Debugging in the (Very) Large: Ten Years of Implementation and Experience

Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul,
Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

ABSTRACT

Windows Error Reporting (WER) is a distributed system that automates the processing of error reports coming from an installed base of a billion machines. WER has collected billions of error reports in ten years of operation. It collects error data automatically and classifies errors into buckets, which are used to prioritize developer effort and report fixes to users. WER uses a progressive approach to data collection, which minimizes overhead for most reports yet allows developers to collect detailed information when needed. WER takes advantage of its scale to use error statistics as a tool in debugging; this allows developers to isolate bugs that could not be found at smaller scale. WER has been designed for large scale: one pair of database servers can record all the errors that occur on all Windows computers worldwide.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *debugging aids*.
D.2.9 [**Software Engineering**]: Management – *life cycle, software quality assurance*.
D.4.5 [**Operating Systems**]: Reliability.

General Terms

Management, Measurement, Reliability.

Keywords

Bucketing, Classifying, Error Reports, Labeling, Minidump, Statistics-based Debugging.

1. INTRODUCTION

Debugging a single program run by a single user on a single computer is a well understood problem. It may be arduous, but follows general principles: when a user reproduces and reports an error, the programmer attaches a debugger to the running process or a core dump and examines program state to deduce where algorithms or state deviated from desired behavior. When tracking particularly onerous bugs the programmer can resort to restarting and stepping through execution with the user's data or providing the user with a version of the program instrumented to provide additional diagnostic information. Once the bug has been isolated, the programmer fixes the code and provides an updated program.[1]

Debugging in the large is harder. When the number of software components in a single system grows to the hundreds and the number of deployed systems grows to the millions, strategies that worked in the small, like asking programmers to triage individual error reports, fail. With hundreds of components, it becomes much harder to isolate the root cause of an error. With millions of systems, the sheer volume of error reports for even obscure bugs can become overwhelming. Worse still, prioritizing error reports from millions of users becomes arbitrary and ad hoc.

As the number of deployed Microsoft Windows and Microsoft Office systems scaled to tens of millions in the late 1990s, our programming teams struggled to scale with the volume and complexity of errors. The Windows team devised a tool that could automatically diagnose a core dump from a system crash to determine the most likely cause of the crash. We planned to deploy this tool as a web site where a system administrator could upload a core dump and receive a report listing probable resolutions for the crash. Separately, the Office team devised a tool that on an unhandled exception (that is, a crash) would automatically collect a stack trace with a small of subset of heap memory and upload this *minidump* to a service at Microsoft that would collect these error reports by faulting module.

We realized we could tie the automatic diagnosis tool from the Windows team with the automatic collection tool from the Office team to create a new service, Windows Error

SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.

[1] We use the following definitions: ***error*** (noun): A single event in which program behavior differs from that intended by the programmer. ***bug*** (noun): A root cause, in program code, that results in one or more errors.

Reporting (WER). WER could automatically generate error reports for applications and operating systems, report them to Microsoft, and automatically diagnose them to help users and programmers.

WER is a distributed, post-mortem debugging system. When an error occurs, client code on the Windows system automatically collects information to create an error report. With authorization from the user or administrator, the client code reports the error to the WER service. If a fix for the error already exists, the WER service provides the client with a URL to the fix. The WER service aggregates and diagnoses error reports. Programmers can access post-mortem data from one or more error reports to debug code. Programmers can also request the collection of additional data in future error reports to aid debugging.

Beyond basic debugging from error reports, WER enables statistics-based debugging. WER gathers all error reports to a central database. In the large, programmers can mine the error report database to prioritize work, spot trends and test hypotheses. An early mantra of our team was, "data not decibels." Programmers use data from WER to prioritize debugging so that they fix the bugs that affect the most users, not just the bugs hit by the loudest customers. WER data also aids in correlating failures to co-located components. For example, WER data can show when a collection of seemingly unrelated crashes all contain the same likely culprit—say a device driver—even though it isn't on any thread stack in any of the error reports.

Three principles account for the widespread adoption of WER by every Microsoft product team and by over 700 third party companies: automated error diagnosis and progressive data collection, which enable error processing at scale, and statistics-based debugging, which harnesses that scale to help programmers more effectively improve system quality.

WER has repeatedly proven its value to Microsoft teams by identifying bugs "in the wild". For example, the Windows Vista programmers found and fixed over 5,000 bugs isolated by WER in beta releases of Vista. These bugs were found *after* programmers had found and fixed over 100,000 bugs [10] with static analysis and model checking tools [6], but *before* the general release of Vista. Every Microsoft application, server, service, and OS team makes a significant reduction in WER reports a part of their ship criteria for product releases.

WER is not the first system to automate the collection of memory dumps. Post-mortem debugging has existed since the dawn of digital computing. In 1951, The Whirlwind I system [12] dumped the contents of tube memory to a CRT in octal when a program crashed. An automated camera took a snapshot of the CRT on microfilm, delivered for debugging the following morning [8]. Later systems dumped core to disk; used partial core dumps, which excluded shared code, to minimize the dump size [32]; and eventually used telecommunication networks to deliver core dumps to the computer manufacturer [19].

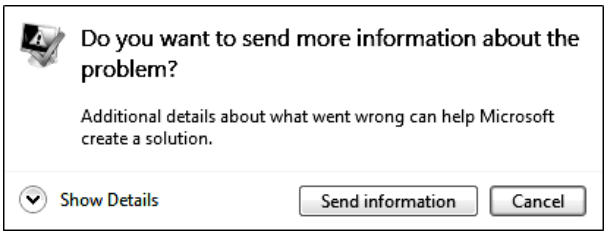

Figure 1. Typical WER Authorization Dialog.

Though WER is not the first system of its kind, it is the first system to use automatic error diagnosis, the first to use progressive data collection to reduce overheads, and the first to automatically direct users to available fixes based on automated error diagnosis. In use since 1999, WER remains unique in four aspects:

1. WER is the largest automated error-reporting system in existence. Approximately one billion computers run WER client code: every Windows system since Windows XP, Windows Mobile, all Microsoft programs on Macintosh, and Windows Live JavaScript code in any browser.

2. WER automates the collection of additional client-side data when needed for further debugging. When initial error reports provide insufficient data to debug a problem, programmers can request that WER collect more data on future error reports including: broader memory dumps, environment data, registry values, log files, program settings, and/or output from management instrumentation.

3. WER automatically directs users to solutions for corrected errors. This automatic direction is helpful as users often unknowingly run out-of-date programs. Currently, 47% of kernel crash reports result in a direction to a software update.

4. WER is general purpose. It is used for OSes and applications by Microsoft and non-Microsoft programmers. WER collects error reports for crashes, non-fatal assertion failures, hangs, setup failures, abnormal executions, and device failures.

Section 2 defines the key challenges and our strategy to diagnose error reports from a billion clients. Sections 3 and 4 describe our algorithms for processing error reports and the use of WER for statistics-based debugging. Section 5 describes the WER implementation. Section 6 evaluates WER's effectiveness over ten years of use. Section 7 describes changes made to Windows to improve debugging with WER. Section 8 discusses related work and Section 9 concludes.

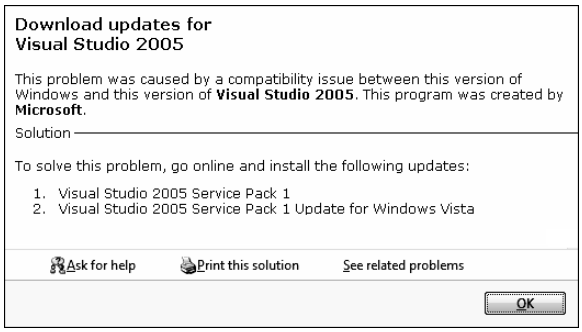

Figure 2. Typical Solution Dialog.

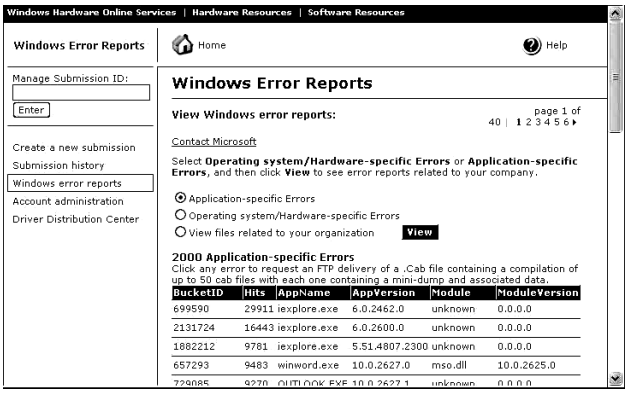

Figure 3. The WER Website (http://winqual.microsoft.com).

2. PROBLEM, SCALE, AND STRATEGY

The goal of WER is to allow us to diagnose and correct every software error on every Windows system. To achieve this goal, WER must produce data that focuses programmers on the root cause of errors. It must help programmers of every class of software in the system: Microsoft and third-party; OS, system, device driver, application, and plug-in. It must help programmers prioritize and fix all errors, even the most non-deterministic. An inability to scale must never prevent clients from reporting errors; it must admit literally every Windows system. Finally, it must address user and administrative policy concerns, such as privacy and bandwidth usage, so that it may be used anywhere.

We realized early on that scale presented both the primary obstacle and the primary resolution to address the goals of WER. If we could remove humans from the critical path and scale the error reporting mechanism to admit millions of error reports, then we could use the law of large numbers to our advantage. For example, we didn't need to collect all error reports, just a statistically significant sample. And we didn't need to collect complete diagnostic samples for all occurrences of an error with the same root cause, just enough samples to diagnose the problem and suggest correlation. Moreover, once we had enough data to allow us to fix the most frequently occurring errors, then their occurrence would decrease, bringing the remaining errors to the forefront. Finally, even if we made some mistakes, such as incorrectly diagnosing two errors as having the same root cause, once we fixed the first then the occurrences of the second would reappear and dominate future samples.

Realizing the value of scale, six strategies emerged as necessary components to achieving sufficient scale to produce an effective system: bucketing of error reports, collecting data progressively, minimizing human interaction, preserving user privacy, directing users to solutions, and generalizing the system.

2.1. Bucketing

WER aggregates error reports likely originating from the same bug into a collection called a *bucket*[2]. Otherwise, if naively collected with no filtering or organization WER data would absolutely overwhelm programmers. The ideal bucketing algorithm should strictly maintain a property of orthogonality: one bug per bucket and one bucket per bug. WER approaches orthogonality through two phases of bucketing. First, errors are *labeled*, assigned to a first bucket based on immediate evidence available at the client with the goal that each bucket contains error reports from just one bug. Second, errors are *classified* at the WER service; they are assigned to new buckets as additional data is analyzed with the goal of minimizing programmer effort by placing error reports from just one bug into just one final bucket.

Bucketing enables automatic diagnosis and progressive data collection. Good bucketing relieves programmers and the system of the burden of processing redundant error reports, helps prioritize programmer effort by bucket prevalence, and can be used to link users to fixes when the underlying cause for a bucket has been corrected. In WER, bucketing is progressive. As additional data related to an error report is collected, such as symbolic information to translate from an offset in a module to a named function, the report is associated with a new bucket. Although the design of optimal bucketing algorithms remains an open problem, Section 6.3 shows that the bucketing algorithms currently used by WER are in practice quite effective.

2.2. Progressive Data Collection

WER uses a progressive data collection strategy to reduce the cost of error reporting so that the system can scale to high volume while providing sufficient detail for

[2] ***bucket*** (noun): A collection of error reports likely caused by the same bug. ***bucket*** (verb): To triage error reports into buckets.

debugging. Most error reports consist of no more than a simple bucket identifier. If additional data is needed, WER will next collect a *minidump* (an abbreviated stack and memory dump, and the configuration of the faulting system) into a compressed Windows cabinet archive file (the *CAB file*). If further data beyond the minidump is required to diagnose the error, WER can progress to collecting full memory dumps then memory dumps from related programs, related files, or additional data queried from the reporting system into the CAB file. Progressive data collection reduces the scale of incoming data enough that one pair of SQL servers can record every error on every Windows system world-wide. Progressive data collection also reduces the user cost in time and bandwidth of reporting errors, thus encouraging user participation.

2.3. Minimizing Human Interaction

WER removes users from all but the authorization step of error reporting and removes programmers from initial error examination by automated error diagnosis. User interaction is reduced in most cases to a yes/no authorization (see Figure 1). To further reduce interaction, users can permanently opt in or out of future authorization requests. WER servers analyze each error report automatically to direct users to existing fixes and, as needed, ask the client to collect additional data. Programmers are notified only after WER determines that an error report does not match any previously resolved bugs. Automated diagnosis allows programmers to focus on finding and fixing new bugs rather than rehashing stale problems.

2.4. Preserving User Privacy

We take considerable care to avoid knowingly collecting personal identifying information (PII). This reduces regulatory burden and encourages user participation. For example, although WER collects hardware configuration information, our client code zeros serial numbers and other known unique identifiers to avoid transmitting data that might identify the sending computer. WER operates on an *informed consent* policy with users. Errors are reported only with user consent. All consent requests default to negative, thus requiring that the user opt-in before transmission. WER reporting can be disabled on a per-error, per-program, or per-computer basis by individual users or by administrators. Because WER does not have sufficient metadata to locate and filter possible PII from collected stack or heap data, we minimize the collection of heap data and apply company data-access policies that restrict the use of WER data strictly to debugging and improving program quality.

To allow administrators to apply organizational policies, WER includes a feature called *Corporate Error Reporting* (CER). With CER, administrators can configure WER clients to report errors to a private server, to disable error reporting, to enable error reporting with user opt-in, or to force error reporting with no end-user opt-out. Where errors are reported locally, administrators can use CER to pass some or all error reports on to the WER service.

2.5. Providing Solutions to Users

Many errors have known corrections. For example, users running out-of-date software should install the latest service pack. The WER service maintains a mapping from buckets to *solutions*. A solution is the URL of a web page describing steps a user should take to prevent reoccurrence of the error (see Figure 2). Solution URLs can link the user to a page hosting a patch for a specific problem, to an update site where users with out-of-date code can get the latest version, or to documentation describing workarounds (see Figure 3). Individual solutions can be applied to one or more buckets with a simple regular expression matching mechanism. For example, all users who hit any problem with the original release of Word 2003 are directed to a web page hosting the latest Office 2003 service pack.

2.6. Generalizing the System

While our original plan was to support error reporting for just the Windows kernel and Office applications, we realized WER had universal value when other teams started asking how to provide a similar service. We considered letting each Microsoft product team run its own WER service, but decided it was easier to open the service to all than to package the server software. Though developed by Office and Windows, the first program to ship with WER client code was MSN Explorer. We run WER as a global service and provide access to WER data to programmers inside and outside Microsoft. We operate a free WER website (see Figure 3) for third-party software and hardware providers to improve their code.

3. BUCKETING ALGORITHMS

One of the most important elements of WER is its mechanism for assigning error reports to buckets. This is carried out using a collection of heuristics. Conceptually WER bucketing heuristics can be divided along two axes. The first axis describes where the bucketing code runs: *labeling* heuristics are performed on clients (to minimize server load) and *classifying* heuristics are performed on servers (to minimize programmer effort). The second axis describes the impact of the heuristic on the number of final buckets presented to programmers from a set of incoming error reports: *expanding* heuristics increase the number of buckets with the goal that no two bugs are assigned to the same bucket; *condensing* heuristics decrease the number of buckets with the goal that no two buckets contain error reports from the same bug.

Expanding and condensing heuristics should be complimentary, not conflicting. An expanding heuristic

	Heuristic	Impact	Description
L1	program_name	Expanding	Include program name in bucket label.
L2	program_version	Expanding	Include program version.
L3	program_timestamp	Expanding	Include program binary timestamp.
L4	module_name	Expanding	Include faulting module name in label.
L5	module_version	Expanding	Include faulting module version.
L6	module_timestamp	Expanding	Include faulting module binary timestamp.
L7	module_offset	Expanding	Include offset of crashing instruction in fault module.
L8	exception_code	Expanding	Cause of unhandled exception.
L9	bugcheck_code	Expanding	Cause of system crash (kernel only)
L10	hang_wait_chain	Expanding	On hang, walk chain of threads waiting on synchronization objects to find root.
L11	setup_product_name	Expanding	For "setup.exe", include product name from version information resource.
L12	pc_on_stack	Condensing	Code was running on stack, remove module offset.
L13	unloaded_module	Condensing	Call or return into memory where a previously loaded module has unloaded.
L14	custom_parameters	Expanding	Additional parameters generated by application-specific client code.
L15	assert_tags*	Condensing	Replace module information with unique in-code assert ID.
L16	timer_asserts*	Expanding	Create a non-crashing error report by in-code ID if user scenario takes too long.
L17	shipping_assert*	Expanding	Create a non-crashing error report if non-fatal invariant is violated
L18	installer_failure	Expanding	Include target application, version, and reason for Windows installer failure.

Figure 4. Top Heuristics for Labeling (run on client).
*Asserts require custom code and/or metadata in each program.

should not introduce new buckets for the same bug. A condensing heuristic should not put two bugs into one bucket. Working properly in concert, expanding and condensing heuristics should move WER toward the desired goal of one bucket for one bug.

3.1. Client-Side Bucketing (Labeling)

The first bucketing heuristics run on the client when an error report is generated (see Figure 4). The primary goal of these labeling heuristics is to produce a unique bucket label based purely on local information that is likely to align with other reports caused by the same bug. This labeling step is necessary because in most cases, the only data communicated to the WER servers will be a bucket label.

The primary labeling heuristics (L1 through L7) generate a bucket label from the faulting program, module, and offset of the program counter within the module. Additional heuristics are generated under special conditions, such as when an error is caused by an unhandled program exception (L8). Programs can use custom heuristics for bucketing by calling the WER APIs to generate an error report. For example, Office's build process assigns a unique permanent tag to each assert statement in the source code. When an assert failure is reported as an error, the bucket is labeled by the assert tag (L15) rather than the module information. Thus all instances of a single assert are assigned to a single bucket even as the assert moves in the source over time.

Most of the labeling heuristics are expanding heuristics, intended to spread separate bugs into distinct buckets. For example, the `hang_wait_chain` (L10) heuristic uses the `GetThreadWaitChain` API to walk the chain of threads waiting on synchronization objects held by threads, starting from the program's user-input thread. If a root thread is found, the error is reported as a hang originating with the root thread. Windows XP lacked `GetThreadWaitChain`, so all hangs for a single version of a single program are bucketed together. The few condensing heuristics (L12, L13, and L14) were derived empirically for common cases where a single bug produced many buckets. For example, the `unloaded_module` (L13) heuristic condenses all errors where a module has been unloaded prematurely due to a reference counting bug.

3.2. Server-Side Bucketing (Classifying)

The heuristics for server-side bucketing attempt to classify error reports to maximize programmer effectiveness. They are codified in `!analyze` ("bang analyze"), an extension to the Windows Debugger [22]. The heuristics for bucketing in WER were derived empirically and continue to evolve. `!analyze` is roughly 100,000 lines of code implementing some 500 bucketing heuristics (see Figure 5), with roughly one heuristic added per week.

The most important classifying heuristics (C1 through C5) are part of an algorithm that analyzes the memory dump to determine which thread context and stack frame most likely caused the error. The algorithm works by first finding any

	Heuristic	Impact	Description
C1	find_correct_stack	Expanding	Walk data structures for known routines to find trap frames, etc. to stack.
C2	skip_core_modules	Expanding	De-prioritize kernel code or core OS user-mode code.
C3	skip_core_drivers	Expanding	De-prioritize first-party drivers for other causes.
C4	skip_library_funcs	Expanding	Skip stack frames containing common functions like memcpy, printf, etc.
C5	third_party	Condensing	Identify third-party code on stack.
C6	function_name	Condensing	Replace module offset with function name.
C7	function_offset	Expanding	Include PC offset in function.
C8	one_bit_corrupt	Condensing	Single-bit errors in code from dump compared to archive copy.
C9	image_corrupt	Condensing	Multi-word errors in code from dump compared to archive copy.
C10	pc_misaligned	Condensing	PC isn't aligned to an instruction.
C11	malware_identified	Condensing	Contains known malware.
C12	old_image	Condensing	Known out-of-date program.
C13	pool_corruptor	Condensing	Known program that severely corrupts heap.
C14	bad_device	Condensing	Identify known faulty devices.
C15	over_clocked_cpu	Condensing	Identify over-clocked CPU.
C16	heap_corruption	Condensing	Heap function failed with corrupt heap.
C17	exception_subcodes	Expanding	Separate invalid-pointer read from write, etc.
C18	custom_extensions	Expanding	Output of registered third-party WER plug-in invoked based target program

Figure 5. Top Heuristics for Classifying (run on server).

thread context records in the memory dump (heuristic C1). The algorithm walks each stack starting at the most recent frame, f_0, backward. Each frame, f_n, is assigned a priority, p_n, of 0 to 5 based on its increasing likelihood of being a root cause (heuristics C2 through C5). Frame f_n is selected only if $p_n > p_i$ for all $0 \leq i < n$. For core OS components, like the kernel, $p_n = 1$; for core drivers $p_n = 2$; for other OS code, like the shell, $p_n = 3$; and for most other frames $p_n = 4$. For frames known to have cause an error, such as any f_n where f_{n-1} is `assert`, $p_n = 5$. Priority $p_n = 0$ is reserved for functions known never to be the root cause of an error, such as `memcpy`, `memset`, and `strcpy`.

`!analyze` contains a number of heuristics to filter out error reports unlikely to be debugged (C8 through C15). For example, since we have copies of all Microsoft binaries (and some third-party binaries), `!analyze` compares the (few, and small) code sections in the memory dump against the archived copies. If there's a difference, then the client computer was executing corrupt code—not much reason to debug any further. `!analyze` categorizes these as one-bit-errors (likely bad memory), multi-word errors (likely a misdirected DMA), and stride-pattern errors (likely a DMA from a faulty device). As another example, kernel dumps are tagged if they contain evidence of known root kits (C11), out-of-date drivers (C12), drivers known to corrupt the kernel heap (C13), or hardware known to cause memory or computation errors (C14 and C15).

4. STATISTICS-BASED DEBUGGING

Perhaps the most important feature enabled by WER is statistics-based debugging. WER records data about a large percentage of all errors that occur on Windows systems into a single database. Programmers can mine the WER database to improve debugging more than would be possible with a simple, unstructured stream of error reports. Strategies which use the database to improve the effectiveness of debugging can be broken into five categories: prioritizing debugging effort, finding hidden causes, testing root cause hypotheses, measuring deployment of solutions, and watching for regressions.

We built WER to improve the effectiveness of debugging. The primary reason to collect large numbers of error reports is to help programmers know which errors are most prevalent. Programmers sort their buckets and start debugging with the bucket with largest volume of error reports. A more sophisticated strategy is to aggregate error counts for all buckets for the same function, select the function with the highest count, and then work through the buckets for the function in order of decreasing bucket count. This strategy tends to be effective as errors at different locations in the same function often have the same root cause. Or at the very least, a programmer ought to be aware of all known errors in a function when fixing it.

The WER database can be used to find root causes which are not immediately obvious from the memory dumps. For example, when bucketing points the blame at reputable code we search error reports to look for alternative explanations. One effective strategy is to search for correlations between the error and a seemingly innocent third party. In many cases we find a third party device driver or other plug-in that has a higher frequency of

occurrence in the error reports than in the general population. For example, we recently began receiving a large number of error reports with invalid pointer usage in the Windows event tracing infrastructure. An analysis of the error reports revealed that 96% of the faulting computers were running a specific third-party device driver. With well below 96% market share (based on all other error reports), we approached the third party and they ultimately found the bug in their code. By comparing expected versus occurring frequency distributions, we similarly have found hidden causes from specific combinations of modules from multiple third-parties and from buggy hardware (in one case a specific hard disk model). A similar strategy is "stack sampling" in which error reports for similar buckets are sampled to determine which functions, other than the obvious targets, occur frequently on the thread stacks.

The WER database can be used to test programmer hypotheses about the root causes of errors. The basic strategy is to construct a debugger test function that can evaluate a hypothesis on a memory dump, and then apply it to thousands of memory dumps to verify that the hypothesis is not violated. For example, one of the Windows programmers was recently debugging an issue related to the plug-and-play lock in the Windows I/O subsystem. We constructed an expression to extract the current holder of the lock from a memory dump and then ran the expression across 10,000 memory dumps to see how many of the reports had the same lock holder. One outcome of the analysis was a bug fix; another was the creation of a new heuristic for `!analyze`.

A recent use of the WER database is to determine how widely a software update has been deployed. Deployment can be measured by absence, measuring the decrease in error reports fixed by the software update. Deployment can also be measured by an increase presence of the new program or module version in error reports for other issues.

Finally, both Microsoft and a number of third parties use the WER database to check for regressions. Similar to the strategies for measuring deployment, we look at error report volumes over time to determine if a software fix had the desired effect of reducing errors. We also look at error report volumes around major software releases to quickly identify and resolve new errors that may appear with the new release.

5. SYSTEM DESIGN

WER is a distributed system. Client-side software detects an error condition, generates an error report, labels the bucket, and reports the error to the WER service. The WER service records the error occurrence and then, depending on information known about the particular error, might request additional data from the client, or direct the client to a solution. Programmers access the WER service to retrieve data for specific error reports and for statistics-based debugging.

Error	Reporting Trigger
Kernel Crash	Crash dump found in page file on boot.
Application Crash	Unhandled process exception.
Application Hang	Failure to process user input for 5 seconds.
Service Hang	Service thread times out.
Install Failure	OS or application installation fails.
AppCompat Issue	Program calls deprecated API.
Custom Error	Program calls WER APIs.
Timer Assert*	User scenario takes longer than expected.
Ship Assert*	Invariant violated in program code.

Figure 6. Errors reported by WER.
*Asserts require custom code in each program

API	Description
`WerReportCreate(type)`	Initiate an error report.
`WerReportAddDump(r,p,opts)`	Include a minidump.
`WerReportAddFile(r,f,opts)`	Include a file.
`WerReportSubmit(r,opts)`	Submit report to service.
`WerRegisterFile(f,opts)` `WerRegisterMemoryBlock(p,c)`	Register a file or memory region for inclusion in future error reports for this process.
`KeRegisterBugCheckReason-Callback(f)`	Registers kernel-mode call-back to provide data for future crash reports.

Figure 7. Key WER Client APIs.

5.1. Generating an Error Report

Error reports are created in response to OS-visible events such as crashes, hangs, and installation failures (see Figure 6), or directly by applications calling a set of APIs for creating and submitting error reports (see Figure 7). For example, the default user-mode unhandled exception filter triggers an error report on program failure, the kernel triggers a report if a process runs out of stack, and the shell triggers a report if a program fails to respond for 5 seconds to user input. Once a report is triggered, the `werfault.exe` service is responsible for securing user authorization and submitting the error report to the WER servers. Reports are submitted immediately or queued for later if the computer is not connected to the Internet.

An important element of WER's progressive data collection strategy is the *minidump*, an abbreviated memory dump [21]. Minidumps are submitted for all kernel crashes when requested for program errors. Minidumps contain registers from each processor, the stack of each thread (or

processor for kernel dumps), the list of loaded modules, selected data sections from each loaded module, small areas of dynamically allocated memory that have been registered specifically for minidump collection, and 256 bytes of code immediately surrounding the program counter for each thread. A minidump does not include entire code sections; these are located by WER out of band using the information in the list of loaded modules.

5.2. Communication Protocol

WER clients communicate with the WER service through a four stage protocol that minimizes the load on the WER servers and the number of clients submitting complete error reports. In the first stage, the WER client issues an unencrypted `HTTP/GET` to report the bucket label for the error and to determine if the WER service wants additional data. In the second stage, the client issues an encrypted `HTTPS/GET` to determine the data desired by the WER service. In the third stage, the client pushes the data requested, in a CAB file, to the service with an encrypted `HTTPS/PUT`. Finally, in the fourth stage the client issues an encrypted `HTTPS/GET` to signal completion, and request any known solutions to the error.

WER is optimized based on the insight that most errors are common to many clients. The division between stages eliminates the need for per-connection state on incoming servers. Separating Stage 1 allows the protocol to terminate at the conclusion of Stage 1 if WER has already collected enough data about an error (the case in over 90% of error reports). Stage 1, being static HTML, is *very* low cost to reduce the load on WER servers and achieve scale. Stage 1 does not transmit customer data so we can use unencrypted `HTTP/GET` as it is the cheapest operation on stock web servers. Error reports from Stage 1 are counted daily by offline processing of the HTTP server logs, and recorded with a single database update per server per day. Finally, separating Stage 2 from Stage 1 reduces the number of read requests on a shared database because the Stage 1 response files can be cached on each front-end IIS server.

5.3. Service

Errors collected by WER clients are sent to the WER service. The WER service employs approximately 60 servers connected to a 65TB storage area network that stores the error report database and a 120TB storage area network that stores up to 6 months of raw CAB files (see Figure 8). The service is provisioned to receive and process well over 100 million error reports per day, which is sufficient to survive correlated global events such as Internet worms.

Requests enter through twenty *Front-End* IIS servers operating behind a TCP load balancer. The IIS servers handle all stages of the WER protocol. Stage 1 requests are

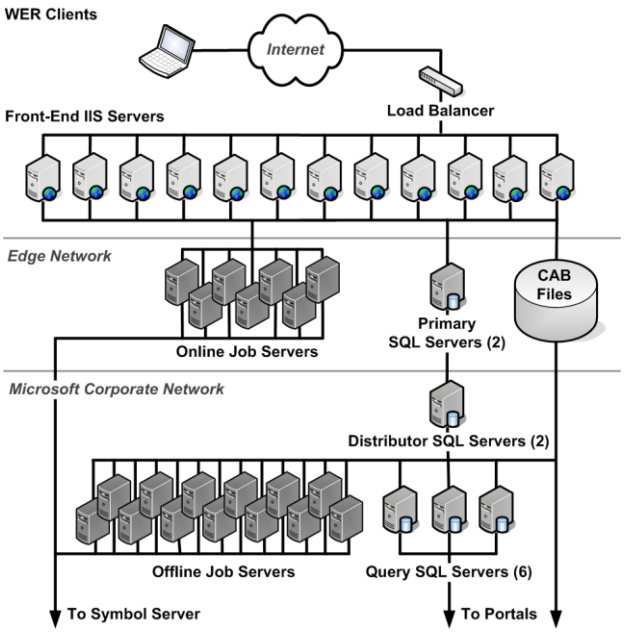

Figure 8. WER Servers.

resolved with the stock HTTP/GET implementation on static pages. Other stages execute ASP.NET code. The IIS servers store bucket parameters and bucket counts in the *Primary* SQL servers. The IIS servers save CAB files for incoming reports directly to the SAN. Data from the Primary servers are replicated through a pair of *Distributor* SQL servers to six *Query* SQL servers, which are used for data mining on the error reports. This three-tiered SQL design separates data mining on the Query servers from data collection on the Primary servers, maximizing overall system throughput.

Error reports are processed by seven *Online Job s*ervers and sixteen *Offline Job* servers. Online Job servers help clients label kernel crashes (blue screens) from minidumps. Offline Job servers classify error reports with `!analyze` as additional data become available. Offline Job servers also perform tasks such as aggregating hit counts from the IIS server logs.

While not strictly a component of WER, the Microsoft Symbol Server [24] service helps immensely by giving `!analyze` access to OS and application debugging symbols. Symbol Server contains an index of debugging symbols (PDB files) for every release (including betas, updates and service packs) of every Microsoft program by module name, version, and binary hash value. As a best practice, Microsoft teams index symbols for every daily build into the Symbol Server. Internal copies of the debug symbols are annotated with URLs to the exact sources used to create each program module. Code built-into the debugger [22] hosting `!analyze` retrieves the debugging

symbols from Symbol Server and source files from the source repositories on demand.

5.4. Acquiring Additional Data

While many errors can be debugged with simple memory dumps, others cannot. Using the WER portal, a programmer can create a "data wanted" request. The request for data is noted in the Primary SQL servers and the Stage 1 static page is deleted. On subsequent error reports the WER service will ask clients to collect the desired data in a CAB and submit it. The set of additional data collectable by WER has evolved significantly over time. Additional data the programmer can request include:

- complete process memory dumps (including all dynamically allocated memory)
- live dumps of kernel memory (including kernel thread stacks) related to a process
- minidumps of other processes in a wait chain
- minidumps of other processes mapping a shared module
- named files (such as log files)
- named registry keys (such as program settings)
- output from a Windows Management Instrumentation (WMI) query (such as data from the system event log).

Beyond gathering additional data, the Stage 2 server response can ask the WER client to enable extended diagnostics for the next run of a program or driver. One extended diagnostic is leak detection, which enables two changes in execution. First, during execution of the process, a call stack is recorded for each heap allocation. Second, when the process exits, the OS performs a conservative garbage collection to find allocations in the heap unreachable from program data sections; these allocations are leaks. Error reports are submitted for leaked allocations using the recorded call stack.

A recent sample showed that the WER servers had 1,503 buckets with one-off requests for additional full memory dumps, 3 for additional files, 349 for WMI queries, and 18 requests to enable extended driver diagnostics for the next boot cycle. By default, WER attempts to collect 3 memory dumps every 30 days for each bucket, but needs can vary dramatically. On the extreme, one team collected 100,000 dumps for a single bucket to debug a set of hangs in their program. Teams can also establish blanket data request policies for a range of buckets.

6. EVALUATION AND MEASUREMENTS

We evaluate WER's scalability, its effectiveness at helping programmers find bugs, and the effectiveness of its core bucketing heuristics. Our evaluation of WER concludes with a summary of additional data learned through WER.

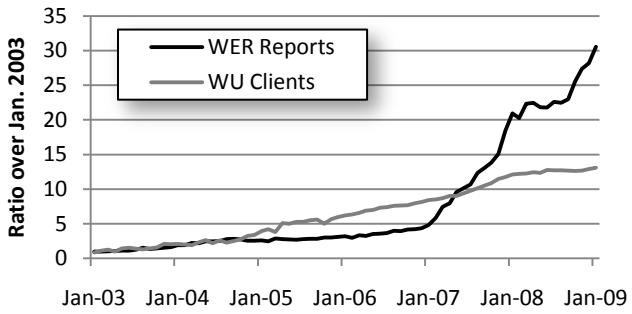

Figure 9. Growth of Report Load over 6 Years.

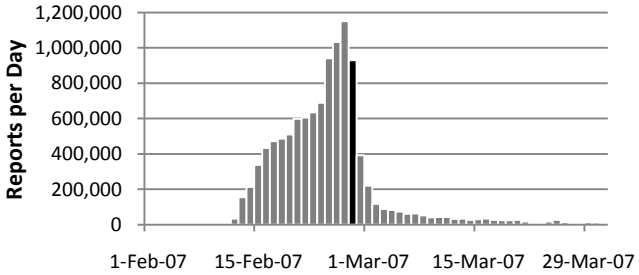

Figure 10. Renos Malware: Number of error reports per day. Black bar shows when the fix was released through WU.

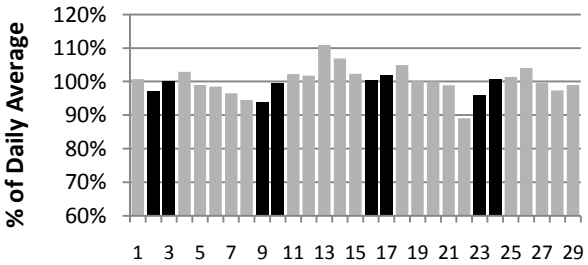

Figure 11. Daily Report Load as % of Average for Feb. 2008.
Black bars show weekends.

6.1. Scalability

WER collected its first million error reports within 8 months of its deployment in 1999. Since then, WER has collected billions more. From January 2003 to January 2009, the number of error reports processed by WER grew by a factor of 30. By comparison, the number of clients connecting to Windows Update (WU) [15] in the same period grew by a factor of 13 (see Figure 9).

The growth in reports has been uneven. The adoption of Windows XP SP2 (starting in August 2004) pushed down the number of errors *experienced* per user—due to bugs corrected with WER—while the likelihood that any error was reported did not increase. With Windows Vista, we

made a concerted effort to increase the likelihood that any error was reported and to increase the classes of errors *reported*. Our efforts there paid off. The adoption of Vista (starting in October 2006) dramatically pushed up both the likelihood of any error being reported and the classes of application errors reported, while the number of errors experienced per user continued to drop. Vista added detailed reporting for previously un-reported events such as hangs in daemons, application installation failures, and non-fatal behaviors such as application memory leaks. The opt-in rate for submission of reports almost doubled with a new feature that allowed one-time opt-in for all reports. From instrumentation on over 100,000 end-user computers, we believe that 40% to 50% (of fewer classes) of error reports on XP are submitted versus 70% to 80% (of more classes) of error reports on Vista.

To accommodate globally correlated events, the WER service is over-provisioned to process at least 100 million error reports per day. For example, in February 2007, users of Windows Vista were attacked by the Renos malware. If installed on a client, Renos caused the Windows GUI shell, `explorer.exe`, to crash when it tried to draw the desktop. The user's experience of a Renos infection was a continuous loop in which the shell started, crashed, and restarted. While a Renos-infected system was useless to a user, the system booted far enough to allow reporting the error to WER—on computers where automatic error reporting was enabled—and to receive updates from WU.

As Figure 10 shows, the number of error reports from systems infected with Renos rapidly climbed from zero to almost 1.2 million per day. On February 27, shown in black in the graph, Microsoft released a Windows Defender signature for the Renos infection via WU. Within three days enough systems had received the new signature to drop reports to under 100,000 per day. Reports for the original Renos variant became insignificant by the end of March. The number of computers reporting errors was relatively small: a single computer reported 27,000 errors, but stopped after the automatic updated.

Like many large Internet services, WER experiences daily, weekly, and monthly load cycles (see Figure 11). The variances are relatively small, typically in the range of ±10% from average.

6.2. Finding Bugs

WER augments, but does not replace, other methods for improving software quality. We continue to apply static analysis and model-checking tools to find errors early in the development process [2, 3]. These tools are followed by extensive testing regimes before releasing software to users. WER helps to us rank all bugs and to find bugs not exposed through other techniques. The Windows Vista

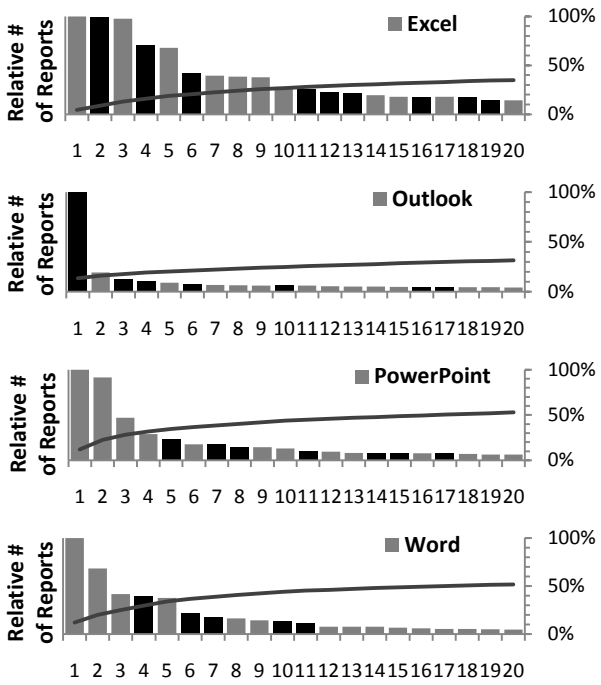

Figure 12. Relative Number of Reports per Bucket and CDF for Top 20 Buckets from Office 2010 ITP.
Black bars are buckets for bugs fixed in 3 week sample period.

Program	1st	2nd	3rd	4th
Excel	0.227%	1.690%	9.80%	88.4%
Outlook	0.058%	0.519%	6.31%	93.1%
PowerPoint	0.106%	0.493%	7.99%	91.4%
Word	0.057%	0.268%	7.95%	91.7%

Figure 13. Percentage of Buckets per Quartile of Reports.
A small number of buckets receive most error reports.

programmers fixed 5,000 bugs found by WER in beta deployments after extensive static analysis.

Compared to errors reported directly by humans, WER reports are more useful to programmers. Analyzing data sets from Windows, SQL, Excel, Outlook, PowerPoint, Word, and Internet Explorer, we found that a bug reported by WER is 4.5 to 5.1 times more likely to be fixed than a bug reported directly by a human. Error reports from WER document internal computation state whereas error reports from humans document external symptoms.

Our experience across many application and OS releases is that error reports follow a Pareto distribution with a small number of bugs accounting for most error reports. As an example, the graphs in Figure 12 plot the relative occurrence and cumulative distribution functions (CDFs)

for the top 20 buckets of programs from the Microsoft Office 2010 internal technical preview (ITP). The goal of the ITP was to find and fix as many bugs as possible using WER before releasing a technical preview to customers. These graphs capture the team's progress just 3 weeks into the ITP. The ITP had been installed by 9,000 internal users, error reports had been collected, and the programmers had already fixed bugs responsible for over 22% of the error reports. The team would work for another three weeks collecting error reports and fixing bugs, before releasing a technical preview to customers.

The distributions of error reports across buckets found in the Office 2010 ITP (see Figure 13) is common to the WER experience. Ranking buckets by number of error reports, the first quartile of error reports occupy significantly less than 1% of the buckets. The distribution has a very long tail; the last quartile of error reports account for 88% to 93% of the buckets. Given finite programmer resources, WER helps focus effort on the bugs that have the biggest impact on the most users.

Over successive service packs, the distribution of error reports to buckets for any program flattens out as a programming team "climbs down" its error curve—finding and fixing the most frequently encountered bugs. Figure 14 plots the cumulative distribution of error reports for the top 500 buckets, by error report volume, for Windows Vista and Vista Service Pack 1. The top 500 buckets account for 65% of all error reports for Vista and for 58% of all error reports for Vista SP1.

With WER's scale, even obscure Heisenbugs [17] can generate enough error reports for isolation. Early in its use WER helped programmers find bugs in Windows NT and Office that had existed for over five years. These failures were hit so infrequently to be impossible to isolate in the lab, but were automatically isolated by WER. A calibrating experiment using a pre-release of MSN Explorer to 3.6 million users found that less than 0.18% of users see two or more failures in a 30 day period.

An informal historical analysis indicates that WER has helped improved improve the quality of many classes of third-party kernel code for Windows. Figure 15 plots the frequency of system crashes for various classes of kernel drivers for systems running Windows XP in March 2004, March 2005, and March 2006, normalized against system crashes caused by hardware failures in the same period. Assuming that the expected frequency of hardware failures remained roughly constant over that time period (something we cannot prove with WER data), the number of system crashes for kernel drivers has gone down every year except for two classes of drivers: anti-virus and storage.

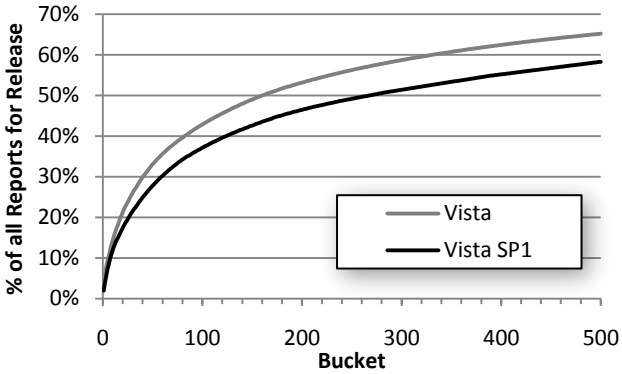

Figure 14. CDFs of Error Reports for the Top 500 Buckets for Windows Vista and Vista SP1. CDF curves flatten as buckets with the most error reports are fixed.

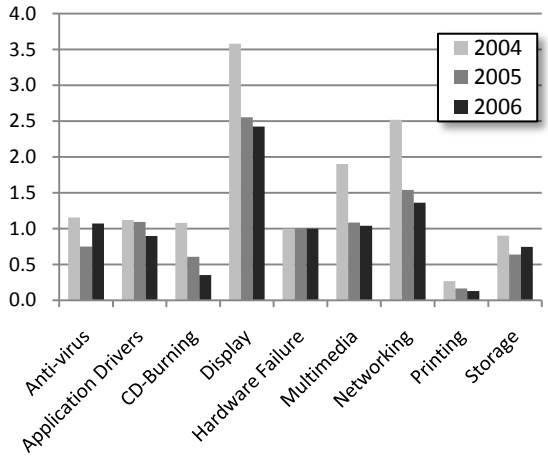

Figure 15. Crashes by Driver Class Normalized to Hardware Failures for Same Period.

While we have no explanation for the rise in failures caused by storage drivers from March 2005 to March 2006, the anti-virus results are not unexpected. Unlike the typical device driver, which gets released with a device and then improves as updates are needed to resolve errors, anti-virus providers are under constant pressure to add new features, to improve, and to rewrite their software. The resulting churn in anti-virus driver code results in periodic outbursts of new errors leading to the divergence from the general improvement trend.

As software providers have begun to use WER more proactively, their error report incidence has reduced dramatically. For example, in May 2007, one kernel-mode provider began to use WER for the first time. Within 30 days the provider had addressed the top 20 reported issues for their code. Within five months, as WER directed users

Program	In Second Bucket	In One-Hit Bucket	Combined
Excel	17.2%	4.4%	21.6%
Outlook	7.7%	10.0%	17.7%
PowerPoint	30.6%	6.2%	36.8%
Word	19.3%	8.8%	28.1%

Figure 16. Percentage of all Reports for Office 2010 ITP in Second or One-Hit Buckets.

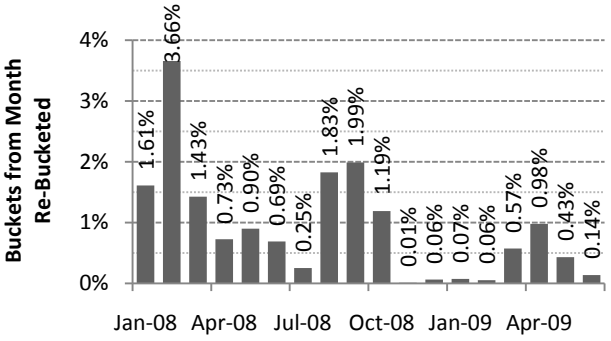

Figure 17. Percentage of Kernel Crashes later Re-bucketed.

to pick up fixes, the percentage of all kernel crashes attributed to the provider dropped from 7.6% to 3.8%.

6.3. Bucketing Effectiveness

The ideal bucketing algorithm would map all error reports caused by the one bug into one unique bucket with no other bugs in that bucket. We know of no such algorithm. Instead WER employs a set of bucketing heuristics for labeling and classifying. Here we evaluate how far WER's heuristics are from the ideal and the relative merits of the key bucketing heuristics.

We know of two forms of weakness in the WER bucketing heuristics: weaknesses in the condensing heuristics, which result in mapping reports from a bug into too many buckets, and weaknesses in the expanding heuristics, which result in mapping more than one bug into the same bucket. Figure 16 estimates an upper bound on the weakness in our condensing heuristics looking at error reports from Microsoft Office 2010 ITP. The first column lists the percentage of error reports placed into some bucket other than the primary bucket for a bug. These error reports were identified when a second bucket was triaged by programmers and found to be caused by the same bug as the first bucket. The second column lists the percentage of error reports in buckets containing exactly one error report—we call these "one-hit wonders". While some of these may be legitimate one-time events, ten years of experience with large sample sets has taught us that real

Contribution	Heuristic
67.95%	L7 `module_offset`
9.10%	L4 `module_name`
5.62%	L1 `program_name`
4.22%	L2 `program_version`
1.90%	L3 `program_timestamp`
1.02%	L8 `exception_code`
0.89%	L6 `module_timestamp`
0.34%	L5 `module_version`
8.96%	All other labeling heuristics.

Figure 18. Ranking of Labeling Heuristics by % of Buckets.

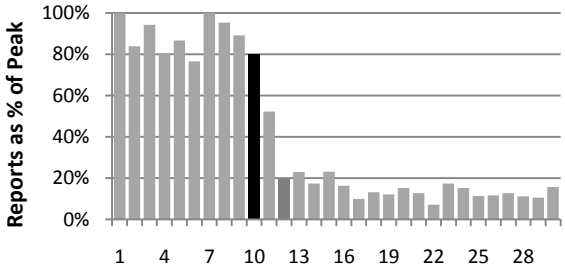

Figure 19. Crashes/Day for a Firmware Bug. Patch was released via WU on day 10.

bugs are *always* encountered more than once. We therefore assume that all one-hit wonders were inaccurately bucketed. For PowerPoint, as many as 37% of all errors reports may be incorrectly bucketed due to poor condensing heuristics.

To a much smaller degree, bucketing heuristics err by over-condensing, placing error reports from multiple bugs into a single bucket. All our improvements to `!analyze` over the last few years have reduced the number of these bucket collisions from classification. Figure 17 plots the percentage of kernel crash error reports from each month from January 2009 to June 2009 that were re-bucketed as we improved the bucketing heuristics. Because we retain minidumps for all kernel crashes for six months, whenever we update classifying heuristics we run `!analyze` on the affected portion of the minidump archive to re-bucket reports. Examples of recent changes to the heuristics include identifying system calls that returned with a thread interrupt priority level set too high and identifying if a crash during a system hibernate resulted from a cross-driver deadlock or from a single-driver fault.

Figure 18 ranks the relative importance of a number of key bucket labeling heuristics. For each heuristic, we measured the percentage of buckets receiving error reports because of the heuristic. The sample set had 300 million error reports

resulting in 25 million buckets. By far the most important labeling heuristic is `module_offset` (L7).

While not ideal, WER's bucketing heuristics are in practice effective in identifying and quantifying the occurrence of errors caused by bugs in both software and hardware. In 2007, WER began receiving crash reports from computers with a particular processor. The error reports were easily bucketed based on an increase in system machine checks and processor type. When Microsoft approached the processor vendor, the vendor had already discovered and documented externally the processor issue, but had no idea it could occur so frequently until presented with WER data. The vendor immediately released a microcode fix via WU—on day ten, the black bar in Figure 19—and within two days, the number of error reports had dropped to just 20% of peak.

6.4. Measurements

Figure 20 summarizes participation by hardware and software providers in the WER third-party program. Most providers register their programs and modules so that they may receive error reports. A smaller number of providers also register solutions to direct users to fixes for bugs.

Figure 21 summarizes size and churn within the Windows deployed base. As part of each kernel-mode error report, WER catalogs the devices and drivers on the computer. The list contains the manufacturer part number for each device, identified by PNP ID[3]. This list is used to isolate errors related to specific hardware combinations, such as when one device fails only in the presence of a specific second device. Each minidump also contains a list of modules loaded into the process or kernel. We can use the devices and loaded modules lists to calibrate the Windows deployed base. For example, driver identifiers collected in 2005 show that on average 25 new drivers and 100 revised drivers are released every day. To date, WER has encountered over two million unique device PNP IDs and over 17.5 million unique module names.

Providers using WER	> 700
Programs registered by providers with WER	6,956
Modules registered by providers with WER	299,134
Providers with registered Solutions	175
Registered Solutions	1,498

Figure 20. Third-Party use of WER.

Average # of *new* drivers released per day	25
Average # of *revised* drivers released per day	100
Unique device PNP IDs identified	> 2,000,000
Unique program binaries identified	> 17,500,000

Figure 21. Drivers and Hardware Encountered by WER.

7. DISCUSSION

7.1. OS Changes to Improve Debugging

Each release of Windows has incorporated changes based on our experience with WER. We've already mentioned some of these changes: the addition of the `WerReport` APIs, particularly `WerRegisterMemoryBlock` and `WerRegisterFile`, to enable program tuning of error-report content; the addition of kernel support for identifying thread wait chains and the `GetThreadWaitChain` API to debug hung programs; and support for process-specific dumps of live kernel memory.

Other changes to improve post-mortem debugging with dumps have included disabling frame-pointer omission when compiling Windows; rewriting the kernel exception handler to invoke `werfault.exe` when structured exception handling fails so errors are reported even on stack overflows; checking zeroed pages for non-zero bits; and collecting hardware model numbers and configuration during boot to a registered area of memory for collection in kernel dumps.

The decision to disable frame-pointer omission (FPO), in Windows XP SP2, was originally quite controversial within Microsoft. FPO suppresses the creation of frame pointers on the stack, instead using offsets from the `ESP` register to access local variables. FPO speeds function calls as frame pointers need not be created on the stack and the `EBP` register is free for local use. Programmers believed FPO was crucial to performance. However, extensive internal testing across a wide range of both desktop and server benchmarks showed that FPO provided no statistically provable benefit but significantly impaired post-mortem debugging of unexpected call stacks. Ultimately it was decided that the benefit of improving post-mortem debugging outweighed the cost of disabling FPO.

The kernel maintains a list of free pages, zeroed when the CPU is idle. On every sixteenth page allocation, the Vista kernel examines the page content and asynchronously creates an error report if any zero has changed to a one. Analysis of these corrupt "zero" pages has located errors in hardware—like a DMA engine that only supported 31-bit addresses—and errors in firmware—like a brand of laptop with a bad resume-from-sleep firmware.

7.2. Improvements in Kernel Minidumps

Section 5.1 outlined the current contents of a WER minidump, which contains registers, stacks, a list of loaded

[3] PNP IDs are model numbers, not serial numbers. All machines with the same hardware configuration will have the same set of PNP IDs.

modules, selected data sections, areas of registered memory, and 256 bytes of code immediately surrounding the program counter for each thread.

Kernel minidumps were introduced with Windows XP and extend the lower limit of WER's progressive data collection strategy. Before XP, Windows supported two types of OS core dump: *kernel dumps* contained only kernel-mode pages and *full dumps* contained all memory pages. XP minidumps were designed to create a much smaller dump that would still provide decent problem characterization. In addition to the minidump, XP crash reports also include a `sysdata.xml` file which provides the PNP ID of the hardware devices, and device driver vendor names as extracted from driver resources. Kernel minidumps allowed us to count and characterize kernel-mode crashes in the general population for the first time. In some cases, a crash could be debugged directly from the minidump. But in most cases, the minidumps helped prioritize which issues to pursue via complete kernel crash dumps. By default XP creates a minidump on system crash, but can be reconfigured to collect kernel or full dumps.

Three improvements were made in Windows XP Service Pack 2. First, in crashes due to pool corruption, the previous minidump didn't contain the corrupted page, so corruption patterns could not be characterized or counted. For SP2, the minidump generation code was changed to explicitly add this page. Second, SP2 added most of the system management BIOS table to the minidump, including the OEM model name and BIOS version. Third, SP2 added information to help identify if the crash report was from a system with an over-clocked CPU including a measurement of the running CPU frequency along with the processor brand string, e.g. "Intel(R) Core(TM)2 Duo CPU T5450 @ 1.66GHz", in the CAB file. Over-clocking is detected by comparing the frequencies. Initially, we were surprised to learn that nearly 5% of the crash reports received during SP2 beta were from over-clocked CPUs. After SP2 was released, that rate fell to around 1%. This pattern has repeated: over-clocking is more prevalent among beta users than in the overall population. Today the rate for the general population bounces between 1-2%, except for 64-bit Windows systems, where the rate is around 12%.

Windows Vista made two more enhancements to kernel crash reports. First, Vista creates a complete kernel dump on crash and after reboot extracts and reports a minidump from the kernel dump. If more data is needed, WER will ask for the complete kernel dump. This application of progressive data collection avoids a worse case in XP, where we would change the default from minidump to kernel dump, but the computer might never again experience the same crash. Second, XP minidumps proved deficient for crashes attributed to graphics drivers. The XP minidump design allocated up to 32KB of secondary data that could be filled in by drivers that registered to add data when assembling the crash dump. This 32KB allowed basic GPU information to be included in the minidump, but in practice the limit was too small, especially as GPUs increased their video RAM. Vista increased the size limit of the secondary data to 128KB, and again to 1MB in SP2. The size has remained the same for Windows 7.

8. RELATED WORK

Following WER's lead, several systems implement one-click error reporting via the Internet. Crash Reporter [1], BugBuddy [4], Talkback [25], and Breakpad [16] collect compressed memory dumps and report to central error repositories. Talkback provides limited error report correlation. WER predates these systems, vastly exceeds their scale, and remains unique in its use of automated diagnosis, progressive data collection, and directing users to existing fixes.

To improve user anonymity, Castro *et al.* [7] use symbolic execution to systematically replace memory dumps with the condition variables that trigger an error. While their system reduces the size of error reports and may improve anonymity, it requires extensive computation on the client. Scrash [5] is an *ad hoc* mechanism that lets programs exclude sensitive memory regions from collection. WER protects user privacy by avoiding the explicit collection of identifying information, minimizing the memory collected in dumps, and avoiding the transfer of user data when reporting 90% of all errors through client-side labeling.

The use of run-time instrumentation to diagnose or suppress errors has been discussed broadly in the literature. Systems have resorted to statistical sampling, delayed triggering, or hoped-for hardware changes to minimize instrumentation costs. Liblit *et al.* [20] addresses the error diagnosis problem by remotely collecting data of all executions of a program, through statistical sampling. They show samples can be combined using logistic regression to find the root cause of an error and that samples can be collected with runtime overheads of a few percent. Triage [31] uses checkpointing and re-execution to identify causes of bugs on the client at program failure, to cut debugging time and reduce the transfer of non-anonymized data. By default, Triage is enabled only after a failed run because it imposes a runtime overhead. Vigilante [9] and Argos [27] suppress failures on some computers by detecting security exploits on other computers and generating filters to block bad input or executions. Failure-oblivious computing [29] and Rx [28] hide failures by either altering the execution environment or fabricating data. Dimmunix [18] suppresses deadlocks by altering lock acquisition ordering based on locking signatures resulted in deadlocks previously on the same computer. The runtime overheads of these systems

and other engineering challenges have prevented their widespread deployment; however, we see great potential for combining these technologies with WER. For example, WER could enable failure-oblivious computing or Triage when correlating data suggests a computer is likely to hit the same error again.

WER avoids runtime instrumentation, except in limited cases where explicitly requesting extended diagnostics on a small number of clients. Data from a 30-day study of an MSN Explorer deployment reinforces this decision to avoid run-time instrumentation. In the study, less than 1% of users encountered any error. Of those, less than a quarter encountered a second error of any kind. Any system using run-time instrumentation will likely pay a high aggregate cost for any bugs found.

The best techniques for isolating bugs are systems based on static analysis and model checking [2, 3, 6, 11]. These systems have the distinct advantage that they can be used as part of the development cycle to detect bugs *before* they are encountered by users or testers. Results from the development of Windows Vista, mentioned in Section 1, suggest that present static analysis and model checking tools will find at least 20 bugs for every one bug found by WER. However, the bugs found by WER are crucial as they are the bugs which have slipped past tools in the development cycle.

As a widely deployed system, WER has been acknowledged and described narrowly by researchers outside our team. Murphy [26] summarized the history and motivation for automated crash reporting using WER as an example. Ganapathi and Patterson [14] used the Corporate Error Reporting (CER) feature of WER, including `!analyze`, to collect and classify roughly 2,000 crash reports, mostly from applications, across 200 computers at UC Berkeley. In a later report, Ganapathi *et al.* [13] classified the failing component in system crashes to find that over 75% of system failures are caused by poorly written device drivers.

Finally, the use of post-mortem core dumps to diagnose computer malware dates to the original Internet Worm, as documented by Rochlis and Eichin [30]. WER's benefit is that the collection and diagnosis of these error reports occurs with little human effort, making it feasible to quickly identify and respond to new attacks.

9. CONCLUSION

WER has changed the process of software development at Microsoft. Development has become more empirical, more immediate, and more user-focused. Microsoft teams use WER to catch bugs after release, but perhaps as importantly, they use WER during pre-release deployments to internal and beta testers. While it doesn't make debugging in the small significantly easier (other than perhaps providing programmers with better analysis of core dumps), WER has enabled a new class of debugging in the large. The statistics collected by WER help us to prioritize valued programmer resources, understand error trends, and find correlated errors. WER's progressive data collection strategy means that programmers get the data they need to debug issues, in the large and in the small, while minimizing the cost of data collection to users. Automated error analysis means programmers are not distracted with previously diagnosed errors and that users are made aware of fixes that can immediately improve their computing experience.

Our experience with the law of large numbers as applied to WER is that we will eventually collect sufficient data to diagnose even rare Heisenbugs [17]; WER has already helped identify such bugs dating back to the original Windows kernel. We have also used WER as an early warning system to detect malware attacks, looking at error reports from data execution exceptions and buffer overruns. Over the last five years, a team at Microsoft has analyzed error reports from WER to identify security attacks on previously unknown vulnerabilities and other security issues.

WER is the first system to provide users with an end-to-end solution for reporting and recovering from errors. WER provides programmers with real-time data about errors actually experienced by users and provides them with an incomparable billion-computer feedback loop to improve software quality.

10. ACKNOWLEDGEMENTS

Many talented programmers have contributed to WER as it has evolved over the last decade. Ben Canning, Ahmed Charles, Tom Coon, Kevin Fisher, Matthew Hendel, Brian T. Hill, Mike Hollinshead, Jeff Larsson, Eric Levine, Mike Marcelais, Meredith McClurg, Jeff Mitchell, Matt Ruhlen, Vikas Taskar, and Steven Wort made significant contributions to the WER client and service. Alan Auerbach, Steven Beerbroer, Jerel Frauenheim, Salinah Janmohamed, Jonathan Keller, Nir Mashkowski, Stephen Olson, Kshitiz K. Sharma, Jason Shay, Andre Vachon, Alexander (Sandy) Weil, and Hua Zhong contributed to the kernel crash portion of WER and the solution response subsystem. Hunter Hudson and Ryan Kivett made significant contributions to `!analyze`. Siamak Ahari, Drew Bliss, Ather Haleem, Michael Krause, Trevor Kurtz, Cornel Lupu, Fernando Prado, and Haseeb Abdul Qadir provided support across the Windows product teams.

Our thanks to the anonymous reviewers and to our shepherd, John Ousterhout, whose comments significantly improved the presentation of our work.

11. REFERENCES

[1] Apple Inc., CrashReporter. Technical Report TN2123, Cupertino, CA, 2004.

[2] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K. and Ustuner, A. Thorough Static Analysis of Device Drivers. In Proc.of the EuroSys 2006 Conference, Leuven, Belgium, 2006.

[3] Ball, T. and Rajamani, S.K. The SLAM Project: Debugging System Software via Static Analysis. In Proc.of the 29th ACM Symposium on Principles of Programming Languages, pp. 1-3, Portland, OR, 2002.

[4] Berkman, J. Bug Buddy. Pittsburgh, PA, 1999, http://directory.fsf.org/project/bugbuddy/.

[5] Broadwell, P., Harren, M. and Sastry, N. Scrash: A System for Generating Secure Crash Information. In Proc.of the 12th USENIX Security Symposium, pp. 273-284, Washington, DC, 2003.

[6] Bush, W.R., Pincus, J.D. and Sielaff, D.J. A Static Analyzer for Finding Dynamic Programming Errors. Software-Practice and Experience, 30 (5), pp. 775-802, 2000.

[7] Castro, M., Costa, M. and Martin, J.-P. Better Bug Reporting With Better Privacy. In Proc.of the 13th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, pp. 319-328, Seattle, WA, 2008.

[8] Corbató, F.J. and Saltzer, J.H. Personal Correspondence. 2008.

[9] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L. and Barham, P. Vigilante: End-to-End Containment of Internet Worms. In Proc.of the 20th ACM Symposium on Operating System Principles, pp. 133-147, Brighton, UK, 2005.

[10] Das, M. Formal Specifications on Industrial-Strength Code - From Myth to Reality. Invited Talk, Computer-Aided Verification, Seattle, WA, 2006.

[11] Engler, D., Chen, D.Y., Hallem, S., Chou, A. and Chelf, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In Proc.of the 18th ACM Symposium on Operating Systems Principles, pp. 57-72, Alberta, Canada, 2001.

[12] Everett, R.R. The Whirlwind I Computer. In Proc.of the 1951 Joint AIEE-IRE Computer Conference, pp. 70-74, Philadelphia, PA, 1951.

[13] Ganapathi, A., Ganapathi, V. and Patterson, D., Windows XP Kernel Crash Analysis. In Proc.of the 20th Large Installation System Administration Conference, pp. 149-159, Washington, DC, 2006.

[14] Ganapathi, A. and Patterson, D., Crash Data Collection: A Windows Case Study. In Proc.of the 2005 Intl. Conference on Dependable Systems and Networks, pp. 280-285, Yokohama, Japan, 2005.

[15] Gkantsidis, C., Karagiannis, T., Rodrigeuz, P. and Vojnovic, M. Planet Scale Software Updates. In Proc.of ACM SIGCOMM 2006, Pisa, Italy, 2006.

[16] Google Inc. Breakpad. Mountain View, CA, 2007, http://code.google.com/p/google-breakpad/.

[17] Gray, J. Why Do Computers Stop and What Can We Do About It. In Proc.of the 6th Intl. Conference on Reliability and Distributed Databases, pp. 3-12, 1986.

[18] Jula, H., Tralamazza, D., Zamfir, C. and Candea, G., Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In Proc.of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), pp. 295-308, San Diego, CA, 2008.

[19] Lee, I. and Iyer, R.K., Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In Digest of Paers of the Twenty-Third Intl. Symposium on Fault-Tolerant Computing (FTCS-23), Toulouse, France, 1993, IEEE.

[20] Liblit, B., Aiken, A., Zheng, A.X. and Jordan, M.I. Bug Isolation via Remote Program Sampling. In Proc.of the 2003 Conference on Programming Language Design and Implementation, pp. 141-154, San Diego, CA, 2003.

[21] Microsoft Corporation. DbgHelp Structures. In Microsoft Developer Network, Redmond, WA, 2001.

[22] Microsoft Corporation. Debugging Tools for Windows. Redmond, WA, 2008, http://www.microsoft.com/whdc/devtools/debugging.

[23] Microsoft Corporation. Plug and Play: Architecture and Driver Support. In Windows Hardware Developer Central, Redmond, WA, 2008.

[24] Microsoft Corporation. Use The Microsoft Symbol Server to Obtain Debug Symbol Files. Knowledge Base Article 311503, Redmond, WA, 2006.

[25] Mozilla Foundation. Talkback. Mountain View, CA, 2003, http://talkback.mozilla.org.

[26] Murphy, B. Automating Software Failure Recovery. ACM Queue, 2 (8), pp. 42-48, 2004.

[27] Portokalidis, G., Slowinska, A. and Bos, H. Argos: An Emulator for Fingerprinting Zero-day Attacks for Advertised Honeypots with Automatic Signature Generation. In Proc.of the EuroSys 2006 Conference, pp. 15-27, Leuven, Belgium, 2006.

[28] Qin, F., Tucek, J., Sundaresan, J. and Zhou, Y. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failure. In Proc.of the 20th ACM Symposium on Operating System Principles, Brighton, UK, 2005.

[29] Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T. and Beebee, W.S., Jr. . Enhancing Server Availability and Security Through Failure-Oblivious Computing. In Proc.of the 6th Symposium on Operating Systems Design and Implementation San Francisco, CA, 2004.

[30] Rochlis, J.A. and Eichin, M.W. With Microscope and Tweezers: The Worm from MIT's Perspective. Communications of the ACM, 32 (6), pp. 689-698, 1989.

[31] Tucek, J., Lu, S., Huang, C., Xanthos, S. and Zhou, Y. Triage: Diagnosing Production Run Failures at the User's Site In Proc.of the 21st ACM SIGOPS Symposium on Operating Systems Principles, pp. 131-144, Stevenson, WA, 2007.

[32] Walter, E.S. and Wallace, V.L. Further Analysis of a Computing Center Environment. Communications of the ACM, 10 (5), pp. 266-272, 1967.