Monitoring of Aging Software Systems affected by Integer Overflows

Domenico Cotroneo and Roberto Natella
Dipartimento di Informatica e Sistemistica
Università degli Studi di Napoli Federico II
Via Claudio 21, 80125, Naples, Italy
Email: cotroneo@unina.it, roberto.natella@unina.it

***Abstract*—Numerical aging-related bugs, which can manifest themselves as the accumulation of floating-point errors and the overflow of integers, represent a known but relatively neglected issue in the field of software aging and rejuvenation. Unfortunately, it is very difficult to avoid and to fix these bugs, since the rules of computer arithmetic and programming languages are often misunderstood or disregarded by programmers. Even though software rejuvenation can potentially mitigate these problems, its adoption is prevented by the lack of approaches for forecasting numerical software aging failures: in order to efficiently plan rejuvenation, the rate of numerical errors has to be known, or at least estimated. In this paper, we focus on software aging phenomena related to integer overflows. We present some examples of integer overflow issues of the MySQL open-source DBMS, and an approach for identifying symptoms of potential integer overflows by on-line monitoring.**

***Keywords*-Software aging; Software rejuvenation; Integer overflows; Numerical errors; Aging-related bugs**

I. Introduction

Software aging, that is, the gradual increase of the failure rate and performance degradation of long-running systems, is caused by bugs (namely, *aging-related bugs*) that manifest themselves in subtle, and often transient ways [1], [2], [3]. Unfortunately, avoiding these bugs through rigorous development and testing can be unfeasible. *Software rejuvenation* is a cost-effective approach to deal with software aging, by the proactive rollback of the system in a clean state before the occurrence of unplanned, and more costly, outages [4], which has been successfully adopted in many systems [5].

Among software aging issues analyzed in the literature, most of previous studies have dealt with the consumption of memory and other OS resources, and with degradation of response time [6]. Indeed, these issues are the most frequent manifestations of aging-related bugs, as showed by empirical studies [7]. Several strategies have been proposed to detect resource consumption and performance degradation trends and to forecast aging-related failures, in order to schedule software rejuvenation in a cost-effective way [8], [9], [10].

However, there is a type of aging-related bugs that represents a known but relatively neglected issue, namely *numerical aging-related bugs* [6]. A well-known example of numerical aging-related bug is the accident of the Patriot missile system [11]: the system kept time internally as an integer in a 24-bit register, counting tenths of seconds; whenever a target was spotted, this integer value was converted into a real value with an inaccuracy proportional to the system runtime, causing the system to miss the target. In general, numerical bugs are aging-related when they lead to the accumulation of numerical errors, such as floating-point rounding errors, or when their manifestation is triggered by the quantity and type of work performed by the system and/or by the total system runtime, such as in the case of some kinds of integer overflows.

Numerical aging-related bugs can represent an issue for many long-running software applications, such as industrial control systems and signal processing. Unfortunately, it is very difficult to avoid and to fix numerical bugs, since the rules of computer arithmetic and programming languages are often misunderstood or disregarded by programmers. Moreover, the analysis of stability and accuracy of numerical algorithms is a costly and difficult process. If numerical algorithms are not carefully designed and implemented, these bugs can lead to an erroneous and failure-prone state of the software system [12], [13]. Even though software rejuvenation can potentially mitigate these problems, its adoption is prevented by the lack of approaches for forecasting aging failures due to numerical errors: in order to efficiently plan rejuvenation, the rate of such errors has to be known, or at least estimated. Therefore, applying software rejuvenation to counteract numerical errors represents an open challenge.

In this paper, we focus on software aging phenomena related to integer overflows. We present some examples of integer overflow issues that affected the MySQL open-source DBMS, in order to understand numerical problems in a real-world system. We then describe an on-line monitoring approach for identifying symptoms of potential overflows, by periodic sampling of state variables of a running process.

The paper is organized as follows. In Section II, we discuss related work on integer overflow bugs. Section III describes some examples of integer overflow bugs in MySQL, and Section IV describes the proposed monitoring approach. Section V concludes the paper.

II. Related work

Integer-related bugs occur when programmers reason about infinitely ranged mathematical integers, while im-

plementing their designs with finite-precision data types. There has been extensive work on tools and libraries for mitigating integer overflows. Many of these efforts have been motivated by the fact that integer-related bugs can introduce security vulnerabilities: this may happen when an integer value from an untrusted source is used, for instance, in pointer arithmetic, as the bound of an array, or as an argument to a memory allocation function [14], [15]. Moreover, in languages such as C and C++, some kinds of integer overflow (e.g., overflows of unsigned integers) lead to an *undefined behavior*, that is, the programming language does not impose any rule when the operation overflows, leading to unpredictable results [12].

The approaches for mitigating integer overflows include static analysis [16], [17], dynamic analysis [15], [18], [19], which instruments a program in order to point out overflows during tests, and libraries that wrap integer operations and deal with overflows at run-time, by generating an exception or by masking the error [14], [20].

Compared to previous work, our perspective is to *prevent* the occurrence of integer overflows, by detecting aging symptoms at run-time and by scheduling rejuvenation, rather than *handling* overflows after they have occurred or *fixing* the underlying bugs. Thus, our focus is on numerical bugs that lead to software aging (numerical aging-related bugs) and on rejuvenation strategies for this specific class of bugs.

III. Numerical aging-related bugs

In this section, we analyze and discuss numerical aging-related bugs in the MySQL DBMS. In our previous work [21], we analyzed problem reports from the Linux 2.6 and MySQL 5.1 projects: we queried their publicly-available bug tracking systems in order to identify aging-related bugs in a subset of components of these projects, and to relate the occurrence of aging-related bugs with software complexity metrics. That study highlighted the occurrence of numerical aging-related bugs in both projects, where both bugs caused integer overflows. In this study, we performed an expanded analysis of bugs in MySQL, by searching for numerical aging-related bugs in all components of the DBMS server process and in all releases of MySQL. In order to focus our analysis on integer overflow issues, we performed a keyword-based search using the terms "integer overflow".

Among the results of our search, most of problem reports were not related to software aging, but were buffer and stack overflow problems. By analyzing the summary of problem reports, we identified a sample of numerical aging-related bugs, that allows us to describe some of the potential problems caused by integer overflows. It should be noted that our goal is not to quantitatively estimate the extent of numerical aging-related bugs, which would require an extensive and in-depth analysis of problem reports as in [21], [22], [23], but to describe and focus on some of the problems experienced by users and their possible solutions.

The first kind of numerical aging-related bugs that we consider is related to the size of database tables. A first example is bug #31732 [24], which is triggered when the number of tuples in a table exceeds $2^{32} - 1$, that is, the highest value for an unsigned 32-bit int variable. In this case, the result of the `count()` SQL function overflows (e.g., if the table has 2^{32} tuples, `count()` reports that the table has only one tuple), even if the tuples are actually stored in the database. This error, in turn, affects the results of SQL queries based on the `count()` function, and the applications performing these queries. In this scenario, an application would fail because of errors from a third-party component (the DBMS). The fault may be difficult to diagnose and to fix, and would require the assistance of MySQL developers to do so. This issue can be considered a software aging problem, since the likelihood of an integer overflow (and, in turn, the failure rate) increases with the amount of `insert` operations that have been performed.

An alternative, cost-effective strategy could be to adopt software rejuvenation, in order to avoid the occurrence of the error. In this case, however, simply restarting the DBMS server process would not be sufficient to avoid the overflow, since the number of tuples would not change after a process restart. To be effective, software rejuvenation should be applied on storage data, by performing a *database rotation*, that is, to move old tuples to another location (e.g., a data warehouse or off-line storage) in order to keep in the table the tuples that are strictly necessary for on-line transaction processing. This practice is common among system administrators for managing large log files, by periodically restarting them so that they do not grow without bounds. In a similar way, rotation can be applied to database tuples if data can be moved off-line when they get old. We found that this practice is adopted in a commercial Security Information and Event Management (SIEM) product [25], where system log data is collected in a database that is periodically rotated in order to keep only "active events" for security surveillance purposes. However, this operation has to be properly scheduled in order to avoid unnecessary overhead or outages.

Bug #43203 [26] is another example of numerical aging-related bug triggered by an excessive growth of tables. In this case, the DBMS server process crashes (due to the failure of an assertion) if the value of an integer key field (with auto-increment of the key at each `insert` operation) exceeds the maximum value allowed by its type; instead, it is expected that the DBMS should gracefully handle this event by returning an SQL error code when a query causes the key to overflow. In a similar way to the previous bug, failures could be prevented by rotating database tables.

It is important to note that even when the overflow of a field is gracefully handled by the DBMS (e.g., an error code is returned as expected), it may still cause the failure of the application that uses the DBMS, since the SQL error may not be correctly handled by the application (e.g., developers

may overlook the possibility of that SQL error). This kind of failure can be prevented by avoiding overflows.

Another type of problem due to numerical aging-related bugs is represented by the bug #42698 [27]. This bug causes the overflow of *status variables* in the DBMS server process. Status variables are adopted in MySQL (and in other DBMS products) by database administrators to analyze the behavior of the DBMS over time [28], [29]. These variables provide information about DBMS operations, such as the number of aborted connections, the amount of transferred data, the number of running threads, and several others. The value of some variables increases with the amount of operations processed by the DBMS, leading to an overflow after some days (in MySQL up to version 5.0.45, these variables were stored in 32-bit integer variables). This problem, in turn, can affect DBMS monitoring tools that are based on these variables and their users, such as database administrators. This was the case of users of MONyog, a platform for tuning and managing MySQL installations, that complained about wrong statistics provided by this tool [30], [31]. An indicator affected by this problem is the "percentage of full table scans", that is, the percentage of operations that require to sequentially inspect the tuples of a table [30]. Since this kind of operation may cause a significant processing overhead, these statistics are used to optimize SQL queries and databases, e.g., by introducing indexes. In the MONyog monitoring tool, this indicator is defined as

$$\textit{Perc. of Full Table Scans} = \frac{\textit{Handler\_read\_rnd\_next} + \textit{Handler\_read\_rnd}}{\left(\begin{array}{c}\textit{Handler\_read\_rnd\_next} + \textit{Handler\_read\_rnd} \\ +\textit{Handler\_read\_first} + \textit{Handler\_read\_next} \\ +\textit{Handler\_read\_key} + \textit{Handler\_read\_prev}\end{array}\right)} \quad (1)$$

where the values with the *Handler_* prefix represent integer status variables collected from a MySQL server about operations on table handlers, and the *Handler_read_rnd_next* and *Handler_read_rnd* variables represent the number of operations that require a full table scan. In a highly loaded DBMS, these variables can overflow after few days of execution, as pointed out by the users. Another indicator affected by overflow is the "cache hit rate", which represents the percentage of requests that read a key block from the cache and do not generate a physical read from the disk [31]. This indicator is adopted for tuning the size of the cache size for index blocks. It is defined as

$$\textit{Cache hit rate} = 1 - \frac{\textit{key\_reads}}{\textit{key\_read\_requests}} \quad (2)$$

where *key_reads* is the number of physical reads of index blocks that have been performed by the DBMS, and *key_read_requests* is the total number of reads of index blocks. When an overflow occurs, the result of these formulas may become obviously wrong and therefore unusable (e.g., a cache hit rate equal to -1804.52%), or may be misleading (e.g., the percentage of full table scans can raise and then abruptly drop after the overflow). To mitigate this issue, MySQL programmers adopted 64-bit variables. However, until the identification of the bug, the solution suggested by MONyog developers was to periodically reset the values of variables by executing the *flush status* command [31], which can be seen as a form of software rejuvenation.

In our analysis, we did not analyzed numerical aging-related bugs that involve floating point arithmetic. As showed in our earlier analysis [21], floating-point issues are unlikely in system software (such as operating systems and DBMSs), since they make little use of floating-point arithmetic. However, this kind of numerical aging-related bugs can be a more severe issue for different types of systems. Therefore, analyzing other types of systems is a worthy direction for future work.

IV. Monitoring for integer overflows

Although software rejuvenation can potentially mitigate integer overflow problems, its adoption is limited by the lack of an *aging indicator*, that is, a quantity that can be measured and that can be related to software aging phenomena of this kind [3], [6]. In the case of software aging phenomena that cause resource consumption or performance degradation, the amount of free resources or the average system response time are usually adopted as aging indicators, in order to forecast the occurrence of aging-related failures (e.g., by using time series and statistical data analysis) and plan software rejuvenation before the failure [8], [10]. It would be thus useful to have an aging indicator for integer overflows, in order to adopt a similar approach for this kind of problems.

Our idea is to periodically sample the value of long-lived integer variables, to estimate the expected time-to-overflow for each monitored variable (e.g., based on trend analysis), and to trigger rejuvenation according to the estimate. In particular, we monitor variables located in the heap or in the global data areas of a process, since their lifetime can be long enough (e.g., the whole execution of the program in the case of global variables) that their values can grow with time and be affected by aging, leading to an overflow. Global variables are typically allocated in the program address space at compile- and link-time, and their memory address can be obtained by analyzing information from compilers and linkers (e.g., "debug information" that is embedded in a program executable). Heap variables are allocated at run-time, and their address and their lifetime are typically unknown before executing the program. We consider long-lived heap variables that are referenced by pointers in the global data area, and neglect heap variables that are referenced by local variables, since it is likely that important data lasting for a long period have a global scope in order to be accessed by more than one procedure and more than one thread.

The schema of the proposed monitoring approach is showed in Figure 1. It consists of the following steps:

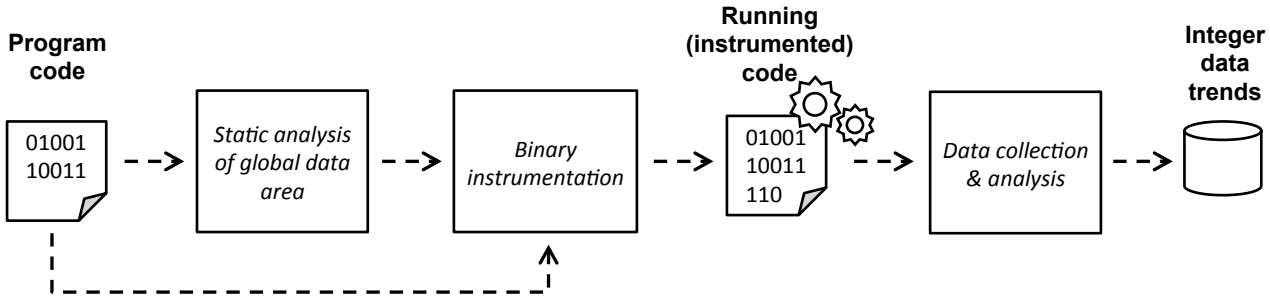

Figure 1. Monitoring approach for integer overflows.

1) We collect informations about global variables of the program, by performing a static analysis of the binary code and its debugging information. These informations include the list of global variables, their type, and their address. In this way, we can determine the address (i) global-scoped integer variables, and (ii) integer variables in the heap that are referenced by a global data structure.
2) The binary program is instrumented, by adding a procedure that is executed periodically and that samples the values of integer variables at each execution.
3) The instrumented program is monitored during its execution; the sampled values are analyzed in order to detect aging trends (e.g., using a statistical test [7]) and to forecast the expected time before a variable will reach a limit value (e.g., the maximum value allowed by the type of the variable).

We implemented a prototype monitoring tool using the SystemTap [32], which is a framework for the Linux OS that can introduce event handlers in both kernel and application code. Handlers can be triggered by the execution of a given code location (like a breakpoint of a debugger), or by a periodic timer; they can be used to collect information about variables of a process with low overhead and without modifying the source code of the program.

In the current prototype, we let the user to specify which variable to monitor, or we monitor all integer variables reachable from the global data area. However, there can be a non-negligible overhead if all of these integer variables are sampled. A potential future extension of the prototype could be to adopt an adaptive sampling, by sampling less frequently those variables that seldom change. Moreover, trend analysis could be performed on a fixed number of recent samples, in order to keep constant its overhead. Another limitation of the current prototype is the need for debug information. To perform instrumentation in the absence of debug information, the monitoring tool needs to be combined with approaches for reverse-engineering of data structures from binary code [33], [34].

We performed an experiment in which we reproduced and analyzed the effects of bug #42698. The experiment exercises the MySQL DBMS by executing repeatedly for 7 days the test cases from the MySQL test suite. During the experiment, we adopted the monitoring approach to sample the value of status variables, which are located in global integer variables of the MySQL server process. We periodically sampled 223 integer variables at the rate of one sample per minute, and saved the samples in a log file. After the experiment, we identified a subset of 8 variables that exceeded 2^{31} at least one time during the experiment, which are showed in Figure 2. The status variable *handler_read_rnd_next* is included in this subset. This variable overflowed for 3 times during the experiment, with a period of about 2 days. The *key_blocks_not_flushed* is another problematic status variable, which overflowed several times during the experiment. By searching for problem reports related to this variable, we found that the DBMS assigns a negative value to the variable due to a bug; the value is converted in a large unsigned integer, and the variable then overflows after some operations [35]. The remaining variables represent the amount of data processed by the DBMS (e.g., data written by the InnoDB storage engine), which can often exceed 4Gb in a highly loaded server.

In a second experiment, we monitored the size of tables using the proposed approach. We set up a database by using the Memory storage engine for MySQL [28], which stores data in memory in order to avoid disk I/O. We considered this storage engine due to its simple internal structure; moreover, this storage engine can be exposed to the problem of over-growing tables, since memory is a relatively scarce resource. We populated a database with three tables, and periodically insert a new tuples in each table, respectively with a period of 5, 10, and 15 seconds.

Figure 3 shows the data structures adopted by this storage engine for handling tables: the *heap_open_list* is a pointer in the global data are, which references the first element of a linked list; each element in the list, in turn, references a structure that represents a table (*struct st_heap_info*), where the *struct st_heap_share* contains information about the table (e.g., the *records* field provides the number of tuples in the table) and its actual data (e.g., the *block* field). Therefore, the monitoring tool can obtain the number of tuples by

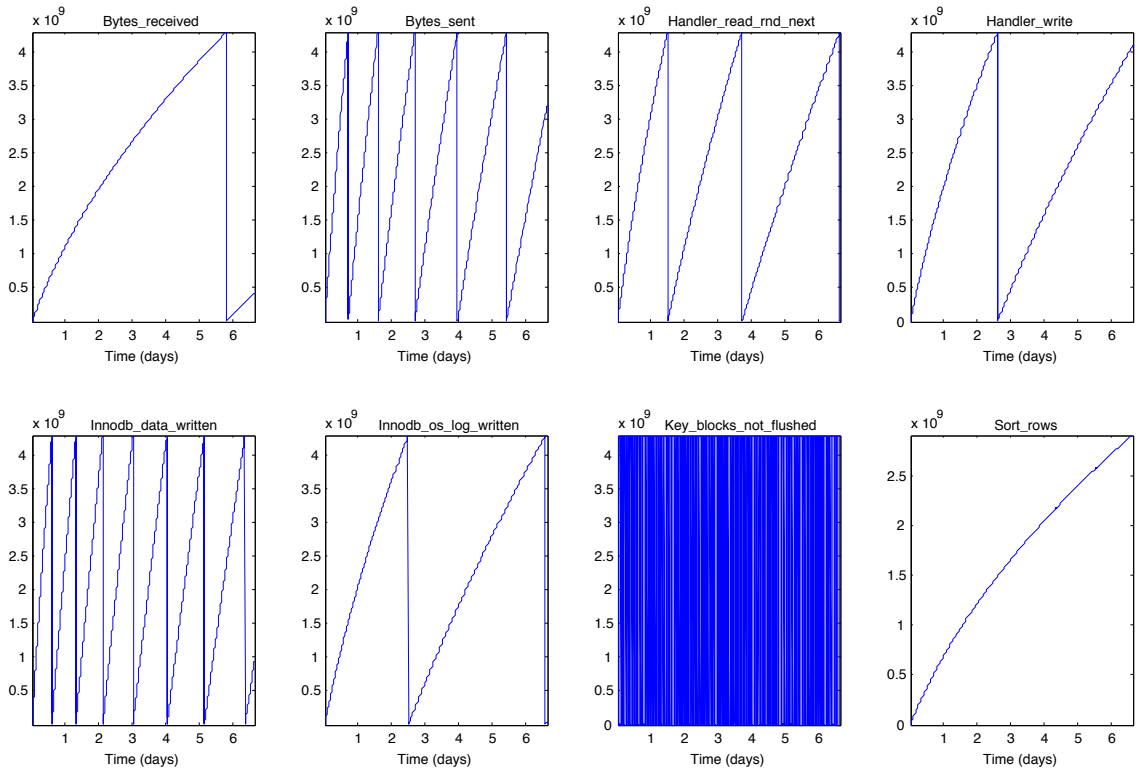

Figure 2. MySQL status variables.

following pointers, starting from the *heap_open_list* pointer in the global area. In Figure 4, we compare the actual number of inserted tuples with the value of the *records* field periodically sampled for each *struct st_heap_share* instance in the process. The samples collected by the monitoring tool matched the actual number of tuples that have been inserted in the table during the experiment, thus confirming that the approach is able to correctly sample integer values. This kind of data can be potentially used during operation in order to anticipate the occurrence of integer overflow problems.

V. Conclusion

In this paper, we analyzed some examples of numerical aging-related bugs in the MySQL DBMS, that could lead to integer overflows and cause subtle failures. Moreover, we proposed an on-line monitoring approach to identify integer overflow issues at run-time and to support software rejuvenation. Future work include the performance evaluation and optimization of the approach (e.g., in terms of computational overhead), to apply the approach to other kind of systems, and to encompass floating-point errors.

Acknowledgment

This work has been supported by the projects "Embedded Systems in Critical Domains" (POR Campania FSE 2007-2013) and "Iniziativa Software CINI-Finmeccanica".

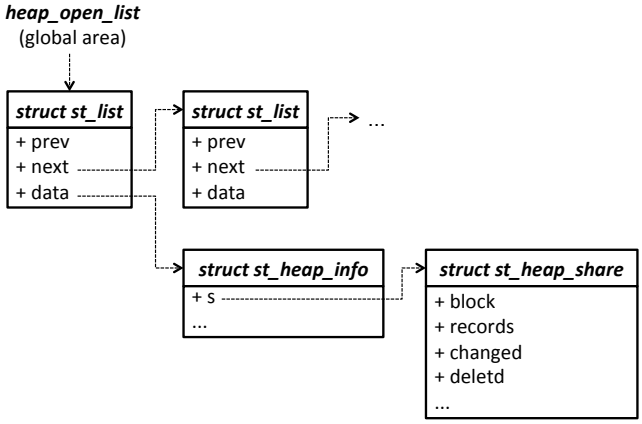

Figure 3. Data structures for handling tables in the Memory storage engine.

References

[1] J. Gray, "Why do computers stop and what can be done about it?" in *Proc. Symp. on Reliability in Distributed Software and Database Systems*, 1986.

[2] L. Bernstein, "Innovative technologies for preventing network outages," *AT&T Technical Journal*, vol. 72, no. 4, pp. 4–10, 1993.

[3] M. Grottke, R. Matias, and K. Trivedi, "The fundamentals of software aging," in *Proc. First Intl. Workshop on Software Aging and Rejuvenation*, 2008.

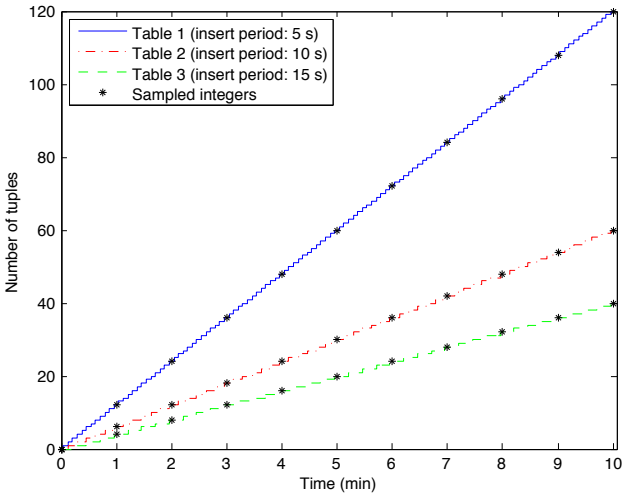

Figure 4. Actual and sampled table size.

[4] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: analysis, module and applications," in *Proc. Int'l. Symp. Fault-Tolerant Computing*, 1995, pp. 381–390.

[5] L. Bernstein and C. Kintala, "Software rejuvenation," *CrossTalk*, vol. 17, no. 8, pp. 23–26, 2004.

[6] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software Aging and Rejuvenation: Where We Are and Where We Are Going," in *Proc. Third Intl. Workshop on Software Aging and Rejuvenation*, 2011.

[7] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. Trivedi, "A methodology for detection and estimation of software aging," in *Proc. Intl. Symp. on Software Reliability Engineering*, 1998, pp. 283–292.

[8] K. Vaidyanathan and K. Trivedi, "A measurement-based model for estimation of resource exhaustion in operational software systems," in *Proc. Intl. Symp. on Software Reliability Engineering*, 1999.

[9] A. Avritzer, A. Bondi, M. Grottke, K. Trivedi, and E. Weyuker, "Performance Assurance via Software Rejuvenation: Monitoring, Statistics and Algorithms," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2006.

[10] J. Alonso, J. Torres, J. Berral, and R. Gavalda, "Adaptive online software aging prediction based on machine learning," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2010.

[11] E. Marshall, "Fatal error: how patriot overlooked a scud," *Science*, vol. 255, no. 5050, pp. 1347–1347, 1992.

[12] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," in *Proc. 2012 Intl. Conf. on Software Engineering*, 2012.

[13] W. Kahan, "How futile are mindless assessments of roundoff in floating-point computation?" 2006. [Online]. Available: http://www.cs.berkeley.edu/~wkahan/Mindless.pdf

[14] R. Dannenberg, W. Dormann, D. Keaton, R. Seacord, D. Svoboda, A. Volkovitsky, T. Wilson, and T. Plum, "As-If Infinitely Ranged Integer Model," in *Proc. Intl. Symp. on Software Reliability Engineering*, 2010.

[15] D. Brumley, D. X. Song, T. cker Chiueh, R. Johnson, and H. Lin, "RICH: Automatically Protecting Against Integer-Based Vulnerabilities," in *Proc. Network and Distributed System Security Symp.*, 2007.

[16] T. Wang, T. Wei, Z. Lin, and W. Zou, "Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution," in *Proc. Network Distributed Security Symp.*, 2009.

[17] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving Integer Security for Systems with KINT," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, 2012.

[18] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie, "Brick: A binary tool for run-time detecting and locating integer-based vulnerability," in *Proc. Intl. Conf. on Availability, Reliability and Security*, 2009.

[19] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proc. USENIX Security Symp.*, 2009.

[20] CERT, "Integral Security." [Online]. Available: http://www.cert.org/secure-coding/integralsecurity.html

[21] D. Cotroneo, R. Natella, and R. Pietrantuono, "Predicting Aging-Related Bugs using Software Complexity Metrics," *Performance Evaluation*, 2012, in press. [Online]. Available: http://dx.doi.org/10.1016/j.peva.2012.09.004

[22] M. Grottke, A. Nikora, and K. Trivedi, "An empirical investigation of fault types in space mission system software," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2010.

[23] R. Chillarege, "Understanding Bohr-Mandel Bugs through ODC Triggers and a Case Study with Empirical Estimations of Their Field Proportion," in *Proc. Third Intl. Workshop on Software Aging and Rejuvenation*, 2011.

[24] Oracle Corp., "Bug #31732: row count(*) overflows after 32-bit integer precision in spite of being bigint"." [Online]. Available: http://bugs.mysql.com/bug.php?id=31732

[25] AlienVault Inc., "Tutorial 2: Syslog data mining with attached md5sum." [Online]. Available: http://labs.alienvault.com/labs/index.php/2007/

[26] Oracle Corp., "Bug #43203: Overflow from auto incrementing causes server segv." [Online]. Available: http://bugs.mysql.com/bug.php?id=43203

[27] ——, "Bug #42698: overflow in status variable." [Online]. Available: http://bugs.mysql.com/bug.php?id=42698

[28] B. Schwartz, P. Zaitsev, V. Tkachenko, J. Zawodny, A. Lentz, and D. Balling, *High Performance MySQL*. O'Reilly Media, Inc., 2008.

[29] Oracle Corp., "MySQL Reference Manual - Server Status Variables." [Online]. Available: http://dev.mysql.com/doc/refman/5.0/en/server-status-variables.html

[30] Webyog Inc., "Percentage Of Full Table Scans." [Online]. Available: http://www.webyog.com/forums/index.php?showtopic=6129

[31] ——, "Bug - Myisam Key Cache - Cache Hit Rate." [Online]. Available: http://www.webyog.com/forums/index.php?showtopic=4492

[32] B. Jacob, P. Larson, B. Leitao, and S. da Silva, "SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems," *IBM Redbook*, 2008.

[33] Z. Lin, X. Zhang, and D. Xu, "Automatic Reverse Engineering of Data Structures from Binary Execution," in *Proc. Network and Distributed System Security Symp.*, 2010.

[34] A. Slowinska, T. Stancescu, and H. Bos, "DDE: dynamic data structure excavation," in *Proc. First ACM Asia-Pacific Workshop on Systems*, 2010.

[35] Oracle Corp., "Bug #61130: Key_blocks_not_flushed values is not correct!" [Online]. Available: http://bugs.mysql.com/bug.php?id=61130