Survival Techniques for Computer Programs

Martin Rinard
MIT Computer Science and Artificial Intelligence Laboratory
Singapore-MIT Alliance
32 Vassar Street, 32-G744
Cambridge, MA 02139

***Abstract*— Programs developed with standard techniques often fail when they encounter any of a variety of internal errors. We present a set of techniques that prevent programs from failing and instead enable them to continue to execute even after they encounter otherwise fatal internal errors. Our results indicate that even though the techniques may take the program outside of its anticipated execution envelope, the continued execution often enables the program to provide acceptable results to their users. These techniques may therefore play an important role in making software systems more resilient and reliable in the face of errors.**

I. Introduction

One of the primary reasons that software systems fail is that one of their components encounters an internal error and stops executing. Another important source of failures is exhaustion of resources. The usual consequence of these kinds of failures is that the software system becomes unable to provide acceptable service to its users.

One standard response to this situation is to attempt to eliminate as many errors as possible from the program. Traditional approaches have included debugging tools that help programmers locate the root cause of the unaccepable behavior [20]. More recently many researchers have focused on dynamic and static analyses that may provide insight into various aspects of the behavior of the program or flag errors in the program [14], [7], [19], [16], [9], [28], [27], [23], [15], [12], [17], [18].

In this paper we discuss an alternative and complementary approach — namely, techniques that enable programs to survive various internal errors and continue to execute [25], [24], [10], [11], [22]. Ideally, the continued execution will enable the program to provide acceptable, if in some cases degraded, service to its users. In comparison with standard approaches, which focus on detecting errors, this approach has several advantages:

- **Acceptable Behavior for New Errors:** It may enable a deployed program to continue to execute acceptably without human intervention even when it encounters a previously unknown and otherwise fatal error. This advantage may be especially important for hard real-time programs that control unstable physical phenomena — if the program stops executing, the larger system will usually encounter a disastrous failure.
- **Reduced Programmer Effort:** It may eliminate the need to invest the programmer time and effort otherwise required to find and eliminate the error that caused the program to fail. This property may become especially important when the original development team has moved on to other projects or is unavailable for some other reason.
- **Fewer Introduced Errors:** Any time a programmer modifies the program, there is a substantial chance that the modification may introduce new errors. Even if the modification is correct, it may still damage the design of the program. Techniques that allow programs to automatically recover from internal errors without source code modifications may therefore result in systems with fewer errors and better design.

Our focus is on enabling the program to productively continue regardless of any internal errors it may happen to encounter. We have developed a range of techniques, each suited to a specific class of errors:

- **Failure-Oblivious Computing:** Failure-oblivious computing is designed to enable programs to survive otherwise fatal addressing errors. The basic idea is to perform bounds checks for each access to memory. If the program attempts to perform an out of bounds write, the technique simply discards the write, thereby preventing any corruption of unrelated data. If the program attempts to perform an out of bounds read, the technique manufactures a new value, hands it back to the program as the result of the read, and the program continues to execute along its normal execution path. The net result is that out of bounds accesses no longer cause the program to throw an exception or terminate.
- **Cyclic Memory Allocation:** Programs with memory leaks may fail because they exhaust the available memory resources. There is a simple strategy that is guaranteed to eliminate any memory leak — simply create a fixed-size buffer to hold objects allocated at the leaking site, then allocate objects cyclically out of that buffer. Instead of growing as the program executes, the amount of memory required to hold objects allocated at that site is fixed at the size of the buffer when the program begins its execution. It is possible to generalize this technique to eliminate other kinds of resource leaks such as file handle leaks.
- **Bounded Loops:** Infinite loops are another form of resource exhaustion - the infinite loop consumes the program counter resource, preventing flow of control from reaching other parts of the program. It is possible to eliminate infinite loops by simply imposing an upper

bound on the number of iterations that any loop is allowed to execute. One way of obtaining such bounds is to perform several training executions of the program running on several inputs, observe the number of iterations that each loop executes, then bound the number of iterations for each loop at some function of the largest number of iterations observed during the training executions.

- **Data Structure Repair:** Many programs rely on their data structures to satisfy key consistency properties. When errors cause these data structures to become inconsistent, it can be difficult for the program to continue successfully. Data structure repair is a technique that detects and eliminates any violations of the key consistency properties. It can enable programs to execute successfully in the face of otherwise fatal data structure corruption errors.

One potential issue with all of these techniques is that they keep the program executing, but in a way that the programmer almost certainly did not anticipate. This raises the possibility that the program may exhibit unanticipated or unacceptable behavior. We investigate this issue empirically by applying these techniques and observing the resulting behavior of the program.

II. Failure-Oblivious Computing

The basic idea behind failure-oblivious computing is to transform the program so that it simply ignores any memory errors and continues to execute normally. Specifically, if the program attempts to read an out of bounds array element or use an invalid pointer to read a memory location, the implementation can simply (via any number of mechanisms) manufacture a value to supply to the program as the result of the read, and the program can continue to execute with that value. Similarly, if the program attempts to write a value to an out of bounds array element or use an invalid pointer to write a memory location, the implementation can simply discard the value and continue. We call a computation that uses this strategy a *failure-oblivious* computation, since it is oblivious to its failure to correctly access memory.

Our current implementation of this technique uses a compiler to perform this transformation. For languages such as Java, whose implementations already perform the required bounds checks, it is straightforward to replace the out of bounds exception code with the code that discards writes or manufactures values for reads as appropriate. For languages such as C, whose implementations typically do not perform bounds checks, it may possible to build on a special implementation that already has the checks built in [29], [26]. In our case, we used such a compiler to obtain a failure-oblivious version of C.

It is not immediately clear what will happen when a program uses this strategy to execute through a memory error. We therefore obtained some C programs with known memory errors, and observed the execution of failure-oblivious versions of these programs. Here is a summary of our observations [25], [24]:

- **Acceptable Continued Execution:** We targeted memory errors in servers that correspond to security vulnerabilities as documented at vulnerability tracking web sites [5], [4]. All of these servers share a common computational pattern: they accept a request, perform the requested computation, send the result back to the requestor, then proceed on to service the next request.
  For all of our tested servers, failure-oblivious computing 1) eliminates the security vulnerability and 2) enables the server to execute through the error to continue on to successfully process subsequent requests.
- **Acceptable Performance:** Failure-oblivious computing entails the insertion of dynamic bounds checks into the compiled program. Previous experiments with C compilers that generate code containing bounds checks have indicated that these checks usually cause the program to run less than a factor of two slower than the version without checks, but that in some cases the program may run as much as eight to twelve times slower [29], [26]. Our results are consistent with these previous results. Note that many of our servers implement interactive computations for which the appropriate performance measure is the observed pause times for processing interactive requests. For all of our interactive servers, the application of failure-oblivious computing does not perceptibly increase the pause times.

An obvious question is why failure-oblivious computing produced these results.

A. Reason for Successful Execution

Memory errors can damage a computation in several ways: 1) they can cause the computation to terminate with an addressing exception, 2) they can cause the computation to become stuck in an infinite loop, 3) they can change the flow of control to cause the computation to generate a new and unacceptable interaction sequence (either with the user or with I/O devices), 4) they can corrupt data structures that must be consistent for the remainder of the computation to execute acceptably, or 5) they can cause the computation to produce unacceptable results.

Because failure-oblivious computing intercepts all invalid memory accesses, it eliminates the possibility that the computation may terminate with an addressing exception. It is still possible for the computation to infinite loop, but we have found a sequence of return values for invalid reads that, in practice, appears to eliminate this problem for our server programs. Our servers have simple interaction sequences — read a request, process the request without further interaction, then return the response. As long as the computation that processes the request terminates, control will appropriately flow back to the code that reads the next request and there will be no unacceptable interaction sequences. Discarding invalid writes tends to localize any memory corruption effects. In particular, it prevents an access to one data unit (such as a buffer, array, or allocated memory block) from corrupting another data unit. In practice, this localization protects many critical data structures (such as widely used application data structures or the call stack) that must remain consistent for the program to execute acceptably.

The remaining issue is the potential production of unacceptable results. Manufacturing values for reads clearly has the potential to cause a subcomputation to produce an incorrect or unexpected result. The key question is how (or even if) the incorrect or unexpected result may propagate through the remaining computation to affect the overall results of the program.

All of our initially targeted memory errors eventually boil down to buffer-overrun problems: as it processes a request, the server allocates a fixed-size buffer, then (under certain circumstances) fails to check that the data actually fits into this buffer. An attacker can exploit this error by submitting a request that causes the server to write beyond the bounds of the buffer to overwrite the contents of the stack or heap, typically with injected code that the server then executes. Such attacks are currently the most common source of exploited security vulnerabilities in modern networked computer systems [1]. Estimates place the total cost of such attacks in the billions of dollars annually [2].

Failure-oblivious computing makes a server invulnerable to this kind of attack — the server simply discards the out of bounds writes, preserving the consistency of the call stack and other critical data structures. For two of our servers the memory errors occur in computations and buffers that are irrelevant to the overall results that the server produces for that request. Because failure oblivious computing eliminates any addressing exceptions that would otherwise terminate the computation, the server executes through the irrelevant computation and proceeds on to process the request (and subsequent requests) successfully. For the other servers (in these servers the memory errors occur in relevant computations and buffers) , failure-oblivious computing converts the attack request (which would otherwise trigger a dangerous, unanticipated execution path) into an anticipated invalid input which the server's standard error-handling logic rejects. The server then proceeds on to read and process subsequent requests acceptably.

One of the reasons that failure-oblivious computing works well for our servers is that they have short error propagation distances — an error in the computation for one request tends to have little or no effect on the computation for subsequent requests. By discarding invalid writes, failure-oblivious computing isolates the effect of any memory errors to data local to the computation for the request that triggered the errors. The result is that the server has short data error propagation distances — the errors do not propagate to data structures required to process subsequent requests. The servers also have short control flow error propagation distances: by preventing addressing exceptions from terminating the computation, failure-oblivious computing enables the server to return to a control flow path that leads it back to read and process the next request. Together, these short data and control flow propagation distances ensure that any effects of the memory error quickly work their way out of the computation, leaving the server ready to successfully process subsequent requests.

B. Scope

Our expectation is that failure-oblivious computing will work best with computations, such as servers, that have short error propagation distances. Failure-oblivious computing enables these programs to survive otherwise fatal errors or attacks and to continue on to execute and interact acceptably. Failure-oblivious computing should also be appropriate for multipurpose systems with many components — it can prevent an error in one component from corrupting data in other components and keep the system as a whole operating so that other components can continue to successfully fulfill their purpose in the computation.

Until we develop technology that allows us to track results derived from computations with memory errors, we anticipate that failure-oblivious computing will be less appropriate for programs (such as many numerical computing programs) in which a single error can propagate through to affect much of the computation. We also anticipate that it will be less appropriate for programs in which it is acceptable and convenient to terminate the computation and await external intervention. This situation occurs, for example, during development — the program is typically not producing any useful results and developers with the ability and motivation to find and eliminate any errors are readily available. We therefore see failure-oblivious computing as useful primarily for deployed programs whose users 1) need the results that the program produces and 2) are unable or unwilling to tolerate failures or to find and fix errors in the program.

III. Cyclic Memory Management

The most straightforward application of cyclic memory management applies to m-bounded allocation sites (an allocation site is a location in the program, such as a call to `malloc` or a `new` construct, that allocates memory), which satisfy the property that, at any time during the execution of the program, the program accesses at most only the last m objects allocated at that site. For such allocation sites, the memory manager can simply allocate a buffer large enough to hold m of the objects allocated at that site. For each allocation, it simply returns the next object in the buffer, wrapping around when it reaches the end of the buffer.

To use cyclic memory management, the memory manager must somehow find m-bounded allocation sites and obtain a bound m for each such site. Our implemented technique finds m-bounded sites and estimates the bounds m empirically. Specifically, it runs an instrumented version of the program on a sequence of sample inputs and records, for each allocation site and each input, the bound m observed at that site for that input. Note that in any single execution, every allocation site has a bound m (which may be, for example, simply the number of objects allocated at that site). If the sequence of observed bounds stabilizes at a value m, we assume that the allocation site is m-bounded and use cyclic allocation for that site.

One potential concern is that the bound m observed while processing the sample inputs may, in fact, be too small: other executions may access more objects than the last m objects allocated at the site site. In this case the program may overlay two different live objects in the same memory, potentially causing the program to generate unacceptable results or even fail.

To evaluate our technique, we implemented it and applied it to several sizable programs drawn from the open-source software community. We obtained the following results [22]:

- **Memory Leak Elimination:** Several of our programs contain memory leaks at m-bounded allocation sites. Moreover, some of these memory leaks make the programs vulnerable to denial of service attacks — certain carefully crafted requests cause the program to leak memory every time is processes the request. By presenting the program with a sequence of such requests, an attacker can cause the program to exhaust its address space and fail. Our technique is able to identify these sites, apply cyclic memory allocation, and effectively eliminate the memory leak (and the denial of service attack).
- **Accuracy:** We evaluate the accuracy of our empirical bounds estimation approach by running the programs on two sets of inputs: a training set (which is used to estimate the bounds) and a larger validation set (which is used to determine if any of the estimated bounds is too small). Our results show that this approach is quite accurate: the validation runs agree with the training runs on all but one of the 160 sites that the training runs identify as m-bounded.
- **Reliability:** We also performed a long-term test of the reliability of two of our programs (Squid and Pine) by installing them as part of our standard computing environment. In several months of usage, we observed no deviations from the correct behavior of the programs.
- **Impact of Cyclic Memory Allocation:** In all but one of the programs, the bounds estimates agree with the values observed in the validation runs and the use of cyclic memory allocation has no effect on the observable behavior of the program (other than eliminating memory leaks). Even for the one program with a single bounds estimation error, the resulting overlaying of live objects has no effect on the externally observable behavior of the program during our validation runs. Moreover, an analysis of the potential effect of the overlaying indicates that it will *never* impair the overall functionality of the program.
- **Bounds Reduction Effect:** To further explore the potential impact of an incorrect bounds estimation, we artificially reduced the estimated bounds at each m-bounded site with $m > 1$ and observed the effect that this artificial reduction had on the program's behavior. In some cases the reduction did not affect the observed behavior of the program at all; in other cases it impaired some of the program's functionality. But the reductions never caused a program to fail and in fact left the program able to execute code that accessed the overlaid objects to continue on to acceptably deliver the remaining functionality.

A. Squid

We illustrate how our technique works by discussing its application to the Squid Web proxy cache [6]. Squid supports a variety of protocols including HTTP, FTP, and, for management and administration, SNMP. We performed our evaluation with Squid Version 2.4STABLE3, which consists of 104,573 lines of C code.

Squid has a memory leak in the SNMP module; this memory leak makes squid vulnerable to a denial of service attack [3]. Our training runs indicate that the allocation site involved in the leak is an m-bounded site with m=1. The use of cyclic allocation for this site eliminates the leak.

For Squid, the training runs find a total of three m-bounded allocation sites with m greater than one. To better evaluate the potential effects that might result from an incorrect estimate of the bounds m, we artificially reduce reduce the bounds at these sites, run the program, and observe the results.

The first site we consider holds metadata for cached HTTP objects; the metadata and HTTP objects are stored separately. When we reduce the bound m at this site from 3 to 2, the MD5 signature of one of the cached objects is overwritten by the MD5 signature of another cached object. When Squid is asked to return the original cached object, it determines that the MD5 signature is incorrect and refetches the object. The net effect is that some of the time Squid fetches an object even though it has the object available locally; an increased access time is the only potential effect.

The next site we consider holds the command field for the PDU structure, which controls the action that Squid takes in response to an SNMP query. When we reduce the bound m from 2 to 1, the command field of the structure is overwritten to a value that does not correspond to any valid SNMP query. The procedure that processes the command determines that the command is not valid and returns a null response. The net effect is that Squid is no longer able to respond to any SNMP query at all. Squid still, however, processes all other kinds of requests without any problems at all.

The next site we consider holds the values of some SNMP variables. When we reduce the bound m from 2 to 1, some of these values are overwritten by other values. The net effect is that Squid sometimes returns incorrect values in response to SNMP queries. Squid's ability to process other requests remains completely unimpaired.

These results illustrate that if cyclic memory allocation overlays live data, the program may lose part of its functionality. Nevertheless, the technique enables Squid to continue to execute to provide its users with the remaining functionality. In particular, cyclic memory allocation eliminates the memory leak that otherwise makes Squid vulnerable to a denial of service attack.

B. Usage Scenarios

Our results indicate that cyclic memory allocation with empirically estimated bounds may provide a simple, intriguing alternative to the use of standard memory management approaches for m-bounded sites. It eliminates the need for the programmer to either explicitly manage allocation and deallocation or to eliminate all references to objects that the program will no longer access. Unlike previously proposed approaches [18], [13], [8], [14], [7], [19], [16], [9], [28], that simply identify leaks and rely on the programmer to modify the program to eliminate any detected memory leaks, it automatically eliminates the leak without the need for any programmer intervention. Unlike approaches that analyze object reachability to reason indirectly about memory leaks [18],

[13], [14], [7], [19], [16], [9], [28]; it reasons about the accesses that the program performs and is therefore capable of recognizing and eliminating leaks even when the leaked object remains reachable. It is therefore appropriate for both garbage collected languages and languages with explicit memory management. One particularly interesting aspect of our results is the indication that it is possible, in some circumstances, to overlay live objects without unacceptably altering the behavior of the program.

We anticipate that our technique will be prove to be most useful for eliminating leaks in deployed programs, especially when the original developers are not easily available or responsive. Because it is automatic, the technique can successfully eliminate leaks without requiring anyone to understand and modify the program. It is also possible to apply the technique directly to stripped binaries, making it possible to eliminate leaks even when there is no realistic possibility of understanding the program or modifying its source. In this kind of scenario, it is hard to imagine *any* technique that requires programmer intervention successfully eliminating the leak.

During active development, programmers may prefer to use the extracted memory access information to find leaks that they then eliminate by modifying the program source. Or, if they convince themselves that the bounds m are accurate, they can simply use cyclic memory management at the corresponding allocation sites. Note that this last alternative can significantly reduce the programming burden — it eliminates the need for the programmer to explicitly deallocate objects allocated at m-bounded allocation sites (if the program uses explicit allocation and deallocation) or to track down and eliminate all references to objects that the program will not access in the future (if the program uses garbage collection).

C. Other Resource Leaks

It is, of course, possible for programs to leak other resources such as file descriptors or processes. These kinds of leaks can have equally disabling effects — a program that exhausts its supply of file descriptors may be unable to write an output file or create a socket to write a result to a client; a server that exhausts its supply of processes may be unable to spawn a thread to service an incoming request. It would be straightforward to generalize the memory leak elimination technique in this paper to eliminate these kinds of resource leaks as well — the program could respond to its inability to allocate a new file descriptor or process by simply reusing an existing file descriptor or process. Potential options include cyclic and least-recently-used reallocation.

D. Risk/Reward Analysis for Unsound Transformations

To take a broader perspective, the research suggests that the field may well benefit from exploring a new class of program transformation techniques that trade off soundness in return for other benefits (such as the elimination of memory leaks). In such cases, as with any engineering tradeoff, one must perform a risk/reward analysis to determine if the reward outweighs the risks. Our results indicate that the risks for cyclic memory allocation are apparently quite small. Specifically, they indicate that the bounds estimation technique is quite accurate and that the consequences of overlaying live data are usually not too serious. In contrast, the rewards can be significant. Specifically, cyclic memory allocation can eliminate memory leaks that could otherwise limit the lifetime of the program and leave it vulnerable to denial of service attacks. Our expectation is that, over time, researchers will develop many other unsound program transformations for which the rewards (potentially far) outweigh the risks.

IV. Bounded Loops

The basic idea behind bounded loops is to eliminate the possibility that the program counter may get trapped in an infinite loop and fail to move on to service other parts of the program that need to execute for the program to produce acceptable results.

One way to eliminate this possibility is to simply bound the maximum number of iterations any loop is allowed to consume. One way to obtain the bounds is to observe previous terminating executions of the loop and apply a safety factor — in other words, keep track of the maximum number of iterations i each loop has been previously observed to execute, then bound the number of permitted loop iterations at $c \times i$ for some appropriate constant c.

Note that some programs contain a few loops that are intended to execute without any specific bound on the number of iterations. The main event-processing loop in many servers, for example, usually executes without any specific bound. For such loops, the programmer can simply provide an annotation that indicates that the loop should be given whatever number of annotations it wishes without bound.

The remaining question is what bound to use for loops that have no previously observed bound. While there are a variety of strategies one can imagine using, it seems likely that the number of iterations is likely to be correlated throughout the program. So one reasonable bound might, for example, be the sum over all the loops in the program of the largest number of observed iterations for that loop multiplied by an appropriate constant.

V. Data Structure Repair

The previously discussed techniques (failure-oblivious computing, cyclic memory allocation, and bounded loops) are designed to apply to virtually any program in a relatively program-independent manner. Data structure repair, however, depends heavily on the data structure consistency constraints in the particular program to which it is applied. Our basic approach is to therefore take a specification of the desired data structure consistency properties and enforce that specification. Note that our goal is not necessarily to restore the data structures to the state in which a (hypothetical) correct program would have left them (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 within its designed operating envelope.

A. Basic Technical Approach

Our approach involves two data structure views: a concrete view at the level of the bits in memory and an abstract view that models the data structures as sets of objects and relations between the objects in those sets [10], [11]. The abstract view facilitates both the specification of higher level data structure constraints (especially constraints involving linked data structures) and the reasoning required to repair any inconsistencies.

Each specification contains a set of model definition rules and a set of consistency constraints. Given these rules and constraints, our tool automatically generates algorithms that build the model, inspect the model and the data structures to find violations of the constraints, and repair any such violations. The repair algorithm operates as follows:

- **Model Translation:** It applies the model definition rules to obtain the abstract model of the data structures.
- **Inconsistency Detection:** It evaluates the constraints in the context of the model to find consistency violations.
- **Disjunctive Normal Form:** It converts each violated constraint into disjunctive normal form; i.e., a disjunction of conjunctions of basic propositions. Each basic proposition has a repair action that will make the proposition true. For the constraint to hold, all of the basic propositions in at least one of the conjunctions must hold.
- **Model Repair:** The algorithm selects a violated constraint, chooses one of the conjunctions in that constraint's normal form, then applies repair actions to all of the basic propositions in that conjunction that are false. A repair cost heuristic biases the system toward choosing the repairs that perturb the existing data structures the least.
- **Concrete Repair:** The repair algorithm applies goal-directed reasoning to the model definition rules to come up with a set of modifications to the concrete data structures. This set of modifications implements the synthesized model repairs.
- **Repeat:** Note that the repair actions for one constraint may cause another constraint to become violated. The algorithm therefore repeats the above steps until there are no more violations.

  To ensure that the repair process terminates, our technique preanalyzes the set of constraints to ensure the absence of cyclic repair chains that might result in infinite repair loops. If a specification contains cyclic repair chains, the tool attempts to prune conjunctions to eliminate the cycles.

We have applied this technique to several programs. In general, we have found that it enabled these programs to execute successfully through otherwise fatal errors. We next discuss some of our experience with two of these programs.

B. AbiWord

AbiWord is a full-featured word processing program available at www.abisource.com. It consists of over 360,000 lines of C++ code, and can import and export many file formats including Microsoft Word documents. It uses a piece table data structure to internally represent documents. The piece table contains a doubly-linked list of the document fragments. A consistent piece table contains a reference to both the head and the tail of the doubly linked list of document fragments. A consistent fragment contains a reference to the next fragment in the list and a reference to the previous fragment in the list. Furthermore, a consistent list of fragments contains both a section fragment and a paragraph fragment. We developed a specification for the piece table data structure. Our specification consists of 94 lines, of which 70 contain structure definitions.

To reduce specification overhead, we developed a structure definition extraction tool that uses debugging information in the executable to automatically generate the structure definitions. This tool works for any program that can be compiled with Dwarf-2 debugging information. For AbiWord, we used this tool to automatically generate all of the data structure definitions. The total specification effort for this application therefore consisted of 24 lines of model definition rules and model constraints.

A bug in version 0.9.5 (and all previous versions) of AbiWord causes AbiWord to attempt to append text to a piece table which lacks a section fragment or a paragraph fragment. This bug is triggered by importing certain valid Microsoft Word documents, causing AbiWord to fail with a segmentation violation when the user attempts to load the document. We obtained such a document and used our system to enhance AbiWord with data structure repair as described in this paper. Our experimental results show that data structure repair enables AbiWord to successfully open and manipulate the document. Further inspection reveals that loading this document causes AbiWord to attempt to append text to an (inconsistent) empty fragment list. Our repair algorithm detects the attempt to append text to the empty list and repairs the inconsistency by adding a section fragment and a paragraph fragment, breaking any cycles in the fragment list, connecting the fragments using their `next` fields, pointing the `prev` field of each fragment to the previous fragment, and redirecting the `head` pointer to the beginning of the list and the `tail` pointer to the end of the list. The result of this repair is that AbiWord is able to successfully append the text to the list and continue on to read and edit Word documents without the loss of any information. Without repair, AbiWord fails as it attempts to read in the document.

C. Parallel x86 emulator

The parallel x86 emulator is a software-based x86 emulator that runs x86 binaries on the MIT RAW machine [21]. The x86 emulator uses a tree data structure to cache translations of the x86 code. To efficiently manage the size of the cache, the emulator maintains a variable that stores the current size of the cache. A bug in the tree insertion method, however, causes (under some conditions) the cache management code to add the size of the inserted cache item to this variable twice. When this item is removed, its size is subtracted only once. The net result of inserting and removing such an item is that the computed size of the cache becomes increasingly larger

than the actual size of the cache. The end result is that the emulator eventually crashes when it attempts to remove items from an empty cache.

We developed a specification that ensures that the computed size of the cache is correct. Our specification consists of 110 lines, of which 90 contain structure definitions. When the repair system applies this specification to an instruction cache with an inconsistent size, the net effect is to use redundant information to recompute the correct size of the instruction cache. We tested the impact of this repair by running the gzip compression program on the x86 emulator. Without repair, the emulator stops with a failed assertion. With repair, the emulator successfully executes gzip.

VI. Synergy

We have observed that there is a significant amount of synergy that develops when the techniques are deployed together. Specifically, we have observed that the continued execution after the application of one kind of error recover technique may contain other kinds of errors. So it is often beneficial to apply the techniques together so that the program does not recover from one kind of error only to fail when it encounters another kind of error.

Cyclic memory allocation provides an interesting example of how the techniques may interact. Overlaying objects can easily cause the program to attempt to access out of bounds memory locations or infinite loop. Applying techniques that allow the program to recover from both of these kinds of errors makes the application of cyclic memory management much more robust [22].

VII. Conclusion

Programs developed with standard techniques remain remarkably brittle in the face of faults. We have described a set of techniques that enable programs to continue to execute through such errors. While these techniques may take the program out of its anticipated operating envelope, our results show that they often make it possible for the program to continue on to provide acceptable results to its users. These techniques may therefore become a key component of future reliable and resilient software systems.

Acknowledgements

The research presented in this paper was supported, in part, by the Singapore-MIT alliance, DARPA award FA8750-04-2-0254, and NSF grants CCR00-86154, CCR00-63513, CCR00-73513, CCR-0209075, CCR-0341620, and CCR-0325283.

References

[1] CERT/CC. Advisories 2002. www.cert.org/advisories.

[2] CNN Report on Code Red. www.cnn.com/2001/TECH/internet/08/08/code.red.II/.

[3] CVE-2002-0069. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0069.

[4] SecuriTeam website. www.securiteam.com.

[5] Security Focus website. www.securityfocus.com.

[6] Squid Web Proxy Cache website. http://www.squid-cache.org/.

[7] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic, programming errors. *Software - Practice and Experience*, 2000.

[8] T. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[9] A. Chou. *Static Analysis for Bug Finding in Systems Software*. PhD thesis, Stanford University, 2003.

[10] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 20th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Anaheim, CA, October 2005.

[11] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *27th International Conference on Software Engineering*, St. Louis, MO, May 2005.

[12] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation*, 2000.

[13] Cal Erikson. Memory leak detection in c++. *Linux Journal*, (110), June 2003.

[14] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, June 1996.

[15] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. Interprocedural compatability analysis for static object preallocation. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, January 2003.

[16] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'05)*, January 2005.

[17] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, 2002.

[18] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[19] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

[20] Mark A. Linton. The evolution of dbx. In *USENIX Summer Technical Conference*, pages 211–220, 1990.

[21] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. In *IEEE Micro*, Mar/Apr 2002.

[22] H. Nguyen and M. Rinard. Using cyclic memory allocation to eliminate memory leaks. Technical Report MIT/LCS/TR-1008, Laboratory for Computer Science, Massachusetts Institute of Technology, 2005.

[23] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Mooly Sagiv. Establishing Local Temporal Heap Safety Properties with Applications to Compile-Time Memory Management. In *The 10th Annual International Static Analysis Symposium (SAS '03)*, June 2003.

[24] M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 2004 Annual Computer Security Applications Conference*, Tucson, Arizona, December 2004.

[25] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and Jr. W. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[26] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[27] R. Shaham, E. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in java. In *Proceedings of the International Conference on Compiler Construction (CC '00)*, March-April 2000.

[28] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of ESEC/FSE 2005*, September 2005.

[29] Suan Hsi Yong and Susan Horwitz. Protecting C Programs from Attacks via Invalid Pointer Dereferences. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003.