# MEI: A Light Weight Memory Error Injection Tool for Validating Online Memory Testers

Xiaoqiang Wang, Xuguo Wang, Fangfang Zhu, Qingguo Zhou*, Rui Zhou
School of Information Science and Engineering
Lanzhou University
Lanzhou, China
E-mail: {wangxq10, wangxg13, zhuff14, zhouqg, zr}@lzu.edu.cn

***Abstract*—Lots of studies have shown that memory hardware error rates are orders of magnitude higher than previously reported. In order to fight with these memory hardware errors, many memory testing tools have been developed, especially software level online memory testers, which means these memory testers implemented in software can work with the OS (operating system) at the same time. However, validation of these online memory testers is a hard work. Since the real broken memory chips is hard to find and using a virtual machine to do this work is really complex. So we have developed a memory error injection tool – MEI (Memory Error Injection), which is implemented in software on Linux platform and easy to use. The core function of MEI is implemented in the form of a Linux kernel module. MEI also provides some user space tools for memory errors injection and manipulation. The testers to be validated need to use the read interfaces provided by MEI. We have used two exists memory testers, the modified RAMpage and COMeT+ model, to validate the effectiveness of MEI. The results of experiments have shown that these two testers could detect the emulated memory hardware errors injected by MEI effectively.**

***Keywords—Reliability; Memory; Error injection***

## I. Introduction

DRAM has been the best choice of main memory of computer in the last few decades because of its low power consumption, low cost, good performance and scalability. Since the density of DRAM is increasing continuously, the size of unit storing one bit information is decreasing and this makes DRAM more vulnerable to external factors, such as radiation, high temperature or even the dust. The result is always one or more bits in the memory chips got flipped, from 0 to 1 or otherwise, which is a big threaten to the reliability of the computer system. Such hardware errors can lead to applications and OS crash, or even worse the applications consume the data corrupted by the bit-flip errors without knowing it, which is called silent error. Silent errors could cause big problems, for example, a flipped sign bit error in a money transfer. And a recent study trying to decrease the power consummation of DRAM increases the probability of errors happening[1]. The study results from Google and Microsoft shows that memory errors are an order of magnitude more common than previously assumed and they are one of three main reasons leading the computer system crash[2][3].

A variety of methods to detect and handle memory errors are studied and developed, which are implemented in hardware level or software level. ECC implemented in hardware is the most common used tool to detect and correct errors in memory running in the background with OS[4]. It reads memory units and then do some check. ECC can only correct single-bit-flip memory errors not multi-bit-flip errors. When multi-bit-flip errors are detected, ECC reports them to the OS. Given the address of errors, OS kills the affected processes or shutdown the whole system. There are some limitations in hardware to detect all kinds of memory errors. Some memory errors can only be detected in some special situations. And not all of computers support ECC, so studies in software level memory testing is very active, such as Memtest86+[5], RAMpage[6] and COMeT+[7].

Memtest86+ is a widely used offline memory tester, which is always installed with some Linux distributions, such as Ubuntu, and can be used to detect all kinds of memory errors. However Memtest86+ runs on the bare metal directly, so the computers to be detected must reboot into Memtest86+ mode and could not do something useful. Online memory testers, such as RAMpage and COMeT+, overcome the shortcoming.

RAMpage is a hybrid memory tester running in kernel space and user space in Linux platform. It consists of a Linux kernel module and user space part. The kernel module is used to acquire the memory from OS (Linux) and map the memory to user space for test. Program running in kernel mode has the highest priority in the system thus kernel module can acquire all the possible memory for test. However the memory acquired by the kernel module is in kernel space and user space tester cannot directly use these memory, the kernel module map the acquired memory to user space for user space part program to test. It is worth mentioning that RAMpage implements as a framework and all of memory test algorithms implemented in software level can be easily integrated into it. RAMpage tries its best to test as much memory as possible in the system (some of memory could not be tested, e.g. memory containing kernel code) periodically and isolates the bad memory to prevent further use in the future. The source code of RAMpage has been released at Github[8].

* Corresponding author

COMeT+ implemented in Linux kernel constructs a tested memory pool, which contains the memory passing through the test. Both OS and applications request memory from this pool to ensure that memory to be used is safety. COMeT+ also isolates the memory including errors. Since memory errors can happen as time goes on, COMeT+ implements two timers. One to flush the out-of-date memory in the memory pool, the other to migrate the data in the old memory to the recently tested memory periodically. Memory pool needs to be replenished when its memory is low. Since the goal of COMeT+ is to implement a reliability memory allocator not to test the whole memory, so the speed of memory testing is very slow. Not like RAMpage, COMeT+ only tests the free memory to supply the memory pool.

Various software level memory testers are developed, and the validation of these testers is really an important work. The direct way to validate is to use a real broken memory chip containing permanent bit-flip errors (the error bit is always 0 or 1, no matter what value is written in it). Unfortunately , it is hard to get these hardware. Some software engineers have developed virtual machines to do this work[9]. But the steps to configure and use these tools are really complex. So some software engineers have developed the easy-to-use error injection tools.

Above these tools, the most well-known tool is mce-inject[10]. MCE (Machine Check Exception) is an exception in the Intel CPU. The exception is used to handle the hardware errors, such as high temperature in CPU, system bus errors, especially memory errors[11]. Engineers implements the MCE handler in the Linux kernel. In order to validate the handler, mce-inject was developed. It constructs a memory area in the kernel to record the information of injected errors. mce-inject injects errors into the area and then raises the MCE exception. MCE handler reads the error information from the memory area and handles the hardware error. We can see that mce-inject is not a general purpose memory injection tool. hwpoison-inject is a similar tool to mce-inject[12].

We have developed a general purpose and easy-to-use memory error injection tool called MEI aimed at validating the software level online memory testers such as RAMpage and COMeT+. It consists of a kernel module which is core part and user space part. The kernel module implements the error injection function and provides kernel programming interface. The user space part provides applications programming interfaces including error injection interface and the read interface to read content from the given memory address. MEI also implements an error injection tool in user space to inject errors and designs a simple file format to describe the injected errors. When user injects errors, the only thing need to do is to write the error code into the described file and then uses the error injection tool to inject the described file. The implemented memory testers only need to be made little modification, that is, using the read interfaces (user space interface or kernel space interface) provided by MEI, to validate its effectiveness.

Part II describes the structure and implementation details of MEI. Evaluation are present in part III. Part IV introduces some related work. Part V and Part VI concludes whole paper and discusses the future work respectively.

## II. Structure And Implementation Details

MEI is implemented in Linux platform and does not depend on any hardware architecture. It includes a kernel module and user space part. The kernel module implements the core function of MEI and the user space part is just a

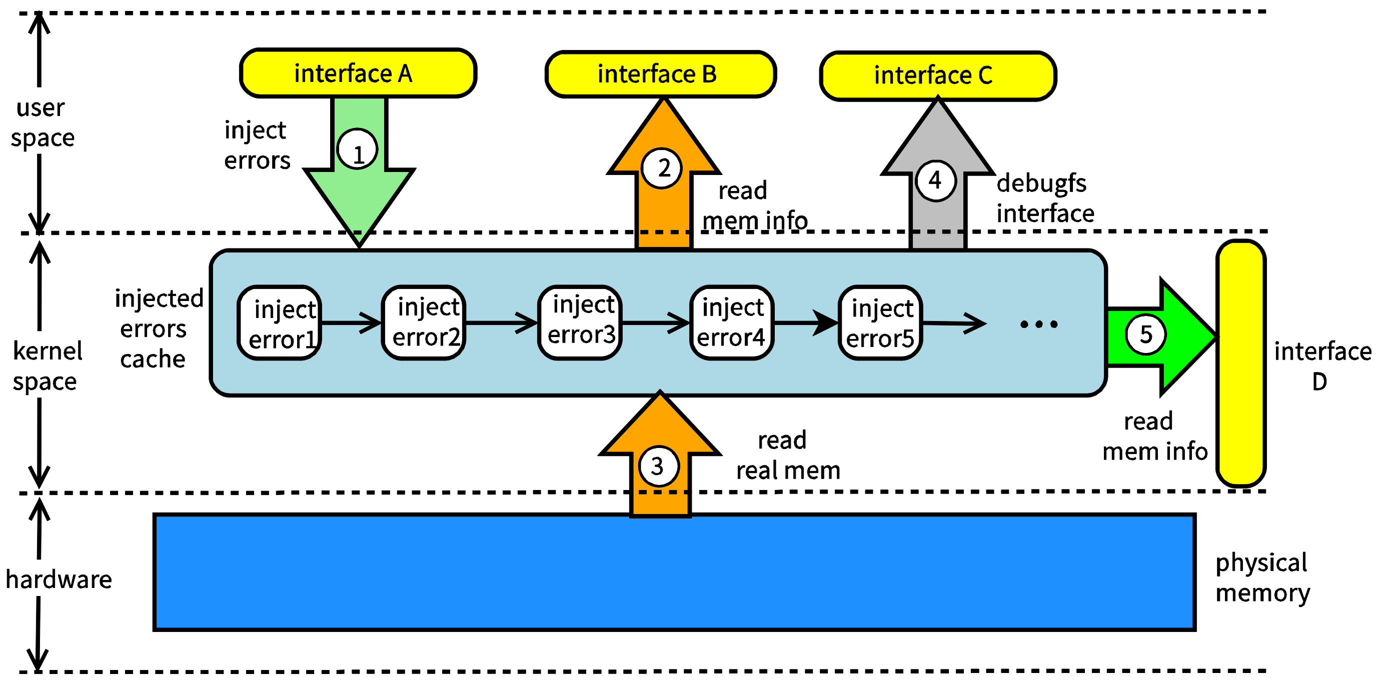


Fig. 1. MEI Structure

simple encapsulation. MEI does not modify the Linux kernel source code, all the code in kernel mode is written in the form of kernel module. So there is no need to recompile the Linux kernel. This part describes the overall structure and some implementation details of MEI.

### *A. Structure*

Fig. 1. illustrates the structure of MEI. There are three levels in the figure — user space, kernel space and hardware level. The user space part provides memory errors injection interface A (①) and user space memory read interface B (②). A user space tool has been provided by MEI to inject errors by using interface A. This tool reads the error information from the error described file, and then parses it and finally injects the error into the memory. Users of MEI can also customize their own error injection tools by using the error injection interface A.

MEI constructs a memory area in the Linux kernel to record the injected errors information like mce-inject. And all error information are organized as a list. Each element of the list represents an injected memory error. In kernel space, MEI provides a kernel programming interface D for memory testers implemented in kernel mode (e.g. COMeT+) to read the memory content. Memory testers need to use the interface B and interface D to read memory when to be validated. The whole mechanism under read interfaces is as follows:

a. Read the real memory content from the given memory address(③);

b. Search the memory area containing injected error information to find whether the injected error is in by searching the given address is in the memory area;

c. If MEI found the memory error related with the given address in the memory area, it means that a simulated memory error should occur at the given address. MEI modifies the real memory content read from the step a according to the error information found in the memory area. For example, set some bit of the real content to 0 or 1 to simulate a memory error. Then MEI returns the modified value to the read interface(②⑤).

d. If not found error information related with the given memory address, just returns the real memory content acquired at step a. This means that the given address has no memory errors injected by MEI.

The Four steps above are the core mechanism of MEI.

### *B. Usage of MEI*

There are only a few very simple steps to use MEI including installing the kernel module, editing the error injection file, and injecting the error into the system, each of which is only a command or just a little job. So the usage is far easier than the usage of virtual machine which needs really complex steps like installing the Linux OS, compiling the kernel etc.

The steps of using MEI are as follows:

a. Encode the injected errors in the error described file. The file format is very simple and is described in the section D of this part;

b. Use the user space error injection tool memerr-inject to inject the errors described by the injected file. memerr-inject takes the injected file as a command line parameter, such as:
   ./memerr-inject inject-file
   If no command line parameter is specified, memerr-inject will print error information to remind the user to specify the injected file;

c. Now the user can use interface C in fig.1 provided by MEI to show the injected errors information. The command to show the error information is like:
   cat /sys/kernel/debug/MEI/inject_errors

d. Modify the source code of memory testers to use the read interfaces (interface B or D) provided by MEI to validate the memory testers;

The quick way to remove all injected errors is to write the “clear” command into the debugfs interface C. The user space tool del-inject can remove one error from the injected cache by specifying the physical address related to the error as a command line parameter each time.

### *C. Injected Errors Cache*

The injected memory errors are recorded in a memory area organized as a list. This cache is implemented in the kernel space. Even though each read operation has to traverse the whole list (maybe only part of the list if found it) to find that whether the related error information is in the cache with low efficiency, luckily the number of elements in the list is very small, and traverse it would not consume much time. We have conducted some experiments showing that each read operation consume average 1.2us with 20 elements in the list, which is acceptable. There is no need to use more advanced data structures like tree.

In order to keep the consistent of the list, MEI use the lock mechanism provided by Linux kernel. Each read and write operation of the list must hold the lock to prevent the list from being accessed at the same time by different processors. Since the operation mode of MEI is injection of errors firstly, then validation, MEI chooses the read-write lock.

Each element of the list describes a injected memory error. The element is inject_memory_err which is defined as a struct type in C programming language. Details of inject_memory_err is described in the following section. When injected errors, MEI adds a new inject_memory_err element into the list. MEI could also remove elements from the list with different granularity. The frequency of allocation and freeing of memory for inject_memory_err is very high. The cost of using the original allocation and freeing function provided by the Linux kernel is very expensive. MEI uses a mechanism called SLAB in the Linux kernel to construct a memory cache containing lots of previously allocated inject_memory_err elements. Each allocation of inject_memory_err is from the memory cache and freeing of memory is also returned to the cache. SLAB mechanism reduces the time of allocation and freeing of memory .

## D. *Injected Errors Format*

MEI uses the struct type in C programming language to encapsulate the injected error information. The structure is defined as follows:

```
struct inject_memory_err {
        /* error injection physical memory address */
        unsigned long phy_addr;
        /* how many bits have errors */
        int err_bit_num;
        /* which bits have errors, the value is 0 or 1 */
        int bit[BYTESIZE];
        int bit_value[BYTESIZE];
        /* the list pointer */
        struct list_head lists;
};
```

Structure inject_memory_err describes the error information in the unit of byte. It is notable that the member phy_addr in the structure records the physical memory address of the injected error not virtual memory address. Each element in the list describes the error information of a byte at the physical memory address. Both OS and applications in Linux use virtual address and use MMU (Memory Management Unit) to translate the virtual memory address to physical memory address. ALL applications have their own virtual memory address space, which means that applications may have same virtual addresses. Recording virtual addresses in the structure makes no sense. Since computers have only one physical memory address space, each physical address in it is unique. The physical memory address can be used to locate the byte correctly.

When uses the read interface from user space, MEI needs to translate the virtual address passed in to physical address firstly by traverse the page table of the user space application, and then uses the physical address as a key word to search in the memory cache containing injected errors to find whether the physical address has an injected error. Page table is the table MMU used to translate the virtual address to physical address[13].Then MEI maps the physical address into kernel space (since Linux only uses virtual address, cannot use physical address directly) for kernel to read.

As for testers implemented in kernel space, MEI can use the kernel read interface which uses the virtual address from the testers directly (since MEI and testers are both in the kernel space, they have the same virtual address space). However MEI also needs to map the virtual address to physical address to search the injected memory cache.

Members "bit" and "bit_value" are used to record error information of the byte. Elements in the "bit" array having 1 value represent that at which position in the byte an error should occur. The values of bits are in the corresponding place at the "bit_value" array. The real memory content read from the memory needs to be modified according to these errors information to simulate an memory error.

The format of file described the injected errors is corresponding to the members in structure inject_memory_err. The file format is as follows:

```
phy_addr bit_num  X X X X X X X X  X X X X X X X X
```

"phy_addr" specifies the physical address to inject errors and "bit_num" is corresponding the "bit_num" member in structure inject_memory_err. Following are 16 "X"s. The value of each "X" is either 0 or 1. The first 8 "X"s are corresponding to the "bit" array and the last 8 "X"s are corresponding to the "bit_value" array in the structure. An example of injected file is as follows:

```
1048580 2  0 1 0 0 0 0 0 1  0 1 0 0 0 0 0 1
```

## E. *User Space Interface*

The memory errors injection interface A and memory read interface B have already been described before. MEI also implemented a user space interface C in debugfs file system to monitor and manipulate the injected errors cache. debugfs is a virtual file system only living in RAM in Linux, which does not have image in the disk.

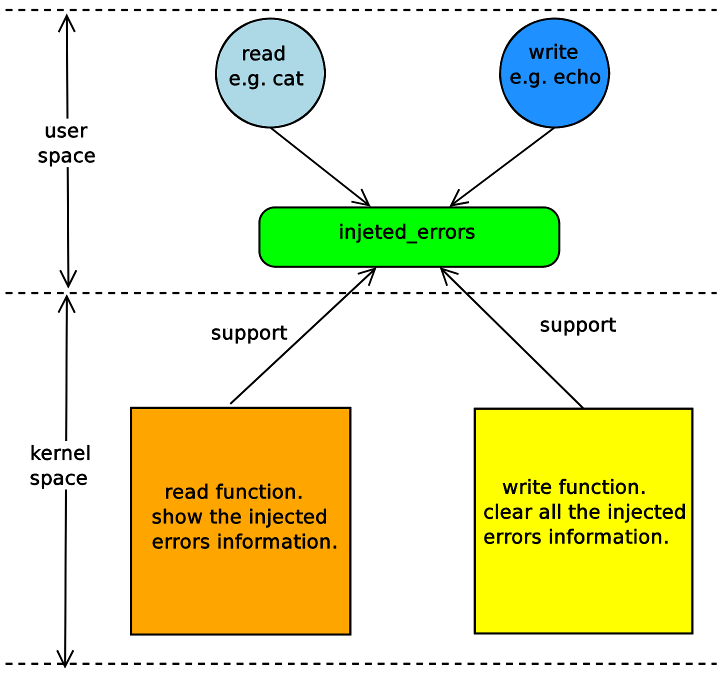
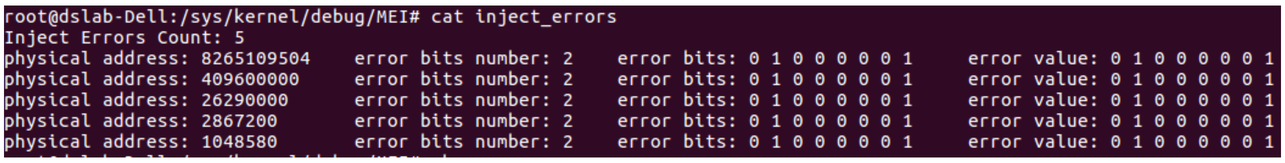

user space
read e.g. cat
write e.g. echo
injeted_errors
support
support
kernel space
read function. show the injected errors information.
write function. clear all the injected errors information.

Fig. 2. debugfs interface



```
root@dslab-Dell:/sys/kernel/debug/MEI# cat inject_errors
Inject Errors Count: 5
physical address: 8265109504    error bits number: 2    error bits: 0 1 0 0 0 0 0 1     error value: 0 1 0 0 0 0 0 1
physical address: 409600000     error bits number: 2    error bits: 0 1 0 0 0 0 0 1     error value: 0 1 0 0 0 0 0 1
physical address: 26290000      error bits number: 2    error bits: 0 1 0 0 0 0 0 1     error value: 0 1 0 0 0 0 0 1
physical address: 2867200       error bits number: 2    error bits: 0 1 0 0 0 0 0 1     error value: 0 1 0 0 0 0 0 1
physical address: 1048580       error bits number: 2    error bits: 0 1 0 0 0 0 0 1     error value: 0 1 0 0 0 0 0 1
```

Fig. 3. Sample output of read injected_errors

The interface C is implemented as a file called injected_errors. MEI implements the read and write operations of the file in kernel module. Each read and write operation in user space is mapped to kernel functions. The structure of “injected_errors” is in Fig. 2.

Users can check details of all the injected errors through the interface such as physical memory address, error bit and error value. For example, a sample output of MEI is in the Fig. 3.

The interface C also provides a function to remove all the injected errors in the cache quickly. MEI provides a user space tool to delete the specified injected errors from the cache to reach fine-grained operation.

All of these functions need the kernel module of MEI to support. The real read and write functions are implemented in the kernel space.

### F. *Kernel Space Interface*

The kernel space interface D is implemented in kernel module. Its principle has been described before. The kernel module uses a mechanism in Linux to export interface D as a global variable in kernel space. The whole kernel space can use this interface including the memory testers implemented in kernel space.

## III. EVALUATION

We use the modified RAMpage and COMeT+ model to validate the effectiveness of MEI. The reason to modifying RAMpage is that its user space part is written in Python language and the memory addresses in Python is different with C language. So we transplant the main part of user space to C language. And the experiments show that RAMpage could detect errors injected by MEI effectively.

Since the source code of COMeT+ is not released. So we have developed a model of COMeT+ to evaluate the MEI, which is really a heavy workload. And the results show that COMeT+ could detect part of the injected errors (Since COMeT+ only detects very small part of the whole memory each time and the limitation of online memory testers).

### A. *Experiment Environment*

We use a DELL laptop to conduct the experiments. The main parameters of hardware and software environment are in TABEL I.

### B. *Transplantation of RAMpage*

The user space part of RAMpage mainly implements the test algorithms and scheduling policy of the RAMpage. In order to simplify the evaluation work, we ignores the scheduling policy since it has no business with the error

TABLE I. EXPERIMENT ENVIRONMENT

| ITEM | PARAMETER |
|---|---|
| CPU | 4 cores i7-2640M@2.80GHz |
| MEMORY | 8GB |
| OS | Ubuntu 16.04 |
| Linux Kernel Version | 3.5.0 (with RAMpage patched) |

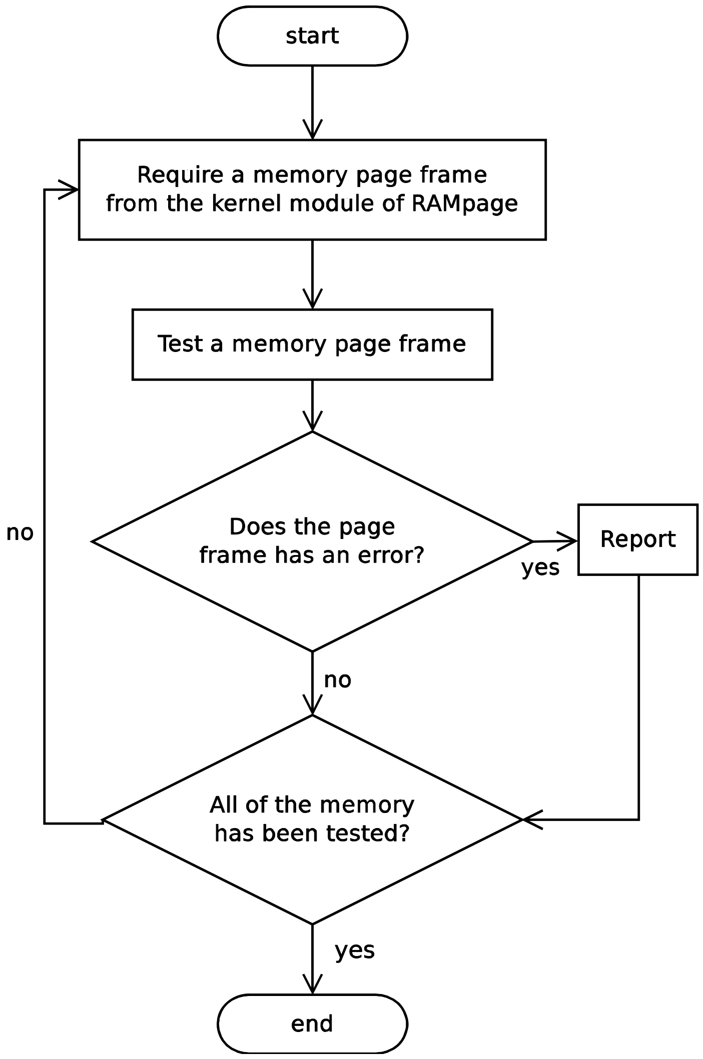

start
Require a memory page frame from the kernel module of RAMpage
Test a memory page frame
Does the page frame has an error?
yes
Report
no
no
All of the memory has been tested?
yes
end

Fig. 4. Flowchart of transplantation RAMpage

detection. The only simple policy the modified RAMpage used is testing the memory page frame by page frame. And we only implements the Linear test algorithm in RAMpage to validate the MEI. The kernel module does not need to do any modification.

The flow chart after transplantation of RAMpage is in Fig. 4. The user space part requests a page frame from the kernel module of RAMpage. If acquired, the user space part uses the Linear algorithm to test it and reports errors if found. If the whole memory has not been tested, RAMpage would continue testing.

## *C. Using RAMpage Validation*

We have transplanted the main user space part of RAMpage to C program and use the Linear test algorithm in RAMpage to evaluate MEI. Linear test algorithm writes some values to the memory and then reads memory to verify if the read value equals the written value. The test steps are as follows:

a. Use the user space memory error injection tool to inject memory errors into memory;

b. Run RAMpage to detect those injected memory errors;

We perform lots of experiments with RAMpage. Table II selected 5 groups of representative data from the whole data set. Each group of data contains the physical memory address of the injected error, type of error and the result of RAMpage test. The memory addresses are randomly selected from low physical memory address to high address.

This paper tests 5 types of errors at the same memory address in order to do a comprehensive validation. We test one bit and more-than-one bits errors at the same address and the error bits contains different 0 and 1 combination. The results show that most of the injected errors can be detected by RAMpage.

We can see that not all injected memory errors are detected by RAMpage from Table II. From the debug information printed by the kernel module of RAMpage, we can see that the undetected memory errors are stayed in the memory RAMpage cannot acquired.

RAMpage can only test the memory acquired by it and not all of the memory in the system can be acquired since some memory may contain critical data of the system, such as code or data of Linux kernel.

TABLE II. RESULT OF RAMPAGE TEST

| Physical memory addresses | Number of error bit | Values of errors bit | Result of RAMpage test |
|---|---|---|---|
| 1048580 | 1 | 0 | Detected |
| | | 1 | |
| | 2 and more | All 0 | |
| | | All 1 | |
| | | 0, 1 | |
| 2867200 | 1 | 0 | Not detected (cannot acquire memory) |
| | | 1 | |
| | 2 and more | All 0 | |
| | | All 1 | |
| | | 0, 1 | |
| 26290000 | 1 | 0 | Detected |
| | | 1 | |
| | 2 and more | All 0 | |
| | | All 1 | |
| | | 0, 1 | |
| 409600000 | 1 | 0 | Detected |
| | | 1 | |
| | 2 and more | All 0 | |
| | | All 1 | |
| | | 0, 1 | |
| 8265109504 | 1 | 0 | Not detected (cannot acquire memory) |
| | | 1 | |
| | 2 and more | All 0 | |
| | | All 1 | |
| | | 0, 1 | |

TABLE III. RESULT OF COMET+ TEST

| Physical memory addresses | Number of error bit | Values of errors bit | Result of RAMpage test |
|---|---|---|---|
| 5374276960 | 1 | 0 | Not detected |
| | | 1 | |
| | 2 and more | All 0 | |
| | | All 1 | |
| | | 0, 1 | |
| 6554266900 | 1 | 0 | Detected |
| | | 1 | |
| | 2 and more | All 0 | |
| | | All 1 | |
| | | 0, 1 | |
| 7255169509 | 1 | 0 | Detected |
| | | 1 | |
| | 2 and more | All 0 | |
| | | All 1 | |
| | | 0, 1 | |
| 8154169600 | 1 | 0 | Not detected |
| | | 1 | |
| | 2 and more | All 0 | |
| | | All 1 | |
| | | 0, 1 | |
| 8265109504 | 1 | 0 | Not detected |
| | | 1 | |
| | 2 and more | All 0 | |
| | | All 1 | |
| | | 0, 1 | |

### D. Using COMeT+ Validation

We also use the model of COMeT+ to do the same test as RAMpage. Since the source code of COMeT+ is not released, we have implemented the model of COMeT+ according to [7].

The results show that only parts of the injected errors are detected by COMeT+, since COMeT+ only tests a very small amount of memory each time to supplement the memory pool. TABLE III shows detection results of COMeT+ model in 12 hours. Only part of the injected errors are detected, but it is acceptable.

Since COMeT+ only tests free memory and does not try to acquire memory in use like RAMpage, the physical memory addresses are selected at high addresses in order to select more free memory for COMeT+ test. But we cannot guarantee that the memory at the selected addresses are always free.

## IV. Related Work

This part describes some other memory error injection technologies including APEI (ACPI Platform Error Interface) and EDAC (Error Detection And Correction)[14][15]. ACPI here stands for advanced configuration and power interface.

APEI is an important part of RAS (Reliability, Availability, Serviceability ) architecture. It defines a standard interface for hardware to handle errors between hardware and software level. APEI consists of four tables BERT (Boot Error Record Table), ERST (Error Record Serialization Table), EINJ (Error Injection Table) and HEST (Hardware Error Source Table). BERT is used to record the errors occurring at system boot. ERST is an abstract interface to record errors permanently. HEST defines the hardware errors source and error types. The EINJ table is used to inject errors in the system and raise these injected errors. EINJ is very useful for debugging and testing the RAS feature in general. For Example, it can be used to validate the MCE handler implemented in Linux kernel.

EINJ can be used to inject all kinds of hardware errors in the system. These errors are not simulated but generated by the coordination of EINJ, firmware and the hardware. There are many types of hardware errors that EINJ can inject, such as CE (correctable errors), UC (UnCorrectable errors) and fatal errors. And EINJ could inject errors in many kinds of hardware such as processor, memory and PCI-E devices. APEI can be used to validate the reliability of system. The limitation of APEI is that it needs firmware and hardware support.

EDAC is another mechanism in Linux kernel that support error injection. However main goal of EDAC is to detect and report hardware errors that occur within the computer system running under Linux[15]. EDAC can detect memory hardware errors, PCI bus errors and other errors occurring at some hardware devices. Memory errors including CE, UC are the primary errors being harvested. The error injection function of EDAC needs hardware support. Only the Intel Nehalem architecture provides hardware support for EDAC[15]. On other architectures, injected errors are only used to test the error decoding function of EDAC.

## V. Conclusion

MEI is a light weight and easy-to-use memory errors injection tool aimed at simplifying the validation of online memory testers implemented in software level. It does not depends on any hardware architecture and does not modify the OS source code. So it is general purpose and very convenient to upgrade MEI when the version of Linux kernel changed.

We use two different exists memory testers — the modified RAMpage and COMeT+ model to detect the errors injected by MEI. The experiments show that most of the injected errors can be detected by these two memory testers. Because the limitation of online memory testers, some injected errors are not detected by RAMpage and COMeT+. The results of experiments prove the effectiveness of MEI.

## VI. Discussion And Future Work

One limitation of online memory testers is that it cannot test all of the memory in the system, since some memory contains critical data. Online memory testers must acquire the memory to be test and some memory may be used by the OS or applications, which is hard to acquire. Although there are some ways to liberate the memory from applications such as offline or migration of these memory, it is still impossible to migrate kernel data from the original memory location to other locations.

Even though most of the injected errors by MEI are detected, some injected errors still cannot be detected by RAMpage and COMeT+. The reason is that we do not know if the memory injecting errors is used by kernel. So errors injected at memory used by kernel cannot be detected by memory testers.

Our future work is to investigate ways to find what the physical memory are used for at a specified address, for example, kernel data or application data. With these knowledge, MEI can inject errors at the place that memory testers can detect.

Code of MEI has been released at Github[16] and more features will be added to MEI.

## Acknowledgement

This work was supported by National Natural Science Foundation of China under Grant No. 61402210 and 60973137, Program for New Century Excellent Talents in University under Grant No. NCET-12-0250, “Strategic Priority Research Program” of the Chinese Academy of Sciences with Grant No. XDA03030100, Gansu Sci.&Tech. Program under Grant No. 1104GKCA049, 1204GKCA061 and 1304GKCA018, The Fundamental Research Funds for the Central Universities under Grant No. lzujbky-2016-140, Gansu Telecom Cuiying Research Fund under Grant No. lzudxcy-2013-4, Google Research Awards and Google Faculty Award, China.

## References

[1] S. Liu et al., “Flikker: Saving DRAM refresh-power through critical data partitioning,” in Proc. 16th Int. Conf. Archit. Support Programm. Lang. Oper. Syst. (ASPLOS), 2011, pp. 213–224.

[2] B. Schroeder et al., “DRAM errors in the wild: A large-scale field study,” in Proc. Int.Conf.Meas.Model.Comput. Syst., 2009, pp. 193–204.

[3] E. B. Nightingale et al., “Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCS,” in Proc. 6th Conf. Comput. (EuroSys’01), 2011, pp. 343–356.

[4] C. Berrou, A. Glavieux, "Near optimum error correcting coding and decoding: Turbo-codes", IEEE Trans. Commun., vol. 44, pp.1261-1271, Oct., 1996

[5] Memtest86+ [Online]. Available: http://www.memtest.org/

[6] H. Schirmeier, J. Neuhalfen, I. Korb, O. Spinczyk, and M. Engel. RAMpage: Graceful degradation management for memory errors in commodity Linux servers. In Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '11), pages 89-98, Pasadena, CA, USA, Dec. 2011. IEEE Computer Society Press.

[7] Rahman, M., Childers, B.R., Cho, S.: COMeT+: continuous online memory testing with multithreading extension. IEEE Trans. Comput. 63(7), 1668–1681 (2014)

[8] RAMpage [Online]. Available: https://github.com/schirmeier/rampage

[9] S. Potyra, V. Sieh, and M. D. Cin, “Evaluating fault-tolerant system designs using FAUmachine,” in EFTS ’07: Proceedings of the 2007 Workshop on Engineering fault tolerant systems. New York, NY, USA: ACM, 2007, p. 9.

[10] mce-inject [Online]. Available: http://linux.die.net/man/8/mce-inject

[11] Intel, Intel IA-64 Architecture Software Developer’s Manual: Vol. 3B: System Programming Guide, Part 2.

[12] hwpoison-inject[Online].Available:http://lxr.free-electrons.com/source/Documentation/vm/hwpoison.txt

[13] Bovet DP, Cesati M. Understanding the Linux Kernel. O’Reilly & Associations, Inc., 2001.

[14] Intel, White Paper: A Tour beyond BIOS Implementing the ACPI Platform Error Interface with the Unified Extensible Firmware Interface., January 2013.

[15] EDAC [Online]. Available: https://www.kernel.org/doc/Documentation/ edac.txt

[16] MEI [Online]. Available: https://github.com/wangxiaoq/MEI