

RepoMan: Monitoring and Failover platform for Windows CE drivers

Adrian Olaru & Steve Rowe
University of Illinois, Urbana-Champaign

Abstract

Mobile devices such as cell phones present unique challenges to the operating system designer. Because of their extended uptime and relatively small amount of available memory, misbehaved drivers can have a significant performance impact. Without a hard drive, many of these devices do not have page files and thus device drivers which leak memory will eventually cause the system to run out and crash or lock up. Extensive work has been done to try to locate leaks in device drivers but no work has been done to contain the problem in a dynamic manner on running phones. RepoMan is a unique approach to alleviating the problem of memory leaks in drivers. It monitors system allocations and accesses, and frees leaked memory when a driver exceeds a specified threshold.

RepoMan tracks memory access patterns thus obtaining knowledge of which memory has been untouched for the longest period of time. When it is called upon to free memory because a driver has exceeded its limit, RepoMan frees the most stale memory—that which has been untouched for the longest time—in the theory that this memory is most likely that which has been leaked. In doing so it is able to free memory for system use while allowing the driver to continue running.

To verify that RepoMan works as intended we implemented a driver which leaks memory and then observed the results. Leaked memory was indeed repossessed and the driver continued to function afterwards. Additionally we measured the performance impact on the system and found it to be acceptable. While there was a noticeable slowdown in memory allocations, these operations happen infrequently enough that no noticeable delay was found in the execution of the overall system.

1. Problem description

The pervasiveness of mobile devices using embedded operating systems makes their reliability a very important point of concern. One big problem in most operating systems is driver instability [12]. As cell phones become more capable and general-purpose, they will inherit this problem. While “full blown” operating systems have had their reliability under scrutiny in countless studies, and possible solutions have been proposed in many research papers, the operating systems used in the embedded world have not had their fair share of spotlight. This is surprising since more and more gadgets are running embedded operating systems such as Windows CE and various embedded Linux distributions. Examples of a few familiar devices running embedded operation systems are Palm Pilot / Pocket PC, phones, medical equipment and Point of Sale (POS) systems as detailed in the most recent IHL Group research [1].

A mobile device running an embedded operating system presents specific characteristics that make it susceptible to a different set of errors than a full blown operating system. Such a mobile device has limited memory, it runs continuously for a very long time even when the device is sleeping, therefore making even small memory leaks become a real threat as evidenced in Ian King’s work [2]. While the crash of a Pocket PC is merely an annoyance to the consumer, the crash of a respiratory system may have tragic consequences. Even a phone crash can be life-threatening if the call is to 911.

This project attempts to establish a common ground for preventing some of the critical failures that these devices are susceptible to. While the work presented in this paper can be applied to the majority embedded operating systems in use today, the solution we present was implemented in Windows CE.

As mentioned earlier, embedded operating systems are quite different than the ones running on regular PCs, and this is true for Windows as well. Windows CE device driver management has many differentiating points when compared with Windows XP as shown by Ian King [2]. Therefore existing device driver reliability research and solutions from the Windows XP world do not directly apply to Windows CE. To the best of our knowledge, current tools and research papers have been focused on testing device drivers in a specific environment, as a pre-requisite step before release. Once the testing is done, there exists no Windows CE platform for real-time monitoring of the device drivers at runtime for the purpose of providing a failover mechanism that can take pro-active or re-active actions related to a driver crash. Existing tools on the market do very little with regard to these kinds of reliability issues at runtime

With this project we tried to address some reliability issues that are of high importance in mobile devices, as described by Ian King [2]. We focused on runtime driver memory leaks and fatal driver crashes. With our project we took the following approach:

- 1) pro-actively prevent crashes and slow-downs due to insufficient memory caused by the accumulation of memory leaks
- 2) restart the driver in the event of a catastrophic failure.

2. Background

While Windows CE versions 5.0 and earlier were very different from desktop operating systems, Windows CE 6.0 has evolved to become similar to Windows or even Linux. In Windows CE 5.0, all drivers ran in one user mode process called device.dll. In version 6.0, many drivers moved to kernel mode, but there is also support for user-mode drivers which can each have their own process. These live in a process known as udevice.exe. The Windows CE 5.0 memory model was a shared 2GB memory space with 32MB slots—one for each of the 32 processes. Version 6.0 supports a full 4GB memory space with 2GB per process and 32,000 possible processes. This model is very similar to the model in embedded versions of Windows or Linux and thus provided a good experimental bed for technologies which can be applied to all mobile device operating systems.

Windows CE 6.0 supports virtual memory to allow for shared pages and memory mapped files. It also uses a

demand paging system for DLL loading. It does not have any support for a page file. This is not unexpected among cell phone operating systems because they do not have the same concept of disks that desktop operating systems have. This lack of a page file exacerbates the effects of drivers leaking memory. Low memory situations are first alleviated by paging out DLLs and EXEs. It cannot, however, page out data.

3. Design

Because cell phones are memory-constrained compared to desktops, we chose to tackle the problem of drivers using excessive memory. In dealing with this problem, we also provided a solution to drivers experiencing fatal crashes.

3.1. Memory Tracking and Recycling

RepoMan tracks the use of memory over time and attempts to reclaim it when a driver exceeds some threshold. Because each driver is in a separate process, memory allocations are automatically associated with each driver. The system monitors page accesses and tracks the last time each page was touched. When a driver exceeds its memory threshold, some of its memory is deallocated. Deallocations take place in an LRU fashion. Using the ideas from Rinard [13], RepoMan creates a scratch page filled with random data and points deallocated page table entries at it. In this manner if the driver ever does try to write to the leaked pointers, it will not corrupt other memory.

3.1.1. Memory Use Tracking

To track memory use, RepoMan creates a table called the *PageAge table*. This is identical in structure to a page table but rather than tracking the physical address and access bits for each page, it tracks the last time the page was accessed. Like the page table, this table consists of two levels and is indexed with the same indices. Each page in a monitored process has an entry in the table. This entry is an integer value representing the last period in which the memory was accessed for either a read or a write. The higher the integer value, the more recently the memory has been accessed.

RepoMan is designed such that memory use in any process can be tracked. A special flag—`TRACK_PROCESS_MEMORY_AGE`—is sent at `CreateProcess()` time which instructs the system to

create the PageAge table, add the process to the tracked process list, and start the memory tracking thread if it has not already been started. In our implementation, all user mode driver processes send this flag.

When a process exits, it is removed from the tracked process list. When the last tracked process thread exits, the tracking thread can be shut down to save system resources.

Memory access times are calculated by periodically marking each page with a *guard flag* and then updating the age value in the page fault handler. A MemWatcher kernel thread is created which runs at a periodicity of 10 milliseconds. When this thread runs it sets the guard bit for every writeable page in each monitored process and increments the global age counter. Read-only pages are much less likely to be leaked and often represent code pages. Windows CE 6.0 already has a mechanism to page out code pages when memory runs low.

The guard flag causes any access to the page to trigger a page fault. The OS then removes the guard bits and future accesses are made without any delay. The handler for the page fault updates the PageAge table with the current age counter value. In this way, it is possible to determine the last access time to each page and thus implement an LRU replacement policy.

Newly committed pages are initialized with the current age counter value. They are not created with the guard bit set. Rather, they will be tagged with the current time and then marked with the guard bit the next time the MemWatcher thread runs.

Because the MemWatcher thread only runs every 10ms, it does not make an appreciable impact on the performance of the system. Likewise, because the pages are only guarded once per 10ms period, the CPU use of the page fault handling is not high.

With a 10ms periodicity, a 32-bit timer will last for approximately 1 ½ years before rolling over. RepoMan does not currently handle the rollover case but if a device is expected to not reboot for a period longer than this, implementing the rollover should be straightforward.

3.1.2. Repossessing Excess Memory

When a driver exceeds its threshold, RepoMan will evict pages from that driver's memory pool. The threshold at which drivers begin losing old memory can

be set either by programmer directive in their install parameters or dynamically by the system. In our implementation we chose a programming directive. Future research will be needed to find the best ways of identifying this threshold. Early ideas include tracking an average usage and marking pages for deallocation when the usage deviates from that by some large amount.

The memory pages to be evicted are chosen with an *LRU algorithm*. Those pages which have not been accessed for the longest period of time—those having the lowest PageAge values—are repossessed. We only consider writeable memory on the process' heap for repossession. Pages that contain code or stack data are not evicted. Windows CE already has code which will page out .dll and .exe code pages. RepoMan also ignores memory that was allocated during driver initialization. Leaks during this time are unlikely and the memory may be used later during driver shutdown.

Repossession is done in two steps. First the memory is freed using the OS's VMDecommit() function. This frees the physical page and zeros out the page table entry. Next RepoMan sets the physical address in the freed page table entries to a shared scratch page. All deallocated pages point to the same physical page.

No attempt is made to make the driver process aware that memory has been freed. This allows drivers to run under RepoMan without modification. Because the memory has been leaked, the process has likely forgotten that it exists and will not attempt to access the memory again. Because freed pages are redirected to a scratch page, drivers which do attempt to access the leaked memory will not immediately fault. If the evicted memory is critical, they will fault and be restarted.

No driver will have more memory deallocated than its threshold. Thus well-behaved drivers will not lose any pages even if they have been inactive for a long period of time.

3.2. Driver Load & Restart

Because each driver is running in its own user-mode process, the system is able to restart drivers if necessary. Driver restarts take place if a) the driver crashes, or b) the system runs out of eligible pages for deallocation and needs more memory. In the latter case, RepoMan will select the driver most over

its cap in terms of memory allocation by bytes, not percentage. Applications above the drivers will be expected to handle the lost state in the driver. This would be problematic for the filesystem drivers but they run under a different driver model and so are not eligible for our protection scheme.

3.2.1. Custom Driver

In order to leverage as much flexibility as possible, we decided to create our own device driver. This enabled us to test all scenarios needed to validate our framework: device driver exhibiting memory leaks, device driver experiencing a fatal crash.

The custom driver is of stream interface category and is loaded in user mode therefore taking advantage of the latest improvements in Windows CE 6.0. The driver is registered within the operating system registry such that it is loaded at operating system startup, as any other driver is – thus ensuring that driver is handled in exactly the same manner as any other “real” driver.

A clear view of how Windows CE 6.0 handles the loading of device drivers is shown in Figure 1 on next page.

The custom driver was designed such that it misbehaves inside its write function: depending on the parameters passed to the write function, the driver allocates memory without de-allocating it or causes a crash by executing a division by 0 operation. Its behavior is controlled by the use of custom IOCTL commands.

3.2.2. Driver Reload

The driver reload mechanism takes advantage of the operating system’s DeviceManager public API, which allows the caller to find and retrieve information about an active/loaded device driver, deactivate/unload a device driver and activate/load a device driver.

3.3. Driver Harness Application

The driver behavior is controlled through an application harness that is calling the write function of the driver. Depending on the parameters passed to the application, it instructs the driver to leak memory or fatally crash.

3.4. Alternatives

We considered several alternate approaches to dealing with memory leaks in embedded system drivers.

The first approach considered was to simply monitor memory usage and restart the driver when memory use became too high. This is problematic because driver restarts are hard on the system. Applications depending on the driver will lose state. We chose to use driver restarts only as a last resort.

If possible, we want to recycle only the memory that has been leaked. This would be possible with a garbage-collector or ref-counting system. These were considered, but drivers are customarily written in C which is not easily garbage collected. Additionally, any such system would require creating a new driver model and rewriting all of the drivers.

The system we have devised should have many of the characteristics of a ref-counted system but be backwards compatible. Important pieces of memory will be accessed on a regular basis. Leaked memory will only be accessed before it is leaked. Thus, by recycling the memory that remains untouched for the longest time, we should be recycling leaked memory.

4. Implementation

Our approach was to modify Windows Embedded CE 6.0 to demonstrate that the operating system can be resilient in the face of drivers using too much memory. Windows CE 6.0 supports both user-mode and kernel-mode (in the same “process” as the kernel) drivers. We modified the user-mode driver system to support memory tracking and will attempt to reclaim unneeded memory. Each driver is in its own process so that it can be monitored and restarted if necessary. The driver restart ability is also be used to handle crashing drivers. We do not attempt to solve the problem of maintaining open handles when a driver restarts.

Modifications were in the form of kernel modifications and additional user-mode applications. Significant changes were made to the virtual memory manager in the kernel. A user-mode application also tracks memory usage and instructs the vm system to free memory from a driver when it exceeds its threshold.

An overall diagram of how the solution is designed is shown in Figure 2 on next page.

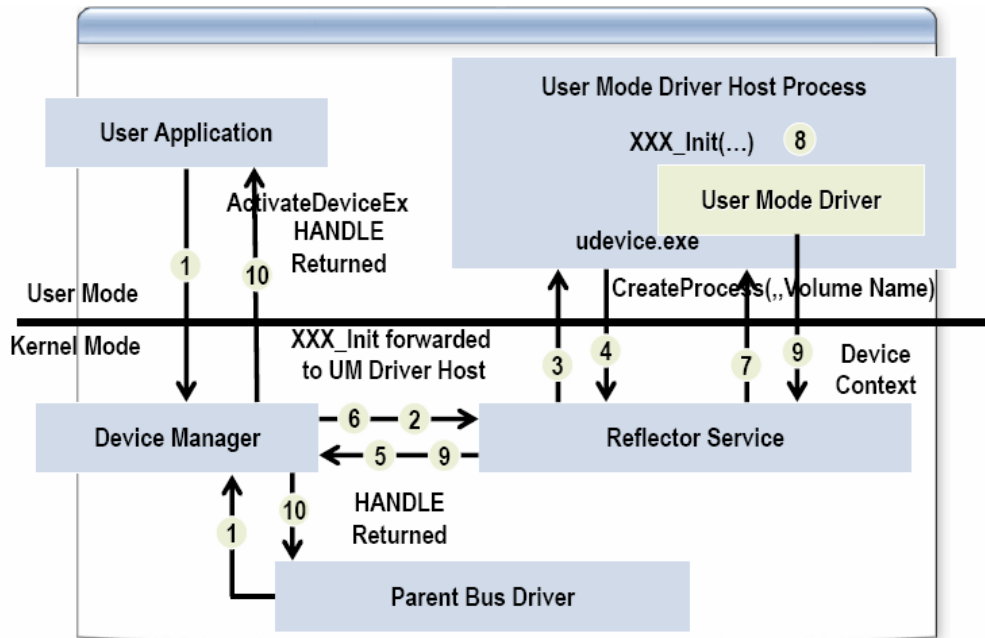


Figure 1

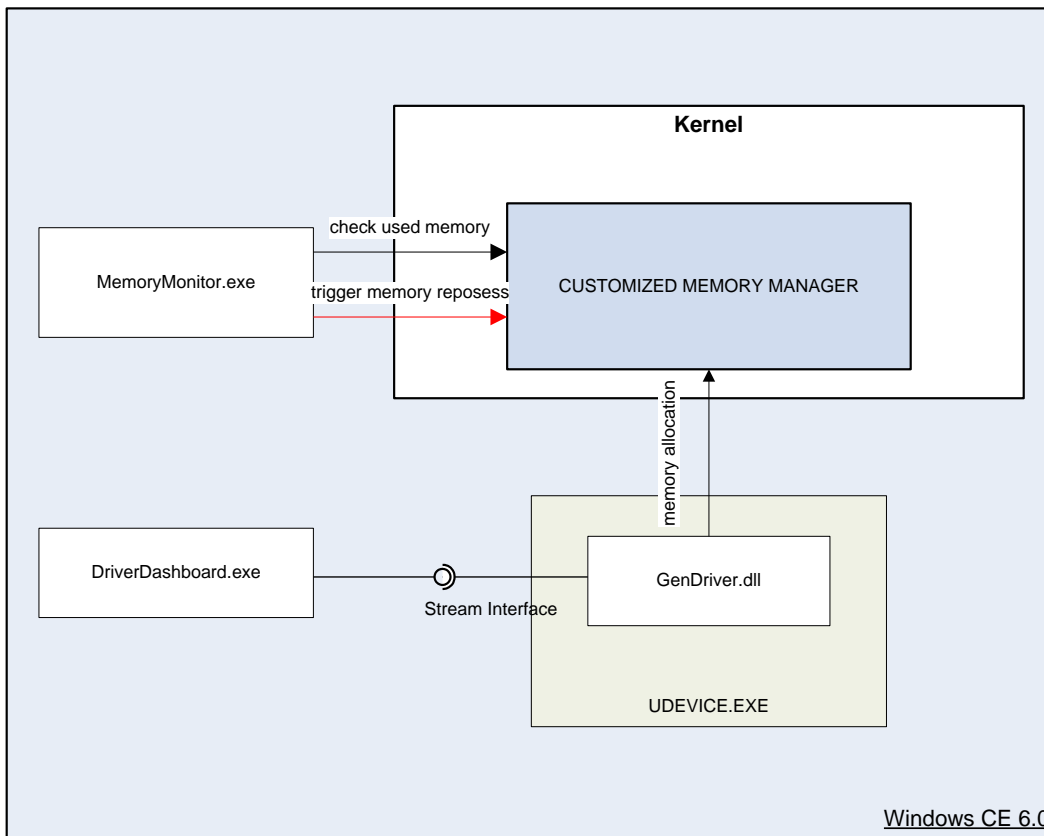


Figure 2

4.1. Memory Subsystem

The operating system kernel was modified by adding an age tracking table—the PageAge table—which shadows the real page table. An instance of this table is created for each udriver.exe process. To keep the size small, the table is made up of two levels just as the page table is. RepoMan uses the same index numbers as the page table and are thus able to reuse the OS's memory address->page table entry mapping functions.

The user mode driver loading code passes a new flag, TRACK_PROCESS_MEMORY_AGE, to the CreateProcess() function call. Inside the kernel, the AllocateProcess() function was modified to create the PageAge table when it sees this flag. PageAge creation invokes code which also starts the MemWatcher thread if it is not already running.

This kernel thread runs every 10 ms and marks all pages in the designated processes with the PAGE_GUARD flag. When guarded memory is accessed for either read or write, this flag causes the system to fault. In the page fault handler, the PageAge table value corresponding to the current page is updated to the current timestamp and the guard flag is removed. Subsequent accesses will not fault.

When RepoMan is called upon to evict memory pages, it searches through the monitored processes for the pages which have been accessed least recently (LRU) and frees them. After freeing the physical pages, it allocates a scratch physical page filled with zeros and points the page table entries for these pages at this scratch page. If the system has chosen the correct pages to evict, the scratch page will never be accessed. If it happens to have chosen a non-leaked page to be evicted, the access will fail gracefully as it writes to the scratch page.

There is no reason this system could not also be utilized for other user-mode programs but we have not done the work to test it there.

4.2. Drivers

For the purposes of this research, we chose to create our own driver that runs in the Windows CE user-mode driver framework and modified it to exhibit crashes and memory leaks. We ran it in its own process using the udriver.exe model added to Windows Embedded CE 6.0. Using this driver, we simulated out-of-memory and

crash conditions and observed the behavior to ensure that our system worked correctly.

4.2.1. Custom Device Driver

Our custom device driver is a stream interface driver named GenDriver. It implements the entire API needed by a stream driver:

- GEN_Init - the driver is initialized (loaded)
- GEN_DeInit - the driver is de-initialized (unloaded)
- GEN_Open - application calls CreateFile
- GEN_Close - application calls CloseHandle
- GEN_IOCTL - application calls DeviceIoControl
- GEN_Read - application calls ReadFile
- GEN_Write - application calls WriteFile
- GEN_Seek - application calls SetFilePointer
- GEN_PowerOn - OS resumes from suspend state
- GEN_PowerOff - OS enters suspend state

In order to have the driver loaded at OS start time, the following registry entries were created:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\GenDriver]
"Dll"="GenDriver.dll"
"Prefix"="GEN"
"UserProcGroup"=dword:3
"Flags"=dword:10
"IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

The UserProcGroup=3 says that this user mode driver will be loaded in the group with id 3, along with other user mode drivers using the same group. The use of Flags = 0x10 indicates that this device driver is to be loaded in user mode (the bit DEVFLAGS_LOAD_AS_USERPROC is set) and that it will load as a driver. Another interesting detail is the fact that the .dll could be loaded as a library (the OS uses LoadLibrary() when Flags = 0x12; the bit DEVFLAGS_LOADLIBRARY is set), in which case the memory pages can be discarded/saved to other storage, but we chose to have it loaded as a device driver instead (the OS uses LoadDriver) such that the memory pages are locked and cannot be discarded/moved. This way we could simulate the memory leak condition that we are trying to produce much faster.

The IClass value is standard for any stream driver implementing the power management functions.

4.2.2. Driver Harness Application

The driver harness application is called DriverDashboard.exe and it accepts 3 parameters:

0 – it writes to the GenDriver driver by using the standard OS functions CreateFile and WriteFile. It writes the proper information such that GenDriver crashes.

1 – it writes to the GenDriver driver by using the standard OS functions CreateFile and WriteFile. It writes the proper information such that GenDriver is experiencing a 1MB memory leak.

2 – it restarts GenDriver by using standard DeviceManager methods:

FindFirstDevice for retrieving the device handle needed to deactivate a device driver

DeactivateDevice for deactivating/unloading the device driver

ActivateDevice for activating/loading the device driver

5. Resources used for the project

We used the Windows Embedded CE 6.0 Platform Builder which includes the source code for Windows CE licensed under a Shared Source license. This includes a device emulator for Arm-based devices, which we used as our cell phone proxy.

6. Results

The system used for overall evaluation and results gathering consisted of a Core 2 Duo 1.86 GHz processor with 2 GB RAM running Windows Vista x64.

The evaluation of the solution was accomplished by running several scenarios focused on two major directions: functional evaluation and performance evaluation in the presence of memory leaks.

A third direction was for prevention in the event of a catastrophic failure, but this was not as important as the previous two, as described in sub-section 6.3 below.

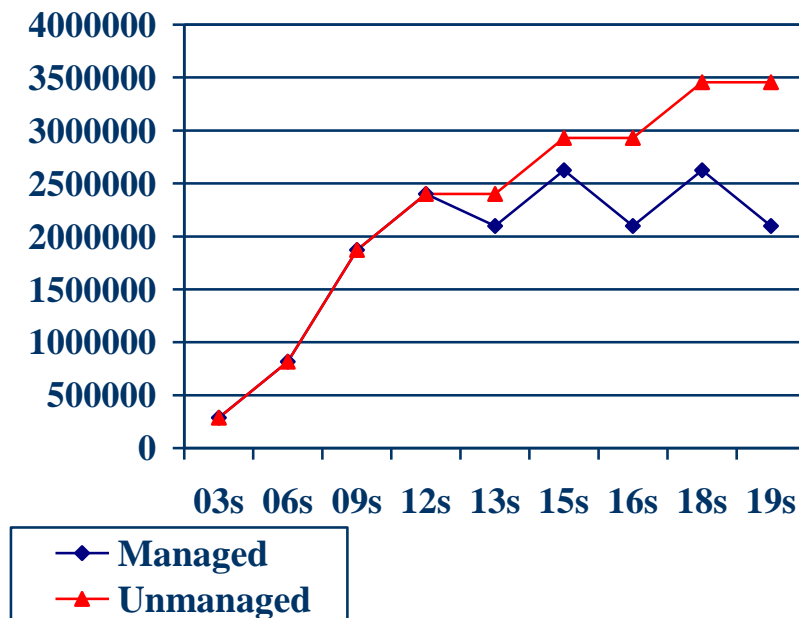
In the following sub-sections we will show the methodology used and the results that were obtained.

6.1. Functional Evaluation

With functional evaluation, the system was tested to determine whether it works as designed: given a device driver that leaks memory, the system is supposed to detect the leak and reclaim device driver's memory pages in a LRU fashion.

The test scenario was to set a 2MB threshold for the memory used by the device driver (more exactly: by the process that contains the device driver as a module). Then a user application was made to access the driver in a way that makes the driver leak 512KB memory for each access. The user application accessed the driver 6 times, with a 3 second delay in between accesses. This role was fulfilled by the DriverDashboard.exe application.

The results are shown in the graph below and they confirm that the system is functioning as designed:



- in the managed case (RepoMan framework working) the memory used by the device driver was successfully reclaimed by the OS every time it crossed the threshold.

- in the unmanaged case (no working RepoMan) the memory used by the device driver increased with every access producing a memory leak.

6.2. Performance Evaluation

The performance evaluation was designed to determine the overhead that the RepoMan framework imposes on the system.

In one test scenario, we measured the time it took Windows CE 6.0 to boot in the case of original kernel versus the RepoMan enabled kernel. The results are shown in the table below:

<i>Kernel type</i>	<i>Boot time (sec)</i>
Original	234
RepoMan (100ms)	247
RepoMan (10ms)	226

These results allowed us to conclude that there isn't any indication of a statistically significant overhead imposed on the OS boot time by the RepoMan framework.

In a second test scenario, we tried to determine whether our framework imposes any overhead on the time it takes a device driver to allocate memory. Again, we compared the memory allocation time for the original kernel versus a RepoMan enabled kernel. The results are shown in the table below:

<i>Kernel type</i>	<i>1MB (millis)</i>	<i>10MB (millis)</i>	<i>100MB (millis)</i>
Original	105	787	111
RepoMan (10ms)	169	891	170

We can see that for one kernel type, the 1MB and 100MB allocation scenarios take about the same time, while between the kernel types the allocation time increases by about 70% for the RepoMan case. The

10MB allocation time is surprising in itself, since it is about 7 times higher than the 1MB and 100MB scenarios, and this happens for both kernel types. We tested and confirmed this additional overhead on a different hardware configuration as well.

A possible explanation is that for 10MB allocation, the kernel has an additional fixed overhead (of about 700 milliseconds for the configuration tested), overhead not related to the RepoMan since it appears for both kernel types. The overhead might be related to the memory page size that the kernel is using in conjunction with the 10MB allocation. However, it is something intrinsic to Windows CE 6.0 and has no relation to RepoMan.

From the numbers above, we can conclude that, in general, RepoMan adds about 70% to the time needed to allocate memory, and the device drivers that do frequent memory allocation should be aware of this. However, we believe that the memory allocation time is insignificant when compared with other time consuming tasks that a device driver is doing (such as I/O operations), therefore the performance impact of the RepoMan framework is minimal when compared with the overall execution time of a device driver.

6.3. Protection In the Event of Catastrophic Failure

One of our original goals was to protect the OS in the event of a driver catastrophic failure, such as a division by zero exception. However, during our research we discovered that with Windows CE 6.0, the user mode drivers are running in a different process than the kernel. Even between the drivers there is a process separation safety mechanism, based on the UserProcGroup value configured in the registry for each driver: only those with the same value will share the same process. All these architectural improvements were introduced with Windows CE 6.0 (previous versions did not have them) and therefore there wasn't any real need from our framework to provide this protection.

7. Further work

While RepoMan demonstrated that memory leaks can be tracked by watching page accesses, it also exposed a semantic gap. RepoMan both tracks memory and frees it at the virtual page level. Tracking memory at this level works fine, but freeing it causes problems with the heap manager.

The heap manager operates at a logical level above the virtual memory manager. It understands only memory addresses, not physical blocks. Because of this, it is unaware that the blocks have been freed beneath it. This causes problems when it tries to compress the heap.

Because of this semantic gap, it is important that future versions of RepoMan inform the heap manager when memory is freed. If we were to continue to improve RepoMan, we would move it up the semantic stack. Each heap allocation would be given a unique virtual page address even if it shared a physical page with other allocations. Memory accesses would be tracked the same as they are now because that acts at the virtual page level.

The code which frees the memory would be moved up the stack into the heap manager. When a virtual page is found to be ready for repossession, the heap manager would free the related allocation. Freeing memory at this level will trigger physical page reclamation once physical pages are free of active allocations but will allow the heap manager to remove any references and sentinels it might have on the freed page.

Other possible improvements could be done in the area of memory monitoring, where instead of a fixed memory used threshold, more sophisticated algorithms can be used: allocation trends, historical tracking, memory used statistics.

8. Conclusions

We started by explaining why the driver reliability in general and memory leaks in particular are an important concern in mobile systems. Then we showed how a solution could be designed such that this issue can be mitigated in a pro-active way. We implemented a proof-of-concept framework named RepoMan that successfully protected a Windows CE 6.0 system against memory leaks from a generic device driver.

Based on the experience creating RepoMan and measuring its performance, we conclude that it is a plausible solution to memory leaks in drivers. It is possible to track leaked memory at the virtual page level by tracking access patterns. Further, performance analysis shows that doing so can be achieved in a way that has little impact in the overall performance of the system. RepoMan can be an effective tool to track memory leaks in user-mode drivers and potentially other user-mode processes.

9. Related Work

The solution proposed by this project fits in the category of dynamic analysis tools, tools that are working with the observed runtime behavior of an application/driver. Moreover, our solution is meant to work within the real runtime environment of an application/driver and not just a runtime environment provided by a testing tool.

We have tried to find previous studies or existing tools that are related to what this project is trying to accomplish and our findings are presented below.

Previous studies:

In the work of Michael Swift et al [6], the authors present a solution that in principle is similar from an approach perspective, to what we are trying to accomplish. However, it is designed and implemented for a full-blown Linux OS. Many of the paper's assumptions cannot be applied to an embedded OS such as Windows CE: device driver management by the OS, the API exposed by the OS to the device drivers, access to OS code.

More intrusive solutions, involving kernel and device driver code change have been proposed by Rui Shi [7], or have been focusing exclusively on memory protection in embedded processors as in Ram Kumar et al work [8].

Other solutions, such as the ones presented by David Monniaux [9] and Nicolas Blanc et al [10], are language or OS specific and rely only on static analysis, thus being unable to provide any runtime failover capability.

Existing tools:

We could find only a few tools that are related to our project's goals, but none of them is capable of doing entirely what we envision:

- *Windows CE Test Kit* [3]: is meant to test and debug a driver in a simulated runtime environment, as a pre-requisite to driver release.

- *Entrek Toolbox Windows CE Edition* [4]: can monitor memory use and API call failure of Windows CE applications with the help of CodeSnitch and ProcMan utilities. However, no driver crash protection is provided.

- *iLauncher* [5]: can track and present to the user the memory consumption of the applications running in Windows CE and it also provides a safe mode start of the entire system. This is the only tool that is tackling the “driver reliability” concern, but what it offers is just a mean to workaround a problematic driver and a way to uninstall it. By contrast, our proposal is trying to transparently provide failover capability to an existing driver through the means of error detection and driver restart (not a whole system restart).

10. References:

- [1] 2008 North American Retail POS Terminal Study - IHL Group, 2008
http://www.ihlservices.com/ihl/product_detail.cfm?page=Store%20Automation&ProductID=1
- [2] Reliability in Windows CE Device Drivers, by Ian King - Microsoft, January 2003
<http://www.windowsfordevices.com/articles/AT9847354001.html>
- [3] Windows CE Test Kit
<http://msdn2.microsoft.com/en-us/library/ms895046.aspx>
- [4] Entrek Toolbox Windows CE Edition
<http://www.entrek.com/products.htm>
- [5] iLauncher
<http://www.sbsh.net/products/ilauncher/>
- [6] Improving the Reliability of Commodity Operating Systems by Michael

Swift, Brian N. Bershad, and Henry M. Levy - December 2003 ACM SIGOPS

[7] Implementing reliable Linux device drivers in ATS by Rui Shi - October 2007 PLPV '07

[8] A system for coarse grained memory protection in tiny embedded processors, by Ram Kumar, Akhilesh Singhanian, Andrew Castner, Eddie Kohler, Mani Srivastava - June 2007 DAC '07

[9] Verification of device drivers and intelligent controllers: a case study, by David Monniaux - September 2007 EMSOFT '07

[10] Thorough static analysis of device drivers, by Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, Abdullah Ustuner - April 2006 EuroSys '06

[11] Windows Embedded CE 6.0
<http://msdn2.microsoft.com/en-us/embedded/aa714518.aspx>

[12] Improving the Reliability of Commodity Operating Systems, Michael M. Swift, Brian N. Bershad, and Henry M. Levy - 2005

[13] Enhancing Server Availability and Security Through Failure-Oblivious Computing, Martin Rinard, Cristian Cadar, Danial Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr.