Automatic Runtime Error Repair and Containment via Recovery Shepherding

Fan Long Stelios Sidiroglou-Douskos Martin Rinard
{fanl, stelios, rinard}@csail.mit.edu
MIT EECS & CSAIL

Abstract

We present a system, RCV, for enabling software applications to survive divide-by-zero and null-dereference errors. RCV operates directly on off-the-shelf, production, stripped x86 binary executables. RCV implements *recovery shepherding*, which attaches to the application process when an error occurs, repairs the execution, tracks the repair effects as the execution continues, contains the repair effects within the application process, and detaches from the process after all repair effects are flushed from the process state. RCV therefore incurs negligible overhead during the normal execution of the application.

We evaluate RCV on all divide-by-zero and null-dereference errors available in the CVE database [2] from January 2011 to March 2013 that 1) provide publicly-available inputs that trigger the error which 2) we were able to use to trigger the reported error in our experimental environment. We collected a total of 18 errors in seven real world applications, Wireshark, the FreeType library , Claws Mail, LibreOffice, GIMP, the PHP interpreter, and Chromium. For 17 of the 18 errors, RCV enables the application to continue to execute to provide acceptable output and service to its users on the error-triggering inputs. For 13 of the 18 errors, the continued RCV execution eventually flushes all of the repair effects and RCV detaches to restore the application to full clean functionality. We perform a manual analysis of the source code relevant to our benchmark errors, which indicates that for 11 of the 18 errors the RCV and later patched versions produce identical or equivalent results on all inputs.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Error handling and recovery; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Corrections

Keywords Divide-by-zero; Null-dereference; Error recovery

1. Introduction

Divide-by-zero and null-dereference errors are an important source of software failures. These errors generate a signal, which is caught by the default signal handler, which then prints an error message and terminates the application. Obvious potential negative consequences include denial of service to users, data corruption, and disasters caused by the failure of large-scale computer-controlled systems. Given these consequences, a technique that enabled applications to successfully survive divide-by-zero and null-dereference errors (and, ideally, other simple errors such as arithmetic overflow errors [17]) would be of enormous value.

We present a new system, RCV, which enables applications to recover from divide-by-zero and null-dereference errors, continue on with their normal execution path, and productively serve the needs of their users despite the presence of such errors. RCV replaces the standard divide-by-zero (SIGFPE) and segmentation violation (SIGSEGV) signal handlers with its own handlers. RCV's divide-by-zero handler manufactures the default value of zero as the result of the divide, then returns back to the application to continue normal execution after the instruction that triggered the divide-by-zero error.

For writes via an address close to zero, RCV's segmentation violation handler discards the write. For reads via an address close to zero, RCV's handler manufactures the default value of zero as the result of the read. For function calls via a function pointer close to zero, the handler skips the call (such calls often correspond to method invocations on a null object). In all cases RCV then returns back to the application to continue normal execution after the instruction that triggered the segmentation violation.

1.1 Error Containment and Recovery Shepherding

Our experience with prior similarly effective recovery and resilience techniques [16, 22, 23, 25, 30, 31] leads us to anticipate concerns that the continued RCV execution may propagate manufactured values (or values influenced by manufactured values) beyond the application to corrupt other processes or persistent data. To address this concern, RCV implements *recovery shepherding*, an influence tracking and error containment system. The influence tracking system tracks all values influenced by the manufactured values that RCV generates. It also detects when all influenced state has either been deallocated or overwritten with clean values.

In the absence of implicit flows, the error containment system prevents influenced values from escaping the recovering application to influence other processes or persistent data. Specifically, RCV can interpret and block any system call that executes after the repair but before the repair effects have been flushed from the system. By providing an enhanced understanding of how the RCV recovery effects may propagate through the system, and by blocking system calls that may be influenced by these effects, recovery shepherding can help provide the reassurance that some may need to confidently deploy RCV.

1.2 Experimental Evaluation

To gain a better understanding of how the continued RCV execution is likely to work out in practice, we evaluate RCV empirically on all divide-by-zero and null-dereference errors available in the

PLDI '14, June 9–11, 2014, Edinburgh, United Kingdom.

ACM 978-1-4503-2784-8/14/06.
http://dx.doi.org/10.1145/2594291.2594337

CVE database [2] from January 2011 to March 2013 that 1) provide publicly-available inputs that trigger the error which 2) we were able to use to trigger the reported error in our experimental environment. We report results for a total of 18 errors in seven real world applications, Wireshark, the FreeType library, Claws Mail, LibreOffice, GIMP, the PHP interpreter, and Chromium. For all but one of these errors, RCV enabled the application, when run on the publicly-available error-triggering input, to produce the identical or equivalent (identical except for error messages) results as a subsequent version with the error eliminated via a developer patch. If we turn off assertions that impede the recovery, RCV produces acceptable results for the last remaining error.

To better understand the effect of RCV on these errors, we perform a source-code based analysis of the applications, the errors, the RCV recovery mechanisms as applied to the errors, and the subsequent developer-provided patches that eliminate the errors. This analysis indicates that 1) for nine of the 18 errors, the RCV and patched versions of the applications produce identical results for all inputs, 2) for two of the remaining nine errors, the RCV and patched versions produce equivalent (identical except for error messages) results for all inputs, and 3) for three of the remaining seven errors, the RCV version produces different but acceptable results in comparison with the patched version. For one of the remaining errors source code is not available.

1.3 Reasons for Success

We attribute much of the success of RCV to a common computational pattern that occurs in all of the benchmark applications. Specifically, all of the applications divide the input into units, then process each unit in turn. Examples of units include packets (Wireshark), fonts (FreeType), email messages (Claws Mail), data sheets (LibreOffice), images (GIMP), PHP statements (PHP interpreter), and web pages (Chromium). The input unit computations are loosely coupled — the computation for one input unit typically has little or no interaction with the computations for other units.

RCV Nullifies the Computation: For 14 of the 18 errors, RCV nullifies the computation of the input unit that triggers the error, in effect turning that computation into a noop and enabling the application to continue on to successfully process subsequent input units. This nullification occurs, for example, by discarding writes via null pointers, skipping method calls on null objects, or causing loops to execute zero iterations.

Nullifying the Computation is the Correct Behavior: Nullifying computations that trigger divide-by-zero and null-dereference errors can be particularly effective because, in many cases, it is the desired correct behavior. Such errors are often triggered by malformed or degenerate (zero sized) input units that the correct patched version of the program is not designed to process and instead simply skips. In other cases the errors are triggered by internal conditions (such as failed memory allocations) that prevent the patched version from successfully processing the input unit at all. In both cases the patched version skips the input unit and the RCV and patched versions produce identical or equivalent behavior.

Other Properties: Our analysis also revealed several other interesting properties of the continued RCV execution:

- **Subsequent Anticipated Error Checks:** The continued RCV execution often eventually encounters a subsequent check for an anticipated error case that the application is coded to handle correctly. Such checks recognize, for example, malformed input units or internal errors such as failed memory allocations or inconsistent data structures. The check detects the error case, correctly executes the recovery code, and RCV has enabled the application to survive long enough to execute correctly.
- **Shallow Errors:** In general, we note that many of the errors are caused by shallow oversights. For example, eight of the 18 errors are caused by missing sanity checks for malformed input units that the application is not designed to process; three of the 18 errors are caused by a failure to check for simple internal errors such as failed memory allocations. The patches for these errors are simple — they all add a check for the missing case and skip the remaining computation for the input unit if the check succeeds. Indeed, the patches for these errors contain, on average, only three lines of code.

  These data are consistent with the hypothesis that most errors in deployed software are shallow, involving developers who simply overlook a (usual trivial) uncommon case that somehow escapes testing. Indeed, only two of the 18 errors involve a logic error or conceptual misunderstanding of the problem.
- **Cascaded Errors:** If an application encounters one divide-by-zero or null-dereference error, it typically encounters more cascaded errors (in some cases hundreds or even thousands of errors) as RCV enables the application to successfully work its way through the computation for the input unit. The fact that the application succeeds despite all of these cascaded errors highlights the ability of RCV to unlock the inherent (but otherwise latent) error resilience present in these applications.
- **Resource Leaks and Cleanup:** Many of the errors occur after the computation has already allocated local resources for processing the input unit. By enabling the application to execute through any cascaded errors to reach the code at the end of the computation that deallocates these local resources, RCV can prevent errors from causing resource leaks.

One of the goals of RCV is to turn fatal errors into benign cases that the program is already coded to handle correctly, for example via an anticipated error check or by doing nothing for zero-sized inputs (see Section 4.2.2). The use of zero as a manufactured value is designed to promote this translation of fatal errors into benign cases — developers often use zero/null to represent missing, erroneous, or zero-sized data or input components.

1.4 The Bigger Picture

In our experience, many researchers expect that such simple repairs as RCV implements are likely to be ineffective in practice. The expectation is that continued execution will prove to be futile because a divide-by-zero or null-dereference error is often just the initial surface manifestation of a more serious corruption from which the application will be unable to recover.

Our results show that this expectation is not accurate. Indeed, our results indicate that the continued RCV execution enables the application to execute acceptably despite the presence of our benchmark errors. Moreover, these results are consistent with previously reported results that demonstrate the success of simple error recovery strategies that (like RCV) combine 1) a simple default repair action (for example, discarding out of bounds writes [25], returning manufactured default values for out of bounds reads [25], or jumping out of infinite loops [7, 16]) with 2) continued execution along the normal execution path [7, 16, 21, 22, 25].

Because of this mounting body of evidence, and because of our analysis of the reasons behind the success of these simple recovery strategies, we anticipate that the basic RCV repair philosophy (simple repairs that enable normal continued execution) will generalize to enable broad classes of applications to successfully recover from similar types of otherwise fatal errors (for example, arithmetic overflow errors [17]). Over time, we can hope that developers will come to embrace a more mature, tolerant, and realistic understanding that acknowledges the impressive resilience that is already inherently present in their applications. Ideally, this understanding will then enable developers to appropriately deploy simple recovery mechanisms that unlock this resilience and enable their applications to deliver more of their true value to society.

1.5 Engineering Concerns

RCV operates directly on off-the-shelf, production, stripped x86 binary executables with no need for source code or debugging information. By default, RCV operates in detached mode — the application runs unmodified until it encounters an error and invokes the corresponding RCV signal handler. At that point, RCV attaches to the application, applies the recovery strategy, and injects the monitoring code that enables RCV to track the recovery effects. Once the recovery effects have been flushed from the system, RCV detaches and the application again runs unmodified. Except during recovery and monitoring, the application therefore runs unmodified in detached mode with negligible overhead.

1.6 Contributions

This paper makes the following contributions:

- **Repair and Recovery Techniques:** We present simple repair and recovery techniques for divide-by-zero and null-dereference errors. These errors discard writes via null references and return zero as the result of a divide-by-zero or read via a null reference. They then return back to the application to continue execution along the normal control-flow path.
- **Recovery Shepherding:** We present recovery shepherding, which combines an influence tracking technique and an error containment technique that addresses concerns that the recovery strategy may corrupt other processes or persistent data.
- **Experimental Results:** We present results from experiments that use RCV to enable applications to execute successfully through errors triggered by the publicly available error-triggering inputs from the CVE database. RCV enables the applications to survive 17 of the 18 errors.
- **Source Code Analysis:** We present a manual analysis of the source code relevant to our set of benchmark errors. This analysis highlights the effectiveness of RCV in enabling successful application behavior for all inputs and not just the error-triggering inputs. It also identifies the computational properties that make RCV so effective. The analysis indicates that for 11 of the 18 errors the RCV and later patched versions produce identical or equivalent results on all inputs.

2. Example

We next present an example that illustrates how RCV enables the Chromium browser to recover from a null-dereference error. Chromium is a popular open source web browser that serves as the code base for Google's chrome browser. It is possible to trigger a null-dereference error in Chromium 19.0.131.0 by loading a certain HTML file (CVE-2011-3083).

Figure 1 presents a screen shot of the terminal when we trigger this error. The yellow box in Figure 1 presents the contents of an HTML file, `ftp.html`, that will trigger the error. The second line of the file, which attempts to open a video stream from a malformed FTP link, triggers the error. As highlighted in the red box in Figure 1, opening `ftp.html` with Chromium triggers a null-dereference error that causes Chromium to terminate with a SIGSEGV (segmentation violation) error.

Figure 2 presents simplified Chromium source code that illustrates this error. When Chromium processes `ftp.html`, it creates a `URLRequestFtpJob` object to handle the FTP link. It then calls `URLRequestFtpJob::StartTransaction()` (Figure 2 presents the source code) to start the FTP transaction. The call to `ftp_transaction_factory()` returns null; Chromium then calls the member function `CreateTransaction()` on the returned object without properly checking whether the returned object is null or not (see line 7). This call causes the null-dereference error.

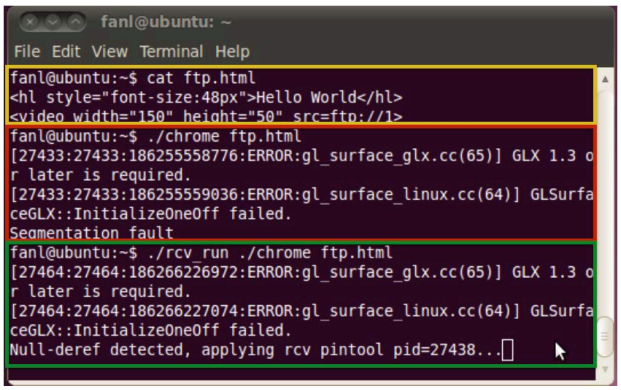

Figure 1. Triggering CVE-2011-3083 with RCV.

```
1  void URLRequestFtpJob::StartTransaction() {
2    // Create a transaction.
3    ...
4    transaction_.reset(
5      /* null-deref error! */
6      request_->context()->ftp_transaction_factory()
7      ->CreateTransaction());
8
9    SetStatus(URLRequestStatus(
10                 URLRequestStatus::IO_PENDING, 0));
11   int rv;
12   if (transaction_.get()) {
13     rv = transaction_->Start(&request_info_,
14      base::Bind(&URLRequestFtpJob::OnStartCompleted,
15                 base::Unretained(this)),
16                 request_->net_log());
17     if (rv == ERR_IO_PENDING)
18       return;
19   } else {
20     /* set up the flag indicating error */
21     rv = ERR_FAILED;
22   }
23 }
```

Figure 2. Simplified Code for CVE-2011-3083.

Figure 3. Chromium-19.0.1031.0 screen shot after continued execution with RCV. The screen shot for the subsequent version with a developer-supplied patch that corrects the error is identical.

Although Chromium has a multi-process architecture that renders each web page in a separate process (thereby minimizing the effect of errors), to avoid performance problems associated with having many open processes [1], Chromium limits the number of rendering processes to 20. Given tab usage patterns in modern browsers [11], the chances of collateral damage caused by an error in a separate tab are high. Despite Chromium's multi-process architecture, this particular null-dereference error occurs in Chromium's main process. The entire application terminates and the user loses any unsaved data, including data in unrelated windows or Chromium tabs.

Running Chromium with RCV. We next show how RCV enables Chromium to successfully survive this error. The green box in Figure 1 shows how to run Chromium with RCV as follows:

- **Start Chromium with RCV:** The user starts Chromium with RCV by running the `rcv_run` script. When Chromium starts, this script uses the `LD_PRELOAD` environment variable to load the RCV runtime shared library. The library sets up signal handlers to intercept signals that Chromium generates when it encounters divide-by-zero or null-dereference errors. The library does not otherwise interfere with the Chromium execution, which runs in native mode with negligible overhead.
- **Start Recovery System:** When the null-dereference error occurs at line 6 in Figure 2, RCV intercepts the signal and starts the RCV recovery shepherding system (see the last line of Figure 1). The recovery shepherding system attaches to the Chromium process that triggered the error to monitor and control the execution of all threads in the attached process.
- **Recovery Execution:** The first null-dereference error is a read from a null address that occurs when Chromium attempts to use the null object returned from `ftp_transaction_factory()` to obtain the address of the invoked virtual member function `CreateTransaction()`. RCV returns the manufactured value zero as the result of the null-dereference. The continued execution immediately encounters another null-dereference error when Chromium interprets the manufactured value as the address of the member function to invoke and calls a function at address zero. RCV skips this call and manufactures zero as the return value. Chromium continues on to encounter other cascaded null-dereference errors, which RCV similarly repairs.
- **Influence Tracking:** When RCV manufactures a return value for the skipped call to `CreateTransaction()`, it tags the value as influenced by the RCV recovery system. RCV then dynamically tracks the flow of influenced values through the program. So, for example, the influence tracking system determines that the manufactured value will influence the value of the reference-counting pointer object `transaction_`.
- **Error Containment:** During the recovery execution, RCV can intercept and skip any system calls which send information beyond the attached process. This strategy can address concerns that the recovery system may transmit influenced values to other processes. In the Chromium experiment specifically, RCV is configured to block all system calls that send information out of the main process except for those that render the GUI display or print information to the terminal.
- **Back to Clean State and Detach:** As the recovery execution continues, Chromium determines that the reference-counting object in `transaction_` holds a null pointer (line 12 in Figure 2). In response, it sets the flag variable `rv` (line 21) to `ERR_FAILED` to indicate that an error has occurred. This error code will be handled by higher level functions in the call stack. If the user removes the focus of this web page in the Chromium GUI (by, for example, switching to the tab of another web page or switching to the window of another application in Ubuntu GUI), Chromium will destroy the `URLRequestFtpJob` object and flush all influenced values from the execution state. At this point the execution has cleanly recovered and RCV detaches.
- **Result:** RCV enables Chromium to execute through the null-dereference error (and subsequent cascading errors) to generate the screen presented in Figure 3 and successfully process subsequent user interactions. Chromium correctly displays the text "Hello World" from `ftp.html`. It also displays a black box in place of the malfunctioning video stream.

Comparison with Developer Patch. The patch that corrects this error initializes the media context object (`request_->context()` in line 6) with a dummy transaction factory object. If the ftp link is empty, the call to `ftp_transaction_factory()` returns the dummy object instead of null. The patched version sets the return value `rv` to `ERR_IO_PENDING` and returns at line 18. The patched version of Chromium later handles `ERR_IO_PENDING` in the same way as the RCV version handles `ERR_FAILED`. The patched and RCV patched versions therefore produce identical results on all inputs. Specifically, for the html page presented in Figure 1, the patched version generates the identical result presented in Figure 3.

3. Design

RCV consists of two parts, a *lightweight runtime monitor* and a *recovery shepherding system*. The lightweight runtime monitor runs together with the application as the application starts. When a divide-by-zero or null-dereference error occurs, it intercepts the generated signal and invokes the recovery shepherding system, providing the recovery shepherding system with the information it needs (e.g., the process id and the heap status of the application) to execute the recovery.

The recovery shepherding system is a just-in-time binary instrumentation library that attaches to the application process and sits in the same address space of the process it may later shepherd. It consists of three components, a *recovery runtime* that implements our recovery strategy, an *influence tracking* system that tracks the effects of our repairs, and an *error containment* system that blocks externally visible side effects of the erroneous process before it reaches a clean execution state. Once the influence tracking system determines that the process has reached a clean execution state (see Section 3.3), the recovery phase ends and the recovery shepherding system detaches from the process.

3.1 Lightweight Runtime Monitor

We implement the RCV lightweight runtime monitor as a shared library. RCV uses the `LD_PRELOAD` Linux environment variable to load the library as the application starts.

Setup Signal Handler: The library initialization routine registers the RCV SIGFPE and SIGSEGV signal handlers. The application generates a SIGFPE signal when it encounters an integer or floating point divide-by-zero error. It generates a SIGSEGV signal when it encounters a null-dereference error.

Intercept Handler Setup Calls: To ensure that the application does not overwrite the RCV SIGFPE and SIGSEGV signal handlers, the runtime monitor wraps functions that set up signal handlers (for example, `signal()` and `sigaction()`), intercepts any attempts to register SIGSEGV and SIGFPE handlers, and records the registered handler. RCV invokes the registered handler only for errors (for example, buffer overflow errors that generate a SIGSEGV signal) that are outside the scope of RCV. In our set of benchmark applications, only Chromium defines its own SIGFPE and SIGSEGV handlers. And these Chromium handlers perform no recovery — like the default signal handlers, they print an error message and terminate the execution.

Intercept Heap Management Calls: To enable the influence tracking system to accurately track the effect of heap deallocation operations, the lightweight runtime monitor wraps heap management functions including `malloc()`, `calloc()`, `realloc()`, `posix_memalign()`, and `free()` to record the mapping between the address and the size of each allocated memory block. It does not otherwise affect the execution of the system heap allocator and therefore incurs negligible runtime overhead.

Invoke the Recovery Shepherding System: When the application encounters a divide-by-zero or null-dereference error, the operating system invokes the corresponding RCV handler. The handler first pauses the execution of all threads in the process in which the error occurred. It then forks a new process to start the RCV recovery

shepherding system. It also transmits the process id and collected heap allocation information to the recovery shepherding system. When the recovery shepherding system successfully attaches to the process that encountered the error, the monitor resumes the execution of all paused threads and allows the recovery system to take control of the execution.

3.2 Recovery Runtime

The recovery runtime in RCV allows the process execution to continue in the presence of runtime errors. The recovery runtime disassembles and analyzes the binary code around the instruction that generated the signal to apply the appropriate repair mechanism. Note that RCV may encounter and repair multiple errors after the failure that initiated the repair and recovery.

Divide by Zero: If the execution of an instruction (e.g., `idiv`) fails due to a divide-by-zero error, the recovery runtime changes the values of the registers that hold the results of the instruction (e.g., `rax` and `rdx` for `idiv`) to a manufactured constant value (RCV empirically uses zero). The recovery runtime then skips the offending instruction and continues the execution at the next instruction by setting the instruction pointer register value (e.g., `rip`) to the start of the next instruction.

Read or Write Null Address: If a memory read or write instruction fails and the accessed memory address is less than a small constant (RCV empirically uses 0x1000), RCV treats the failure as a null-dereference. This mechanism enables RCV to capture null array access like `p[i]` where `p` is a null array pointer and `i` is a non-zero index. For a read instruction to a close-to null address, the recovery runtime returns a manufactured constant value (RCV empirically uses zero) as the result of the read instruction. For a write instruction, the recovery runtime ignores the memory write instruction and continues the execution at the next instruction.

Call to Null Address: If the execution fails because of a call instruction to a null address, the recovery runtime ignores the call and sets the register that holds the return value (e.g., `rax`) to a manufactured constant value (RCV empirically uses zero). It then continues execution after the offending call instruction.

Null Dereference in Loops: RCV handles the case that a thread repeatedly reads and/or writes consecutive close-to null addresses because of null array accesses inside a loop. If the same instruction fails more than five times consecutively, the recovery runtime stops the computation of the current loop and jumps out of the loop [7]. If the RCV binary analysis cannot determine the exit point of the loop, the recovery runtime immediately returns to the function that called the current function. The manufactured return value is zero [30, 31].

Cases to Abort: RCV does not to attempt to repair failures other than divide-by-zero and memory accesses via close-to-null pointers. In particular, the current version of RCV leaves other signal handlers in place and does not attempt to recover from fatal assertion violations.

3.3 Influence Tracking

The influence tracking system in RCV implements a dynamic data flow tracking technique to track how the manufactured values of the repair mechanisms (see Section 3.2) influence the execution state of the recovering process. We implement this tracking system based on the libdft library [13]. For brevity, we focus on the deviation of RCV from the standard dynamic data flow tracking technique.

Track Manufactured Values: Whenever the recovery runtime generates a manufactured value (see Section 3.2), the influence tracking system flags the appropriate bits in the corresponding shadow context or in the memory bitmap for the register or the memory address that will hold this value. As with standard dynamic data flow tracking, RCV inserts callbacks before each instruction and each system call to appropriately track these values.

Deallocation in Stack: Unlike standard dynamic data flow tracking, whenever the stack pointer value increases (i.e., the stack shrinks), RCV clears the influence bits that correspond to the memory region between the old stack pointer value and the new stack pointer value. To detect memory deallocations in the stack during the execution (e.g., when a function exits), RCV inserts callbacks before the instructions that may change the stack pointer register (i.e., `rsp`).

Deallocation in Heap: Unlike standard data flow tracking, whenever the process frees a memory block from the heap, RCV clears the influence bits that correspond to the bytes in the freed memory block. The influence tracking system inserts callbacks before all heap management function calls. The influence tracking system tracks the size of every memory block the process allocates in the heap. Note that the process may free a memory block that the process allocates before the recovery shepherding system attaches to the process. The influence tracking system relies on the information from the lightweight runtime monitor to know the size of the freed memory block (see Section 3.1).

Detach from the Process: The influence tracking system also inserts callbacks to periodically check whether all influence bits are cleared for all threads. If so, the recovery phase ends and the entire recovery shepherding system detaches from the process.

3.4 Error Containment

During the recovery process, the error containment system in RCV intercepts all system calls that could possibly send data and/or messages out of the process (e.g., `write`, `sendmsg`, etc.). RCV determines the type of this communication based on the file descriptor id argument of the intercepted system call and then handles the system call based on the communication type as follows:

Console I/O: If the file descriptor argument of the intercepted system call corresponds to a console device, RCV (by default) allows this system call to continue. It notifies the user via a console warning message if the RCV influence tracking system detects that the repair effects influence the printed data.

GUI I/O: If the file descriptor argument of the intercepted system call corresponds to a domain socket that connects to special GUI service processes (e.g., X-Window processes), RCV will allow this system call to continue by default.

File I/O: If the file descriptor argument of the intercepted system call corresponds to a file in the hard disk, RCV (by default) logs this file write request, then silently skips the file write system call. This mechanism ensures that a process in a potentially erroneous execution state does not corrupt persistent data. The RCV design also supports an extension that would buffer the writes, enable the user to inspect the buffered writes, then commit the writes at the end of the recovery phase if approved by the user (or any other appropriate person or mechanism).

Other Types of Communications: RCV skips other intercepted system call, returning to the process with a manufactured error number that indicates a low level I/O error [30].

3.5 Implementation

We have implemented RCV in more than 15,000 lines of C/C++. The current implementation supports applications running on x86 machines with 64-bit Linux operating system. We implemented the recovery shepherding system based on the PIN binary instrumentation framework [20]. Our implementation of the influence tracking system is based on libdft [13], a dynamic taint tracking system. We have extended libdft to support the 64-bit x86 architecture for our system. We have also implemented a lightweight version of RCV that does not support influence tracking and the error containment. The lightweight version is based on dyninst [6] and libunwind [3].

CVE ID	Application	Type	Location	Survives With RCV	RCV Versus Patched	Error Flush Time	Cascaded Repairs
CVE-2013-2483	Wireshark-1.8.5	Divide	packet-acn.c:1045	Yes	Identical	35s	3
CVE-2012-4286	Wireshark-1.8.1	Divide	pcapng.c:1092	No*	Different*	No	8
CVE-2012-4285	Wireshark-1.8.1	Divide	packet_dcp_etsi.c:273	Yes	Equivalent	41s	3
CVE-2012-1143	FreeType-2.4.8	Divide	ftcalc.cc:555	Yes	Identical†	No	1608
CVE-2012-5668	FreeType-2.4.8	Deref	bdflib.c:2475	Yes	Identical	<1s	6
CVE-2012-4507	Claws-Mail-3.8.1	Deref	procmime.c:1756	Yes	Identical	23s	3
CVE-2012-4233	LibreOffice-3.5.5.2	Deref	libscfiltlo.so	Yes	Identical	44s	12
CVE-2012-3236	GIMP-2.8.0	Deref	fits-io.c:1059	Yes	Equivalent	2s	1
CVE-2012-1593	Wireshark-1.6.5	Deref	packet.c:469	Yes	Equivalent	34s	24
CVE-2012-1128	FreeType-2.4.8	Deref	ttinterp.c:5690	Yes	Identical†	3s	23050
CVE-2012-0781	PHP-5.3.8	Deref	libtidy localize.c:1038	Yes	Identical	<1s	8
CVE-2011-4153	PHP-5.3.8	Deref	zend_constant.c:430	Yes	Equivalent	N/A	34
CVE-2011-3182	PHP-5.3.6	Deref	parse_date.c:354	Yes	Equivalent	N/A	216
CVE-2011-3083	Chromium-19.0.1031.0	Deref	url_request_ftp_job.cc:68	Yes	Identical	19s	20
CVE-2011-2849	Chromium-12.0.747.112	Deref	websocket_job.cc:495	Yes	Identical	10s	3
CVE-2011-1956	Wireshark-1.4.5	Deref	proto.c:1740	Yes	Identical	N/A	90+34
CVE-2011-1691	Chromium-12.0.717.0	Deref	CSSComputedStyled -Declaration.cc:766	Yes	Identical	2s	2
CVE-2011-0421	PHP-5.3.5	Deref	zip_name_locate.c:64	Yes	Identical	<1s	1

*For CVE-2012-4286, the RCV version of Wireshark-1.8.1 terminates with an assertion violation. Disabling assertions and configuring `free()` to ignore frees of invalid memory blocks enables the RCV version to survive the error and produce an acceptable result.

†For CVE-2012-1143 and CVE-2012-1128, both the patched and RCV versions terminate normally and display performance statistics (which vary in different runs) on the console. The two versions produce identical results except for the performance statistics (which vary even for different runs of the same version).

Table 1. Experimental Results for Executions on Error-Triggering Inputs

4. Experimental Results

We evaluate RCV on all divide-by-zero and null-dereference errors in the CVE database [2] from January 2011 to March 2013 that 1) provide publicly-available inputs that trigger the corresponding errors which 2) we were able to use to trigger the reported errors. These errors occur in seven different benchmark applications (as shown in Table 1), Wireshark (an interactive network packet analysis application), the FreeType library (a font processing library), Claws Mail (an interactive email client), LibreOffice (an open source productivity suite), GIMP (an interactive image processing application), the PHP interpreter (the PHP program interpretor), and Chromium (an open source web browser).

4.1 Results on Error-Triggering Inputs

We first evaluate the effectiveness of RCV in enabling the applications to successfully process the publicly-available error-triggering inputs. For the three errors inside the FreeType library, we run the micro benchmark ftbench that comes with the FreeType library to trigger the errors and conduct our experiments. For all errors except the three null-dereference errors in Chromium, we conduct the experiments on an Intel Core i7 2.8GHz machine running 64-bit Ubuntu 12.04 on a virtual machine. Because some of the Chromium versions containing the errors do not build on Ubuntu 12.04, we conduct the experiments for these errors on 64-bit Ubuntu 10.04 running on the same hardware. During our experiments, we observe negligible overhead (less than 1%) for the normal executions of all benchmark applications.

Table 1 summarizes the results. There is one row in the table for each error. The first column (CVE ID) presents the identifier of the error in the CVE database. The second column (Application) presents the name and version number of the application that contains the error. The third column (Type) specifies whether the error is a divide-by-zero error (Divide) or a null-dereference error (Deref). The fourth column (Location) identifies the location in the source code where the error occurs. For CVE-2012-4233, the null-dereference error occurs in a binary shared library (`libscfiltlo.so`) for which the source code is not available.

Survives With RCV. The fifth column (Survives With RCV) indicates whether RCV enables the application to survive the error (Yes) or not (No). For all but one of the errors (CVE-2012-4286), RCV enables the application as released to survive the error and successfully continue normal execution. Either the application terminates normally (CVE-2012-1143, CVE-2012-1128, CVE-2012-0781, CVE-2011-3182, CVE-2011-0421), exits gracefully (CVE-2012-5668, CVE-2011-4153), or remains functional awaiting further user input (the remaining 10 errors). For CVE-2012-4286, continued execution in the released version encounters a fatal assertion violation. Disabling assertions and appropriately configuring libc to ignore invalid memory frees enables the application to survive (see Section 4.3 for more information).

RCV Versus Patched. For all of the errors we were able to obtain a subsequent version of the application with a developer-provided patch that eliminates the error. CVE-2011-4153 (PHP-5.3.8) and CVE-2011-3182 (PHP-5.3.6) remain unpatched in the current stable version of PHP (PHP-5.3.27). For these two errors we therefore use patches that were suggested in the error reports. For each of the other errors we use the next stable version of the respective application in which the error has been patched.

The sixth column of Table 1 (RCV Versus Patched) compares the results from the patched and RCV versions of the application, both when run on the error-triggering input. The two versions either produce identical results (Identical, 12 of 18 errors), identical results except that the two versions generate different error messages (Equivalent, five of 18 errors), or the RCV version produces different output (Different, CVE-2012-4286, see Section 4.3).

In many cases the continued RCV execution encounters a subsequent check for an anticipated error case that the application is coded to handle correctly, typically by generating an error message and skipping the remaining computation for the input unit (see Section 4.2). For three of the five Equivalent errors (CVE-2012-4285, CVE-2012-3236, CVE-2011-4153), the error messages differ because two versions encounter different anticipated error checks and therefore generate different error messages. For CVE-2012-1593, the RCV and the patched versions generate different error messages for the malformed packet in the rightmost information column in the Wireshark GUI. For CVE-2011-3182 (the PHP interpreter), the patched and RCV versions print different values on the console for a failed PHP statement. See Section 4.3 for details.

Error Effect Flush Time. The seventh column (Error Flush Time) presents whether the RCV recovery shepherding system was able to detect that the effects of the error were flushed from the system and, if so, how long it took. An entry of the form Xs (or <Xs) indicates that the application executed under the control of the RCV recovery shepherding system for X (or <X) seconds before RCV was able to verify that the effects had been flushed. The effect flush times, when available, are small — for all errors except CVE-2011-3083 (Chromium-19.0.1031.0), they are less than 50 seconds.

For CVE-2011-4153 (PHP-5.3.8) and CVE-2011-3182 (PHP-5.3.6), it is possible to trigger the null-dereference error only when running with a (restrictive) memory limit of 200 MBytes for the entire application. With this limit, there is not enough memory for libdft [13] (upon which our recovery shepherding system depends) to attach to the application. For CVE-2011-1956 (Wireshark-1.4.5), the effect of the repair propagates into floating point calculations. The current version of RCV does not implement influence tracking for floating point calculations. For these three errors we use the lightweight version of RCV without influence tracking and error containment (see Section 3.5).

For CVE-2011-3083 (Chromium), the error flush time depends on the actions the user takes after RCV enables the browser to survive the error and support further interactive use (see Section 2). We measured the flush time for an execution in which the user loads the error-triggering web page, immediately switches the GUI focus to other applications, and then switches back.

Based on experiments running SPEC CPU 2006 benchmark programs [4] with the RCV recovery system, we measure application performance with the RCV recovery system as (on average) approximately 80 times slower than native execution. We attribute the overhead to the binary instrumentation present in the RCV recovery system. We note that, despite this overhead, the error flush times remain acceptably small. Because the RCV recovery system detaches and restores the application to fully efficient native execution after the effects of the error are flushed, RCV does not significantly impair application usability. We also note that the lightweight version of RCV mentioned above implements the RCV repair mechanisms fully in native mode with no binary instrumentation overhead.

Cascaded Repairs. In many cases RCV enables the application to execute through multiple cascaded errors as the application works its way through the computation for the input unit that triggered the error. The eighth column (Cascaded Repairs) presents how many errors RCV repaired before the application finished processing the error-triggering input. All of the cascaded errors are of the same type — if the first error is a divide-by-zero error, then so are all of the remaining errors, and similarly for null-dereference errors.

For CVE-2011-1956, the error-triggering input contains two error-triggering packets (as well as other benign packets). The first packet triggers 90 errors, the second packet triggers 34 errors, so to enable Wireshark to successfully process the input, RCV repairs a total of 124 errors. For CVE-2011-5668 and CVE-2012-1128, the null-dereference error occurs inside loops. The number of repairs for CVE-2011-5668 is relatively small because RCV jumps out of the loop after repairing the sixth null-dereference error (see Section 3.2). The number of repairs for CVE-2012-1128 is relatively large because it has nested loops and the inner loop only executes one iteration for each error.

4.2 Source Code Analysis

To better understand the interaction between RCV and the benchmark applications, we performed a manual analysis of the source code (when available) surrounding each error. The goal was to understand the computation in which the error occurred, the continued execution under RCV, the developer patch that eliminated the error, and the differences (if any) between the patched and RCV versions.

4.2.1 Basic Computational Pattern

As we analyzed the benchmark applications, it became clear that all of the applications exhibit the following computational pattern:

- **Input Units:** The application divides its input into input units, then processes each input unit in turn. For example, Wireshark divides its input stream into network packets, then performs a computation to process each network packet in turn. The input unit computations are loosely coupled — the computation for one input unit tends to have little or no interaction with the computations for other input units, with any interactions mediated by shared data data structures available to both computations.
- **Sanity Checks:** The computation for each input unit is designed to process well-formed input units that satisfy certain consistency constraints. The computation therefore performs sanity checks that determine if the input unit satisfies these constraints. If a sanity check fails, the application skips the input unit, performs any required cleanup (see below), and proceeds on to process the next input unit.
- **Local Resource Allocation:** The computation for each input unit allocates resources (such as memory) required to perform the computation. The lifetimes of these resources are contained within the computation for that input unit (so the resources should be deallocated when the computation finishes).
- **Process Input Unit:** Using the allocated local resources, the computation processes the input unit.
- **Updates:** Using the computed results, the computation appropriately updates any shared data structures and/or produces any required externally visible output.
- **Cleanup:** The computation for the input unit finishes by performing any required local resource deallocation operations.

Viewing the error, the patch, and the RCV recovery mechanisms through the prism of this pattern can provide significant insight into why RCV works and how well our results are likely to generalize to other errors in other applications.

4.2.2 Analysis Result Summary

Table 2 summarizes the results of our analysis. The table contains a row for each error. Note that we omit the analysis of CVE-2012-4233 because the source code of the library where the error occurs is not available. The first column of Table 2 (CVE ID) presents the CVE ID of the error; the second column (Input Unit) identifies the input units for the application that contains the corresponding error.

Missing Sanity Checks. Our analysis indicates that eight of the 18 errors are caused by missing sanity checks. The error is triggered when the missing sanity check causes the computation to attempt to process a malformed input unit that does not satisfy one or more of the required consistency constraints. The third column of Table 2 (Missing Sanity Check) presents, for each error, whether the error was caused by a missing sanity check (Yes) or not (No).

Patch Skips Computation. For 12 of the 18 errors, our analysis indicates that the patch causes the application to skip the computation for the input unit if the unit would trigger the error, in effect turning that computation into a noop. These 12 errors include all of the errors that were caused by a missing sanity check as well as CVE-2011-4153, CVE-2011-3182, CVE-2011-3083 and CVE-2012-1128. CVE-2011-4153 and CVE-2011-3182 are caused by a failure to check for a null return value from a (failed) memory allocation operation. CVE-2011-3083 is caused by a failure to anticipate a case associated with a null ftp link (see Section 2). CVE-2012-1128 is caused by a logical error that causes the computation to incorrectly attempt to execute one iteration of a specific loop instead of zero iterations (see Section 4.3). The fourth column of Table 2 (Patch Skips Computation) presents whether the patch skips the computation and turns it into a noop (Yes) or not (No).

CVE ID	Input Unit	Missing Sanity Check	Patch Skips Computation	RCV Nullifies Computation	RCV Hits Anticipated Error Check	Cleanup Required	Patched vs. RCV, All Inputs
CVE-2013-2483	Network Packets	Yes	Yes	Yes	No	Yes	Identical
CVE-2012-4286	Network Packets	No	No	No	Yes	Yes	Different
CVE-2012-4285	Network Packets	Yes	Yes	Yes	Yes	Yes	Equivalent
CVE-2012-1143	Font	Yes	Yes	Yes	No	Yes	Different
CVE-2012-5668	Font	No	No	No	No	Yes	Identical
CVE-2012-4507	Emails in Inbox	Yes	Yes	Yes	Yes	Yes	Identical
CVE-2012-4233	Data Sheets	Source code is not available					
CVE-2012-3236	Images	Yes	Yes	Yes	Yes	Yes	Equivalent
CVE-2012-1593	Network Packets	No	No	Yes	No	Yes	Acceptable
CVE-2012-1128	Font	No	Yes	Yes	No	Yes	Identical
CVE-2012-0781	PHP Statements	Yes	Yes	No	No	No	Acceptable
CVE-2011-4153	PHP Statements	No	Yes	Yes	Yes	Yes	Acceptable
CVE-2011-3182	PHP Statements	No	Yes	Yes	No	Yes	Different
CVE-2011-3083	Web Pages	No	Yes	Yes	Yes	Yes	Identical
CVE-2011-2849	Web Pages	No	No	Yes	No	No	Identical
CVE-2011-1956	Network Packets	No	No	Yes	No	No	Identical
CVE-2011-1691	Web Pages	Yes	Yes	Yes	No	Yes	Identical
CVE-2011-0421	PHP Statements	Yes	Yes	Yes	Yes	No	Identical

Table 2. Source Code Analysis Summary Table.

RCV Nullifies Computation. For 14 of the 18 errors, RCV nullifies the computation for the input unit that triggers the error, in effect turning that computation into a noop. The fifth column of Table 2 (RCV Nullifies Computation) indicates whether RCV nullifies the computation (Yes) or not (No). Unlike the corresponding patches, which skip the computation, RCV causes the computation to continue along the standard execution path. But for one or more of the following reasons, this continued execution has no effect:

- **Anticipated Error Check:** For seven of the 18 errors, the continued execution always encounters a check for an anticipated error case that the application is coded to handle correctly. The sixth column of Table 2 (RCV Hits Anticipated Error Check) indicates whether the continued execution always encounters such a check (Yes) or may not encounter such a check (No).
  Three of these seven checks are sanity checks designed to detect malformed inputs. Two of these three sanity checks (CVE-2012-4507, CVE-2012-3236) are influenced by RCV manufactured values, the other (CVE-2012-4285) is not.
  The remaining four checks are designed to catch a variety of internal errors such as failed memory allocations (CVE-2011-4153), assertions (CVE-2012-4286, but see Section 4.3), and null objects (CVE-2011-3083, CVE-2011-0421). Two (CVE-2011-3083, CVE-2011-0421) are influenced by RCV manufactured values, two (CVE-2012-4286, CVE-2011-4153) are not.
  As these results illustrate, one of the benefits of RCV is that it can enable applications to survive long enough to encounter an anticipated error check, execute recovery code, and continue its execution instead of crashing with an error.
- **Discarded Writes via Null Pointers:** CVE-2012-1593, CVE-2012-1128, and CVE-2011-1956 use a null pointer to update the shared data structures with the results of the computation. Because RCV discards writes via null pointers, it nullifies these updates and converts the computation into a noop even though the update code executes.
- **Discarded Calls via Null Pointers:** CVE-2011-3083 and CVE-2011-2849 invoke methods on null objects. RCV nullfies the invoked method by skipping the method invocation.
- **Zero Loop Count:** CVE-2011-1691 uses a null pointer to fetch the iteration count for the loop that performs all of the externally visible updates. The RCV manufactured value (zero) nullifies the computation by causing the loop to execute zero iterations.
- **Zero Input Field:** CVE-2013-2483 and CVE-2012-4285 use an input field that contains zero as the divisor in a length or size calculation. In both cases the computation is coded to be a noop if the input field is zero (in which case the computation never uses the manufactured RCV values). So the net effect of RCV is to enable the computation to survive the divide-by-zero and continue on to perform the computation, which is correctly coded to handle the zero input field.

Cleanup Required. For 13 of the 18 errors, our analysis indicates that the computation, even if skipped or nullified, must perform some cleanup of allocated resources to avoid a resource leak. When execution hits an anticipated error check (except CVE-2012-4286), the application's recovery code correctly performs this cleanup. Otherwise (except CVE-2012-4233), RCV enables the computation to execute through the errors to reach the cleanup code at the end of the computation. The seventh column of Table 2 indicates whether cleanup is required (Yes) or not (No).

Patched Versus RCV, All Inputs. For all of the errors except CVE-2012-4233 (for which source code is not available), we analyzed the source code to determine the differences (if any) between the patched and RCV versions of the computation. The eighth column of Table 2 (Patched vs. RCV, All Inputs) presents whether the patched and RCV versions produce identical results on all inputs (Identical, nine of 18 errors), equivalent results on all inputs, i.e, the same except that the patched and RCV versions may generate different error messages (Equivalent, two of 18 errors), acceptable results, i.e., the patched and RCV versions may produce different results, but both are acceptable (Acceptable, three of 18 errors) , or different results (Different, three of 18 errors).

4.3 Analysis of Individual Errors

We next discuss relevant aspects of the RCV recovery for each of the errors. For CVE-2011-3083, see Section 2.

CVE-2013-2483: A malformed ACN packet with a zero value in the `acn_count` field triggers a divide-by-zero error when Wireshark-1.8.5 uses the `acn_count` field to compute the length of a packet fragment (see `packet-acn.c:1045`). The error occurs when Wireshark processes the Device Management Protocol (DMP) information in the ACN packet. The patch checks if the `acn_count` field is zero, in which case it skips the computation for the DMP information. The RCV version continues through the error to execute the loop that processes the DMP information. Because the iteration count of this loop is the value in the `acn_count` field, the loop executes zero iterations, in effect nullifying the computation for the DMP information to produce identical results as the patched version.

CVE-2012-4286: A PCAP Next Generation Dump (PCAPNG) file with a malformed interface description block (IDB) causes Wireshark-1.8.1 to invoke the unknown block handler (`pcapng_read_unknown_block()`) instead of the IDB block handler (`pcapng_read_if_descr_block()`) for the malformed IDB block. The invoked handler writes unknown block information into a union data structure. If the application subsequently processes a packet that references the malformed IDB block, it interprets the union data structure as containing information for an IDB block. This misinterpretation/corruption eventually triggers a divide-by-zero error. The patch rewrites the handler dispatch code to check for a malformed interface description block. If it encounters a malformed block, it discards the malformed block and any remaining unprocessed interface description blocks. The patched version then proceeds on to process the packets in the file. If it encounters a packet that references the malformed block (or any of the unprocessed interface description blocks), it generates an error message and skips any any remaining packets in the input file.

The RCV version processes the malformed block along with any remaining interface description blocks. If the application encounters a packet that references the malformed block, RCV enables the application to execute through the resulting divide-by-zero error, but the application as released then encounters a fatal assertion violation. Disabling assertions enables the application to finish processing the packet and (empirically) continue on to successfully process any remaining packets in the input file. Note that the patched version would not process any of these remaining packets. The RCV version therefore provides more robust and resilient execution than the patched version.

We note that when the RCV version closes the input file, data corruption in the union data structure causes the application to attempt to free memory that it did not allocate. Configuring libc to ignore such attempts enables the application to, depending on user interaction, either exit gracefully or continue on to process another input file (as opposed to terminating with an error inside libc).

CVE-2012-4285: A malformed DCP-ETSI packet with a zero payload length triggers a divide-by-zero error when Wireshark-1.8.1 uses this length to compute a minimum number of required fragments (see `packet-dcp-etsi.c:4285`). The patch checks if the payload length is zero, in which case it generates an error message and skips the computation for the packet. The RCV version executes through the error to eventually encounter a subsequent anticipated packet format check. Because the packet does not satisfy the check, the application generates an error message and skips the computation for the packet. The check occurs before the computation performs any externally visible updates. The patched and RCV versions differ only in the error messages that they generate.

CVE-2012-1143: A malformed font file can trigger a divide-by-zero error in the utility division routine `FT_DivFix()` in the FreeType-2.4.8 library. The patch checks if the denominator parameter in this utility routine is zero, in which case it skips the division and returns `0x7fffffff` as the result of the division. The RCV version continues through the error and returns zero as the result of the division. These values are eventually stored in the data structures that represent the font and are flushed from the system when these data structures are deallocated. It is not clear to us how or even if the difference in these two values may potentially affect the behavior of the client of the font.

CVE-2012-5668: A malformed Bitmap Distribution Format (BDF) font with a large `props_size` field (which controls the number of properties in the font) triggers a null-dereference error in the cleanup routine of the FreeType-2.4.8 library. The program allocates an array of `props_size` pointers (see `bdflib.c:2164`) and then fills in the allocated pointer array with references to additional allocated memory. If the allocation of the pointer array fails, a null-dereference error occurs when a loop in the FreeType library iterates over the null pointer array attempting to deallocate the additional allocated memory blocks (see `bdflib.c:2475`).

The patch sets the `props_size` variable to zero if the allocation of the pointer array fails, which nullifies the deallocation loop (it performs zero iterations). The RCV version executes the deallocation loop six times (it then jumps out of the loop, see Section 3.2), but nullifies each iteration by passing the manufactured value of zero into the invoked `free()` operation (which is a noop when passed a zero pointer to `free()`). Therefore the patched and RCV versions produce identical results on all inputs.

CVE-2012-4507: An email triggers a null-dereference error when Claws Mail-3.8.1 processes an unsupported MIME header parameter (either a content type, content disposition, or content transfer parameter). The hashset lookup of the parameter returns a null pointer (for supported parameters, this lookup returns a string that encodes the value of the parameter). As part of the computation that attempts to extract the value of the parameter from the string, the application passes this (null) pointer into `strchr()`, which dereferences the pointer when it attempts to access the string.

The patch checks if the hashset lookup attempt failed, in which case it skips the unsupported parameter. The RCV version continues through the error, returns from `strchr()`, and checks if `strchr()` found the desired character. The check fails and the RCV version skips the unsupported parameter. The patched and RCV versions produce identical results on all inputs.

CVE-2012-4233: A malformed spreadsheet triggers a null-dereference error when LibreOffice-3.5.5.2 opens the spreadsheet. The error occurs within the shared library `libscfiltlo.so`. The source code of this library is not available.

For the error-triggering input, the RCV version generates a warning message that it cannot display part of the spreadsheet. The RCV version then proceeds on to display the properly formed part of the spreadsheet, with results identical to the patched version of LibreOffice on the same input. All values influenced by the RCV manufactured values are flushed from the state at the end of the spreadsheet parsing phase.

CVE-2012-3236: A FITS image with a malformed or missing XTENSION field triggers a null-dereference error when the GIMP FITS plugin (in GIMP-2.8.0) parses the image (see `fits-io.c:1059`). The error occurs when the application attempts to dereference the (null) pointer to the object that holds data from the parsed XTENSION field. The patch checks if the function that parses the XTENSION field returns null (indicating a malformed or missing XTENSION field), in which case it aborts the parse, sends an error message to the main process, and gracefully exits the plugin (which runs in a separate process from the main GIMP process). The RCV version continues through the error. Before it performs any externally visible updates, it encounters an anticipated error check for a malformed GCOUNT/PCOUNT subfield in the XTENSION field. The check indicates that the subfield is malformed. The RCV version performs the same error handling actions as the patched version, aborts the parse, sends a slightly different error message to the main process, and gracefully exits the plugin.

If the RCV error containment system is turned on, it blocks the transmission of the error message from the plugin back to the parent. With the error containment system turned off, the RCV version displays a slightly different error message than the patched version. In either case, the patched and RCV versions differ only in the error messages they generate.

CVE-2012-1593: A malformed ANSI Mobile Application Pact (MAP) packet with an abnormal fragment order may trigger a null-dereference error in Wireshark-1.6.5, which depends on input packets occurring in an expected order to properly initialize a shared pointer variable `g_pinfo` in `packet-ansi_a.c`. `g_pinfo` refer-

ences a data structure that holds information that is displayed in the rightmost column of the Wireshark GUI. If the fragments occur out of the expected order, `g_pinfo` is null when the application processes the first out of order fragment.

The patch removes the `g_pinfo` variable and rewrites the function interfaces in `packet-ansi_a.c` to pass the data structure as a parameter to the fragment processing functions. The RCV version continues through the error to discard all updates to the shared data structure via the null `g_pinfo` pointer when it processes out of order fragments. The only difference between the patched and RCV versions is the contents of the shared information data structure (in the RCV version, this data structure is missing the updates from the out of order fragments). The only visible difference is the contents of the rightmost column in the Wireshark GUI.

CVE-2012-1128: A TrueType font causes the FreeType-2.4.8 library to incorrectly calculate the iteration count of the loop at `ttinterp.c:5867`, which eventually triggers a null-dereference error inside the loop. The patch corrects the iteration count calculation so that the loop will not execute any iterations for inputs that trigger the error. The RCV version executes the loop iterations, but nullifies them by discarding writes via the null pointer (the loop performs no other externally visible updates). The patched and RCV versions produce identical results on all inputs.

CVE-2012-0781: A PHP program calls the diagnose method with a null Tidy object. The PHP interpreter (PHP-5.3.8) maintains a data structure associated with this null object. This data structure contains a null reference to the corresponding Tidy document, which the PHP interpreter attempts to dereference. The patch checks if the Tidy object is null, in which case it skips the diagnose method.

The RCV version continues through the error to append a diagnose message to an error buffer in the data structure associated with the null Tidy object. The application can observe the difference between the two versions only if it uses Tidy object APIs to query the modified error buffer of the null Tidy object.

CVE-2011-4153: A PHP program that invokes `str_repeat()` may trigger a null-dereference error in the PHP interpreter (PHP-5.3.8) when a memory allocation in `strndup()` fails. In this case, `strndup()` returns null (see `zend_builtin_function.c:685`); the error occurs when the PHP interpreter attempts to deference the returned null reference. The patched version checks if the returned reference is null, in which case it skips the computation for the current PHP statement.[1] The RCV version continues through the error to encounter another failed memory allocation. This is an anticipated error case that the application is coded to handle correctly, specifically by printing an error message and gracefully exiting the PHP program execution.

CVE-2011-3182: A PHP program that invokes `strtotime()` without sufficient available memory triggers a null-dereference error in the PHP interpreter (PHP-5.3.6) when the memory allocation operation `malloc()` at `parse_date.c:24674` fails and the application attempts to dereference the returned null reference. The patched version checks if the returned value is null, in which case it skips the computation for the current PHP statement and returns the PHP boolean value false as the result of `strtotime()`, which indicates that an error occurred in `strtotime()`.[1] The RCV version continues through the error and returns the PHP value zero from `strtotime()` (indicating a time of Jan. 1, 1970, 00:00:00 UTC).

CVE-2011-2849: A malicious web page triggers a null-dereference error in Chromium-12.0.742.112 when it causes Chromium to use a Chromium WebSocketJob object to generate repetitive communication between the browser and the web server that serves the page. The code in the WebSocketJob object that handles the communication contains a data race. This data race can cause the object to enter an inconsistent state in which the underlying `SocketStream` object has been released but the state flag in the WebSocketJob object indicates that the connection is still active. This inconsistency triggers a null-dereference error at `websocket_job.cc:495` when the application, in an attempt to send data, invokes the `SendData()` method on the released `SocketStream` object.

The patch rewrites the code to eliminate the data race. The RCV version continues through the error by skipping the invoked `SendData()` method. This repair turns the communication request into a noop. Chromium then retries the communication until it succeeds (the inconsistent state is transient, when the inconsistency resolves the communication will succeed).

CVE-2011-1956: When Wireshark-1.4.5 processes a TCP packet that contains data, it passes a null `start_ptr` parameter to the `proto_tree_add_bytes_format()` function (see `packet-tcp.c:1888`). The null-dereference error occurs when the function attempts to copy the raw TCP payload data from `start_ptr` into another data structure (see `proto.c:1740`).

The patch passes the correct `start_ptr` parameter (this correct parameter points to the raw payload data) to the `proto_tree_add_bytes_format()` function. The RCV version continues through the null pointer dereferences, returning zero as the result of the reads from the null `start_ptr` parameter (as if the payload were all zeros). Because Wireshark never uses the copied raw TCP payload data, the patched and RCV versions produce identical results on all inputs.

CVE-2011-1691: A malicious web page that operates on an empty Computed Style Sheet (CSS) declaration object triggers a null-dereference error in Chromium-12.0.717.0. The web page includes a script that looks up the counter increment attribute of the empty CCS declaration object. Such attributes are stored in the counter derivative map, but because the CSS declaration object is empty, the map is null. The null-dereference occurs when the application attempts to iterate over the elements in the null map.

The patch checks if the counter derivative map of the CSS declaration object is null, in which case it returns zero indicating that the attribute was not found. The RCV version nullifies the attribute lookup loop (the loop executes zero iterations, see `CSSComputedStyleDeclaration.cpp:769`) and eventually returns zero again indicating that the attribute was not found. The patched and RCV versions produce identical results on all inputs.

CVE-2011-0421: A PHP program that invokes `nameLocate()` to look up a file in a null PHP ZIP archive object triggers a null-dereference error in the PHP interpreter (PHP-5.3.5) at `zip_name_locate.c:64`. The PHP interpreter maintains a data structure that represents the null PHP ZIP archive object. This data structure contains a reference to a data structure that the PHP interpreter uses to represent the PHP ZIP archive object. This reference is null if the PHP ZIP archive object is null. The error occurs when the `nameLocate()` function in the PHP interpreter attempts to use this null reference to access the directory information for the null PHP ZIP archive object.

The patch checks whether the PHP ZIP archive object is null, in which case it skips the lookup and returns an error code indicating that the file was not found. The RCV version continues through error to execute the loop that looks up the file. Because RCV returns zero as the result of all reads via null references (see zip_name_locate.c:65-83), the PHP interpreter does not find the file. The RCV version then returns the same file not found error code as the patched version.

[1] Because the developer of the PHP interpreter declined to integrate a patch that corrects this error, the current version of PHP (PHP-5.3.28) still contains this error. We therefore use the patch suggested in the CVE error report.

5. Related Work

Runtime Program Recovery: Failure-Oblivious Computing (FOC) [25] discards out of bounds writes, manufactures values for out of bounds reads, and (like RCV) enables applications to continue along their normal execution path. FOC leverages existing bounds checks in safe languages (e.g., Java) or adds bounds checks for unsafe languages (e.g., C). Instead of relying on software bounds checks to detect errors, RCV relies on signals, which are generated by hardware exceptions. RCV therefore entails little to no error detection overhead even for unsafe languages. FOC was evaluated on five errors in five server applications.

SRS [21] suppresses memory corruption errors in servers that process a sequence of requests — when it detects such an error, it enters a crash suppression mode in which instructions that access corrupted values do not execute. It exits this mode when it returns back to process the next request. SRS relies on user annotations to identify the start of request processing code and on profiling runs to identify shared memory locations that may transmit corrupted data between requests. Even during normal execution, SRS adds binary instrumentation to every store instruction to maintain a record of the user request id of the request that generated the write. SRS was evaluated on four errors from four server applications.

RCV, in contrast, requires no user annotations and no profiling runs, does not rely on any particular application structure, and imposes no binary instrumentation during normal execution. RCV also detects when the recovery effects have been flushed from the process state, no matter how or when these effects are flushed. We also evaluate RCV on a larger, more comprehensive set of errors and therefore obtain more insight into how well the RCV techniques will generalize to other applications and other classes of errors. We also evaluate RCV on client applications and find that RCV provides two benefits in this context: 1) the continued RCV execution enables the developer to save any pending changes even in the presence of otherwise fatal errors, and 2) in comparison with approaches that terminate and then restart the client application, RCV provides efficient continued execution with no interruption.

Because they tend to nullify the associated computations, RCV, FOC, and SRS all work well with programs (such as servers) that process a sequence of largely or completely independent input units or requests. Previous research also shows that discarding tasks with errors can enable applications to survive errors to produce acceptably accurate results even when the application obtains the final result by combining the results that the tasks produce [26, 27].

APPEND [10] is a compile-time tool that inserts null-dereference checks into Java programs. If a check detects a null-dereference, it executes error-handling code that may, for example, allocate a default object or skip the computation and continue. APPEND was evaluated on three null dereference errors in three Java programs. RCV, in contrast, operates directly on x86 binaries, does not insert any null checks into the program, was evaluated on a more comprehensive set of errors, applies a recovery strategy that does not allocate memory but, in effect, interprets null references as references to an object containing all zeros, and detects when recovery effects have been flushed from the system.

Jolt [7] and Bolt [16] enable applications to survive infinite loop errors. When an infinite loop error occurs, they jump out of the loop or the enclosing function to escape the error. RCV targets a different set of errors, but could easily be combined with Jolt or Bolt to obtain a unified system for infinite loop, divide-by-zero, and null-dereference errors.

ClearView [23] learns a set of runtime invariants from training runs. It then collaboratively patches errors by enforcing the learned invariants at runtime. RCV has no learning phase and (unlike ClearView) does not apply expensive binary instrumentation during normal execution.

DieHard [5] provides probabilistic memory safety in the presence of memory errors. In stand-alone mode, DieHard replaces the default memory manager with a memory manager that places objects randomly across a heap to reduce the possibility of memory overwrites due to buffer overflows. In replicated mode, DieHard obtains the final output of the application based on the votes of multiple replications. Unlike RCV, DieHard does not attempt to enable the program to survive null-dereference errors. It instead treats null-dereference errors as uninitialized reads and will abort the application execution when uninitialized reads occur.

Rx [24] takes periodic checkpoints. When an error occurs, Rx reverts back to a previous checkpoint and makes semantically equivalent system-level changes (e.g, memory allocations, process scheduling, etc.) to search for executions that do not trigger the error. ARMOR [8] is a similar checkpoint-based recovery system for Java applications. ARMOR relies on user-provided specifications to find semantically equivalent workarounds. It then re-executes the application with these workarounds when an error occurs.

Error Virtualization [28–31] is a general error recovery technique that retrofits exception-handing capabilities to legacy software. Failures that would otherwise cause a program to crash are turned into transactions that use a program's existing error handling routines to survive from unanticipated faults. ASSURE [30] generalizes Error Virtualization to enable the system to transactionally terminate any one of the functions on the call stack at the time of the error (and not just the function containing the error). Attack replay on a triage machine enables the system to evaluate which function to terminate to provide the most successful recovery. The applied patch takes a checkpoint at the start of the function. It responds to errors by restoring the checkpoint, then returning an effective error code to terminate the function and continue execution at the caller. In contrast, RCV does not depend on checkpointing and is therefore applicable to situations where the cost of taking checkpoints is unaffordable (e.g, client applications).

Static Program Repair: Researchers have also developed techniques to automatically synthesize a program patch for an error [14, 15, 32]. GenPro [32] is a automatic program repair tool that uses an evolutionary algorithm to synthesize program patches from existing source code with a set of mutation rules. PAR [15] applies patch templates to automatically generate patches for new errors. Khmelevsky et al. [14] present a source-to-source error repair technique for missing condition checks after a method call. Unlike these tools, the goal of RCV is to repair the execution of the application, not the source code — continued execution with RCV can reduce or eliminate data losses in running applications.

Self-Stabilizing Java: SJava [12] is a Java type system that enables the compiler to prove that the effects of any error will be flushed from the system state after a fixed number of iterations.

Input Rectification and Filtering: Input rectification [18] empirically learns a set of input constraints from benign training inputs, then enforces learned constraints on incoming inputs to nullify potential errors. Starting with critical program sites such as memory allocation or block copy sites, sound input filter generation [19] statically analyzes the program to generate filters that ensure that any input passed to the program will not trigger an error at that site.

Data Structure Repair: Data structure repair enables applications to recover from data structure corruption errors [9].

The Ariane-5 Disaster: The Ariane-5 disaster is one of the most prominent and widely studied software disasters in history [17]. Shortly after launch, the Ariane-5 control software encountered an arithmetic overflow error. In response, the exception handler terminated the control software and switched to the backup. The backup encountered the same arithmetic overflow error. The invoked exception handler again terminated the execution, leaving the rocket flying with no control and forcing the launch team to destroy the

rocket. The value whose computation caused the arithmetic overflow error was never used. Applying the basic RCV recovery strategy (return a manufactured value as the result of the operation that generated the exception, then continue execution along the normal execution path) would have enabled the Ariane-5 to successfully perform its mission.

6. Conclusion

Divide-by-zero and null-dereference errors cause undesirable application crashes. RCV implements a recovery shepherding technique that attaches to the application process when an error occurs, repairs the execution, contains the error within the process during recovery, and detaches from the process when the effect of the recovery has been flushed from the system. For the majority of the benchmark errors in this paper, the RCV versions produce identical or equivalent results as the subsequent versions with developer-supplied patches that correct the errors. These results, which are consistent with previous results for philosophically similar repair strategies for other kinds of errors [7, 16, 21, 22, 25, 30, 31], contribute to a mounting body of evidence that simple repairs can effectively unlock the inherent resilience already present in many applications and enable these applications to successfully survive otherwise fatal errors.

Acknowledgements

We thank Christopher Musco, Deokhwan Kim, Sasa Misailovic, and the anonymous reviewers for their insightful comments. This research was supported by DARPA (Grant FA8650-11-C-7192).

References

[1] Chromium's multi-process architecture. `http://blog.chromium.org/2008/09/multi-process-architecture.html`.

[2] Common vulnerabilities and exposures. `http://cve.mitre.org/`.

[3] The libunwind project. `http://www.nongnu.org/libunwind/`.

[4] SPEC CPU2006. `http://www.spec.org/cpu2006/`.

[5] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06', pages 158–168. ACM, 2006.

[6] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, Nov. 2000.

[7] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 609–633. Springer-Verlag, 2011.

[8] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 782–791.

[9] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12):931–951, 2006.

[10] K. Dobolyi and W. Weimer. Changing java's semantics for handling null pointer exceptions. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 0:47–56, 2008.

[11] P. Dubroy and R. Balakrishnan. A study of tabbed browsing among mozilla firefox users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 673–682. ACM, 2010.

[12] Y. h. Eom and B. Demsky. Self-stabilizing java. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12', pages 287–298. ACM, 2012.

[13] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12', pages 121–132. ACM, 2012.

[14] Y. Khmelevsky, M. Rinard, and S. Sidiroglou. A source-to-source transformation tool for error fixing. CASCON, 2013.

[15] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13', pages 802–811. IEEE Press, 2013.

[16] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12', pages 431–450. ACM, 2012.

[17] J. Lions. Ariane 5 flight 501 failure: Report by the inquiry board., 1996.

[18] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 80–90. IEEE Press, 2012.

[19] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. C. Rinard. Sound input filter generation for integer overflow errors. In *POPL*, pages 439–452, 2014.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05', pages 190–200. ACM, 2005.

[21] V. Nagarajan, D. Jeffrey, and R. Gupta. Self-recovery in server programs. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09', pages 49–58. ACM, 2009.

[22] H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07', pages 15–30. ACM, 2007.

[23] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102. ACM, 2009.

[24] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies–a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 25(3), Aug. 2007.

[25] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[26] M. C. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*, pages 324–334, 2006.

[27] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA*, pages 369–386, 2007.

[28] S. Sidiroglou, Y. Giovanidis, and A. Keromytis. A Dynamic Mechanism for Recovery from Buffer Overflow attacks. In *Proceedings of the 8th Information Security Conference (ISC)*, September 2005.

[29] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE Workshop on Enterprise Technologies*, June 2003.

[30] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: Automatic software self-healing using rescue points. In *ASPLOS*, pages 37–48, 2009.

[31] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the general track, 2005 USENIX annual technical conference: April 10-15, 2005, Anaheim, CA, USA*, pages 149–161. USENIX, 2005.

[32] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09', pages 364–374. IEEE Computer Society, 2009.