem[].terrain>=0` into the model constraint `forall q in SElem,q.Terrain>=0`, where the relation `Terrain` models the `terrain` field and the set `SElem` contains the contents of `m.tiles.elem[]`.

3.2.1 Non-null Invariants

The set and relation abstraction used by the repair tool does not allow null values to appear in a set or relation. As a result, an invariant of the form `p!=null` that states that a variable `p` is not equal to null must be translated to a constraint that specifies that the size of the set that abstracts the variable is greater than 0. For example, the translation step would translate this invariant into the model definition rule `p != null => p in P`, which constructs the set `P`, and the model constraint `sizeof(P)>0`, which ensures that the set `P` is not empty.

3.2.2 Sequence–Index Invariants

Sequence–index invariants involve a comparison between a number (to be used as an array index) and an array element indexed by that number. For other types of invariants, there is a degree of freedom in choosing the set to quantify over. For sequence–index invariants, the constraint must quantify over the array index because the value of the variable is compared to the array index. For example, the algorithm would translate the invariant `x.array[i].value>i` into the model constraint `forall i in indexset, i.Array.Value>i`, where the `Array` relation maps an integer `i` in the `indexset` to the value of `x.array[i]`.

3.3 Extracting Data Structure Layouts

We have developed a tool that extracts the layout of the data structures in a program from Dwarf-2 debugging information, derived from the Fjalar toolkit [19]. The tool automatically generates the data layout specification in the correct format.

4. EXPERIENCE

This section presents our experience 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 ran Daikon to automatically extract a consistency specification for the data structures of interest from this trace. We manually reviewed this consistency specification and (for these systems) found nothing we wanted to change. We next ran the data structure repair tool to compile the consistency specifications into C code that detects and repairs any inconsistencies in the data structures, then augmented 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 cor-

ruptions. For BIND, we used a workload that exercised known data structure corruption errors. For Freeciv, we used fault injection to corrupt the data structures, and a Red Team contracted to test the system did likewise. 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 completely 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 the Internet service responsible 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 that 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 at `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 [32, 35, 14], which have also become less common in recent BIND versions, perhaps because of more careful coding practices and auditing.

Our evaluation considers attacks based on higher-level data structure changes, which represent a greater proportion of recent vulnerabilities.[1] We selected two previously-discovered (and -corrected) problems: a "negative caching bug" (section 4.2.2) and an "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.

[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.

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 `qq.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.

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 the authoritative server for `attacker.com`, then when the victim queries that domain's server for the address of `mailserver.attacker.com`, the attacker can send 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 `mailserver.attacker.com` to a host 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.

4.2.2.1 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.

4.2.2.2 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).

4.2.3.1 Obtaining Specifications.

We selected fields 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 to 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 function `isc__rdatalist_first()` 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.

```
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 property, `nsecset.type == 47`, indicates that the type of the record set passed to `nsecnoexistnodata()` must always be 47, which denotes an NSEC record. A second property, `rdataset.private1.rdata.head != null` $\Longrightarrow$ `rdataset.type == rdataset.private1.rdata.head.type`, indicates that if a record set of the "rdatalist" class contains a record, that record's type must be the same as the type of the record set overall.

4.2.3.2 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` $\neq$ `rdataset.type`, and so 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 selected the `civmap`, `tile`, and `map_positions` data structures as being of interest. Together, these data structures hold most of the game's state. 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 a non-interactive 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 Manual Specifications

In previous work, we manually generated specifications for the Freeciv program. We found automatically inferring specifications to require less effort, to be more complete, and to be more likely to accurately capture the important consistency properties.

The first advantage is that automatically inferring the specification took less effort. We developed a test suite containing 11 different game configurations; ran Freeciv to generate execution traces for each of these test suites; and 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 generate specifications, we had to deeply understand how Freeciv manipulated the data structures and then write the appropriate invariants. In our experience manually generating the specification required significantly more effort than obtaining the specification automatically via Daikon. Manually generating a specification for Freeciv that contained only 6 constraints took a few days, and these constraints were limited to the constraints that could be easily inferred from reading the data structure declarations with a basic high level understanding of the game play. In order to use Daikon, we spent one day developing the required test cases. Daikon was then able to infer many more constraints, and many of these constraints would have required us to read a significant amount of code to manually generate them. Note that the original Freeciv specification was written by the developer of the repair system; we imagine that manually generating specifications would be more difficult for novice users and, therefore, they would find even greater benefit.

The second advantage is that the inferred specification includes significantly more invariants than the original manually-generated specification: the inferred specification consists of 21 constraints while the original manually-generated specification only contains 6 constraints. The manually-generated specification contained constraints that ensure that the map data structure and tile array existed, that tiles have valid terrain values (i.e., that the terrain value of a tile is between an upper and lower bound), that cities are referenced by exactly one tile, and that cities are not placed in the ocean. The automatically inferred specification specification contained 13 constraints on 14 fields in the map data structure, 6 constraints on 5 fields (including the terrain constraints) in the tile data structures, and 2 constraints on 2 fields in the start positions data structure. These constraints all ensured that numerical fields had legal values. Many of these constraints were missing from the manually-generated specification, because we were simply unaware of them.

The third advantage is that inferred specifications may be more likely to accurately capture appropriate consistency specifications. 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 two such invariants, which are present in the manually-generated specification: 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; however, our experience indicates that it is much less effort to review than to create specifications.

4.3.3 Overhead of Data Structure Repair

The repair algorithm has two primary sources of overhead: the overhead of checking the consistency properties, and the overhead of wrapping the memory allocation calls in the application to determine which memory addresses are valid. The checking overhead depends on the amount of data that must be abstracted (converted from concrete form to the abstract model) and the complexity of the consistency properties that must be checked on the abstract model. Checking Freeciv's map data structure has a rather large overhead: the algorithm must build a model of the 14 fields and check the 13 consistency properties over these fields for each of the 5,525 tiles.

We benchmarked the repair algorithm on a 3.06 GHz Pentium 4 Linux box. Our repair algorithm takes 10.4 milliseconds to perform a single consistency check, and in the execution of Freeciv it performs 31 such checks. The overall overhead of the consistency checking and memory allocation instrumentation increased the execution time of Freeciv from 1.23 seconds without repair to 1.59 seconds with repair. We expect that actual data structure repairs will remain an infrequent operation, and therefore will not significantly affect the performance of the application. We measured the time taken to check the consistency properties and perform a repair to the terrain field Freeciv to be 24.6 milliseconds.

Building the abstract model and recording which memory addresses are valid imposes a memory overhead. The repair algorithm and memory instrumentation used a maximum of 1,129 kilobytes of memory out of a total of 5,692 kilobytes used by the repair-enabled Freeciv application.

4.3.4 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. Our experiments evaluate a wide range of simultaneous corruptions — we present the repair algorithm with tens to hundreds of simultaneous corruptions. In practice, however, we expect to run the data structure consistency checking and repair algorithm frequently to detect any corruptions quickly after they appear. In this case we would expect the repair algorithm to encounter relatively few simultaneous corruptions.

To gauge success we counted 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

Simultaneous corruptions	Crashes out of 100 executions		
	Original	With repair	Crashes averted
10	68	2	66
20	77	9	68
30	85	13	72
40	95	16	79
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 4: 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.

code. We then applied the same corruptions to both programs and counted the number of times that each one crashed.

Figure 4 reports the results. These results show that, for small numbers of simultaneous corruptions, the repair algorithm is extremely effective at producing data structures that enable the program to continue to execute without crashing. Even when the number of simultaneous corruptions becomes as high as 50 memory locations simultaneously corrupted, the version with repair crashes only 12% of the time. In contrast, the original version crashes 92% of the time.

In general, the number of crashes in both the original version and the version with repair increases 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. Our results show that repair can avert a substantial number of these crashes.

4.3.5 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 the 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 (as well as 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 successfully 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 two main categories. The first 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 second involved corruption of all redundant copies of some information. For example, as illustrated in Figure 4, it is possible to overwhelm the repair system with an enormous amount of lost information.

Several other crash examples highlight kinds of corruptions that our system had some difficulty handling successfully. In one case, the Red Team used 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 (it might be possible to address this problem by invoking the repair algorithm when the program fails in an attempt to resuscitate the program) or the propagation of corruption into so many data structures that successful recovery was not possible. In another case, the Red Team corrupted both of the x and y values for the board map. In this case the repair algorithm was able to use the size of the allocated map data structure to detect the corruption, but was unable to come up with an effective repair — the original x and y values were the only values that enabled the program to continue successfully, and there was not enough information remaining in the corrupted data structures to select the original x and y values from the set of x and y values that satisfied the board size constraints.

4.4 Discussion

The two target 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.

- The Freeciv evaluation 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.

Our case studies focus on specific data structures — particularly the BIND case study. Ideally, the inference and repair system would cover all data structures in a program, and check them frequently for violations. Our tools do not currently scale to that challenge, but the results of our case studies are suggestive of how such a scaled-up system would perform.

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 the repair actions 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 static guarantee of termination in return for handling more properties.

There is no guarantee that the inferred specifications accurately capture appropriate consistency properties; they might be violated by an anomaly rather than a true corruption. This is a general problem with program assertions. Changing a legal data structure to satisfy an incorrect specification could degrade the quality of the target program. Previous research suggests that dynamic analysis, even on modest test suites, is surprisingly accurate. Furthermore, we propose a manual review step, which tends to be much easier and more effective than writing the specifications from scratch. Inferred specifications could be merely checked at first, and the repair code enabled at a later date, when the specifications inspire more confidence.

We chose to use the crash metric as our measure of success because it was an objective, easily measured indication of whether the program satisfied a basic acceptability property (continued execution). We acknowledge that the absence of a crash does not necessarily mean that the program is executing completely acceptably. For example, it is possible for the repair algorithm to produce data structures that do not allow the program to support all of the functionality its users desire. It is even possible for the current repair algorithm to destroy meaningful information as part of the repair process. We saw no evidence of such problems. For example, there were no later crashes, and the Red Team failed to trigger such a situation.

In some cases, crashing may be preferable to some kinds of continued execution. In some contexts external monitoring and intervention is readily available, so it is feasible to respond to data structure consistency violations by crashing and waiting for some external entity to repair any damage and bring the system back up. In other contexts, it may be important for the program to continue to execute successfully through recurrent inputs that trigger data structure corruption errors, it may not be economically desirable to invest in monitoring and recovery systems, it may be unacceptable for the program to go out of service for a significant period of time, or there may be little chance of any repaired data structure anomalies persisting for long periods of time or propagating into persistent data structures. As with any technique, data structure repair should only be applied where it is appropriate. In such situations, data structure repair may enable the delivery of substantially more robust systems that continue to execute successfully after their data structures first corrupted, then repaired. And automatically learning the data structure consistency constraints may substantially reduce the time, effort, and cost required to successfully apply data structure repair.

5. RELATED WORK

We survey related work in invariant inference, software error detection [6, 10, 22, 5], traditional error recovery, manual data structure repair, and databases.

5.1 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 [7]) to obtain semantic properties.

We use the Daikon dynamic invariant detector to generate runtime properties [15]. 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.

The properties are sound over the observed executions but are not guaranteed to be 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 [16, 20].

5.2 Traditional Error Recovery

Reboot, potentially augmented with checkpointing [36], is a traditional approach to error recovery. In the reboot approach, the user simply reboots a crashed or corrupted software system. This returns the system to a known consistent state, the initial state. One drawback of this approach is that all of the volatile state in the software system is lost. Database systems use a combination of logging and replay to avoid the state loss normally associated with rolling back to a previous checkpoint [17]. There has recently been renewed interest in applying many of these classical techniques in

new computational environments such as Internet services [31] and in extending these techniques to reboot a minimal set of components rather than the complete system [2].

5.3 Manual Data Structure Repair

The Lucent 5ESS telephone switch [23, 21, 27, 18] and IBM MVS operating system [30] 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 [17].

5.4 Constraint Programming

Researchers have incorporated constraint mechanisms into programming languages. One such system is Kaleidoscope [28]. 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, and 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 [24]. Rule based programming [29, 9] is a related technique in which the developer defines a test condition and an action to take in response.

5.5 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 [4]. This approach has been extended to enable the system to automatically generate both the triggering components and the repair actions [3]. 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) [34]. Our system supports a broader class of constraints — logical formulas instead of Horn clauses. It also supports constraints that 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.

5.6 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 [25, 33]. 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.

6. CONCLUSION

Automatically generating data structure consistency specifications can reduce or even eliminate the manual specification development overhead previously associated with the use of data structure repair. It can also produce a specification with more comprehensive coverage of important data structure consistency properties and eliminate manual specification development errors.

The key results in this paper are that (1) we were able to automatically obtain data structure consistency specifications, (2) the process of reviewing these specifications was straightforward and required little time, (3) the quality of the automatically generated

specifications was comparable to our manually-generated specifications, and (4) the automatically generated specifications effectively repaired data structure inconsistencies in our benchmarks.

In our experiments, our tools were able to ameliorate the effects of two real denial-of-service attacks on the BIND DNS server, which is a widely deployed piece of critical Internet infrastructure. This successful result validates the technique and the tools and illustrates the potential of combining automatic specification generation with data structure repair. Our technique also enabled Freeciv to recover from corruptions that affected many different memory locations simultaneously in both randomly generated corruptions and for corruptions produced in the adversarial context of a Red Team evaluation.

These results indicate that data structure repair can be an effective technique for enabling programs to recover from data structure corruption errors and that automatically obtaining the necessary data structure consistency specifications by observing program executions can be an effective way to reduce the development effort and potential specification issues such as omitting constraints otherwise associated with the use of data structure repair.

Acknowledgments

This research was funded by DARPA contract FA8750-04-2-0254. We thank Lee Badger for his suggestions and encouragement.

7. REFERENCES

[1] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2002.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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 Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977.

[8] Center-TRACON automation system. `http://www.ctas.arc.nasa.gov/`.

[9] 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.

[10] 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.

[11] B. Demsky, C. Cadar, D. Roy, and M. Rinard. Efficient specification-assisted error localization. In *Proceedings of the Second International Workshop on Dynamic Analysis*, May 2004.

[12] 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.

[13] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[14] 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.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[16] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.

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

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

[19] P. J. Guo. Fjalar: A dynamic analysis framework for C and C++ programs. `http://pag.csail.mit.edu/fjalar/`.

[20] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, Portland, ME, USA, July 18–20, 2006.

[21] 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.

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

[23] 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.

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

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

2001.

[26] V. Kuncak, H. H. Nguyen, and M. Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *20th International Conference on Automated Deduction, CADE-20*, Tallinn, Estonia, July 2005.

[27] 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.

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

[29] 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.

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *Proceedings of the 25th Fault-Tolerant Computing Symposium*, 2005.

[37] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.