Rebootless Security Patches for the Linux Kernel

Caglar Ünver
FAU Erlangen-Nürnberg
caglar.uenver@studium.fau.de

ABSTRACT
Commodity operating systems regularly release patches to fix security vulnerabilities and other bugs. Updating an operating system typically requires rebooting it, which causes temporary loss of availability of services. Delaying an update is also not suitable because of the high risk of security vulnerabilities. Updating an operating system without downtime is a desired feature for many critical tasks running on them.
In this paper, we evaluate different frameworks for patching the Linux Kernel on-the-fly. The evaluated frameworks support different ranges of possible patches and upgrades. The first framework covers live updating the Linux Kernel using virtualization technologies. In that context, a prototype called LUCOS, which updates the Linux Kernel running on XEN Virtual Machine Monitor (VMM), is explained [1]. Besides that, DynAMOS, which employs a dynamic code instrumentation technique termed adaptive function cloning [2], will also be mentioned. Another framework, called Ksplice, which operates at the object code layer and supports hot Linux security patch updates with little or no programmer involvement [3], will be examined as well.

Keywords
Dynamic software updates, dynamic instrumentation, adaptive operating system, hot updates, live update, virtualization, operating system, availability

1. INTRODUCTION
The process of applying patches usually requires rebooting a running operating system, which requires costly supervision. Rebooting can result in the loss of availability of critical tasks/processes running on that operating system. With the increasing usage of this sort of systems over time (e.g. by webhosting companies), patching the operating system without stopping and restarting it has become a critical feature [4]. Fast reboots or live migrations of processes are not a desired option, since these can result in unacceptable disruption of the services. Therefore, reboots are specially scheduled and supervised typically by system administrators.
Because of its costly supervision, system administrators tend to delay performing security updates. Since more than 90% of the attacks exploit known security vulnerabilities [11], delaying the updates exposes a great security risk and the kernel will be vulnerable to security flaws. Major Linux distributions ask their customers to install security updates more than once per month [8]. System administrators are forced to trade off the unplanned downtime against the increased vulnerability of a security flaw.

The target of updating an operating system on the fly is to minimize the planned and unplanned downtime and to prevent costly supervision of the reboots. Contemporary operating systems are large and complex and they are not designed with a live update capability in mind. Unlike the object oriented operating system kernels like K42 [9] [10], the commodity kernels do not have well defined boundaries between its modules and components. Moreover, defining the updatable unit, achieving a safe point for the update and detecting the idle or quiescent state of the units require employing additional techniques. Furthermore, some modules rarely enter to quiescent state (e.g. Network modules in webservers) or do not have a quiescence state at all (e.g. Linux process scheduler). Besides that, different rollback and versioning strategies and more complex solutions for redirecting invocations from the original unit to the newly updated unit should be applied. Finally, changing the semantic of a function or triggering a kernel module through update units can result in a dead lock situation or an inconsistent state [1].
To cover all these problems, we evaluated two proposals for updating operating systems without rebooting it. In the first approach, system virtualization is argued as a seamless capability to support live updates. A Virtual Machine Monitor (VMM) allows us to modify the state of an operating system running as a guest operating system on a Virtual Machine (VM) without rebooting or stopping it [12]. In order to exploit that feature and test the live update capability of such a system, a working prototype, named LUCOS, is evaluated [1]. On the other hand, Ksplice analyzes the original kernel and source code patch by comparing the object code rather than the source code. For creating live updates, Ksplice uses two techniques called *pre-post* differencing to generate object code for the patch and *run-pre* matching to resolve symbols. While investigating the methods proposed by LUCOS and Ksplice, DynAMOS will serve as a reference framework for comparing the characteristics of the solutions.

2. CLASSIFICATION AND OVERVIEW OF THE SOLUTIONS
Kernel updates are commonly classified in two categories:

- *Updates that modify the code:* Updates affecting code only may introduce new global variables and data structures while keeping the existing data structures unchanged.
- *Updates affecting both code and existing data:* The changed data structures can be global, single instance data or multiple instance data.

Usually changes in the data require changes in the code as well. Therefore we exclude updates changing data only. Updates affecting the code can be divided into two subcategories: Updates that change the semantic of the patched code and updates that maintain semantic equivalence with the original code [1] [2] [3].
Besides that, the live kernel updates have the following three important characteristics [2]:

- *Quiescence*. A resource becomes quiescent if no parts of the resource are in use, either by sleeping processes or partially-completed transactions. Some resources never reach quiescence state such as the scheduler of an operating system.
- *Safe update points*. A point in time where the resource is temporarily inactive (but not in quiescence state) and can be updated safely.
- *Userspace, external and internal requirements*. Modifying the behavior of a system call will change its userspace requirements. Updating the API of a kernel subsystem changes the external requirements of that subsystem. Finally, some updates can change the internal implementation of a kernel subsystem without a side effect in the rest of the kernel. Such updates change the internal requirements of that subsystem.

2.1 Using Virtualization for Live Update

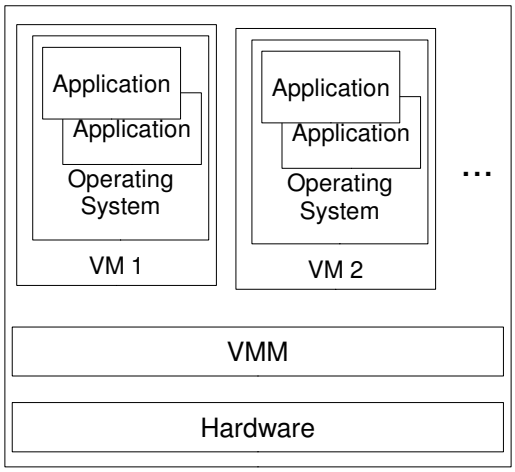

Figure 1: VMM hosting an Operating System running on two Virtual Machines.

As shown in Figure 1, the Virtual Machine Monitor (VMM), also known as a hypervisor, is in full control of system resources such as processor(s), memory and hardware devices. A VMM is able to retain selective control of processor resources, physical memory, interrupt management, and I/O. It can also intercept and emulate accesses to the memory and I/O address space executed by the VMs [12] [13]. Thus, using VMM to track and update the guest operating system without rebooting it, or without detecting the quiescence state of the updated unit can be possibly employed as a convenient technique.
As stated in [1], quiescence state of the updated unit is not a prerequisite for the LUCOS framework. Thus, live updates are possible any time. While executing the patch, the VMM takes the responsibility of maintaining the coherence of different data structure versions by calling state transfer functions.

2.2 Hot Updates at the Object Code Level

Ksplice, DynAMOS and LUCOS extension allow system administrators to apply live updates to a running Linux kernel [1] [2] [3]. Most Linux kernel security patches do not make semantic changes to data structures [3]. Unlike DynAMOS and LUCOS, Ksplice can operate at the object code layer and its design allows to generate automatic patch construction for the security patches that did not introduce semantic changes to data structures. For doing that, Ksplice requires a programmer for checking the patches whether they make any semantic changes to data structures. For the patches that do introduce semantic changes, Ksplice requires extra source code to be written.

3. DESIGN AND IMPLEMENTATION OF THE SOLUTIONS

This section explains the design and concepts of Lucos and Ksplice for updating the kernel on the fly. For applying patches, DynAMOS, Ksplice and Lucos epmloys the following common methods:

- *Stack walk-through method.* This method is mainly used for quiescence state detection. The framework iterates each kernel thread and inspects each thread's call stack for determining if it is running within the function to be patched. As illustrated in Figure 2, for doing that, a copy of the stack pointer of a process is decremented until its value reaches the bottom of the stack. An updated function is in use, if any address belonging to that function can be found on the stack below the stack pointer's value.

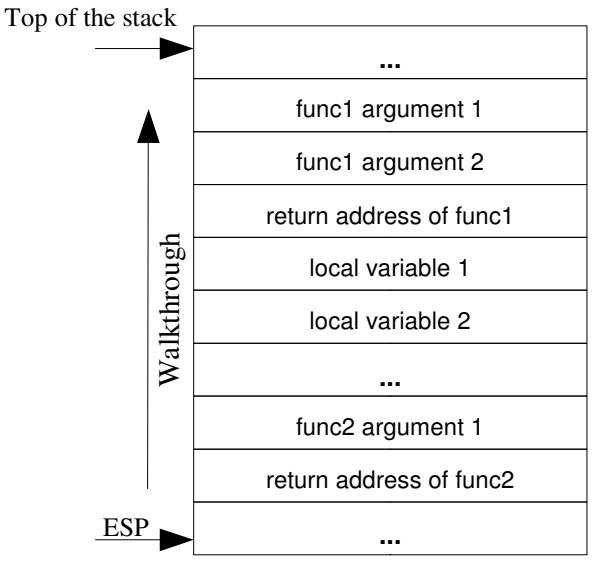

Figure 2: Stack walk-through method

- *Binary rewriting* is employed for adding a jump instruction to the first 5 or 6 bytes of the function to be patched. By applying this method, the execution is safely redirected from the original function to the newly

added function. Before doing that, it is necessary to make sure that no thread context or interrupt context is currently executing in the first 5 or 6 bytes of the function to be patched. Therefore, the framework iterates all kernel threads of the operating system, and makes sure that none of them is executing between the starting address of the function to be patched and 5 or 6 bytes beyond that address.

Since Linux supports executable code injection, the patch files are injected in the form of loadable kernel modules.

3.1 Architecture of LUCOS

As depicted in Figure 3, LUCOS consists of three components. As a user mode application, the control interface is responsible for controlling the patching process such as starting or rolling back an update. The Update Manager, which serves as a proxy between the control interface and update server, verifies the overall patch legitimacy requested by the Control Interface. After the verification, the Update Manager sends the request to the Update Server via hypercall. Hypercalls or known as VMMCalls are instructions that allow guests to explicitly make system calls to VMM [12] [13]. After the hypercall, the control is passed to VMM on the corresponding processor core via VMExit. At this point, update Server is responsible for applying the patch. Redirecting function calls, setting up necessary data structures for data consistency between different versions of data structures and invoking state transfer functions are the main functionalities of the update server. After executing the patch request, the VMM will pass the control to the VM, and the operating system will be active on that processor core again.

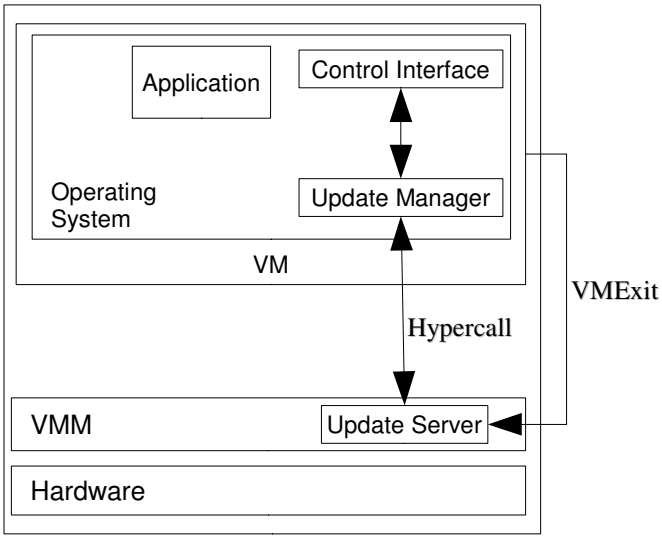

Figure 3: Architecture of LUCOS

In order to maintain the coherence of the newly defined and old data structures, for each modified data structure, a state transfer function should be provided. For the patches that have semantic changes in the code, again a state transfer function should be provided to prevent deadlock conditions and other inconsistencies.

Generally speaking, the accepted patches may include all or parts of the following units: New declaration of data structures, callback functions, patch startup and cleanup functions and state transfer functions. Among them, callback functions and state transfer functions are required for the core functionality of the framework. Three types of callback functions are supported.

- *Function callbacks*, which are called when a thread leaves a function being patched.
- *Thread callbacks*, which are called when all threads have left a function being patched.
- *Data callbacks*, which are called when all threads using a data structure have left all their functions manipulating that data structure.

3.2 Performing Live Updates using LUCOS

Lucos applies live updates in three steps.

3.2.1 Patch Initialization

As the first action, the update manager, which is a loadable kernel module, needs to gather needed information about the state of the operating system. This includes counting the number of threads or interrupt contexts executing in the code to be patched, invoking the startup function provided by the patch module and doing other initialization work before passing the control to VMM.

Applying patches affecting only the code without changing the semantic of the patch code is done by applying *binary rewriting method* for replacing the first 5 bytes of the original function with a relative jump instruction addressing the new function, as illustrated in Figure 4. For the updates affecting both the code and existing data, in case there are no threads or interrupt contexts running in the code to be patched, the Update Server does not need to monitor the states of the newly defined and the original data structures. In that case, the update can be simplified to “code only update", after transferring the state from the old data to the new data through state transfer functions. On the other hand, in case there are threads or interrupt contexts running in the code to be patched, applying changes affecting both code and data requires additional steps for ensuring the coherence of the data.

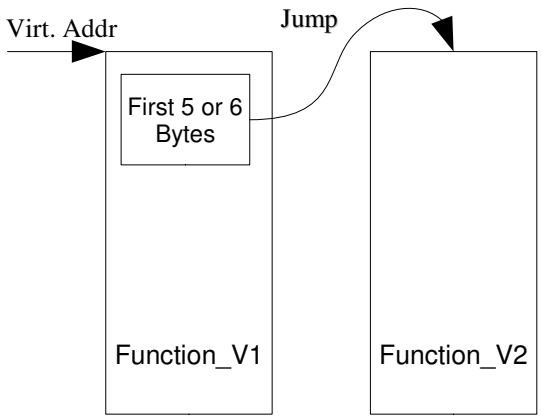

Figure 4: Live Updating Code by adding a Relative Jump

After the patch initialization step, the control will be passed to the VMM via Hypercall.

3.2.2 *Patching the System*

As the second step, the Update Server starts patching the operating system's data structures and functions. If the updated modules are not in quiescence state, both the old and new version of data structures must exist during the patching process. For ensuring the coherence of the coexisting data structures, accesses to both data structures need to be monitored and intercepted by the VMM. For doing that, the VMM will write protect the pages of the original data and newly introduced data in order to intercept the accesses to these pages. For that purpose, the x86 architecture provides two memory virtualization possibilities. As the first generation solution, shadow paging can be employed, which is keeping shadow page tables in the VMM for the VM's page tables. This software based approach is rather slow and generates lots of VMExits for every page fault exception that occurs while the guest operating system is running. The second generation solution is hardware supported memory virtualization, known as Intel's extended page tables or AMD's nested page tables, which has dramatically increased the performance by rendering shadow page tables in the VMM unnecessary [12] [13].

In case the VM attempts to write to a monitored page, which contains either version of the data structures, a VMExit will occur and the control will be passed to the VMM. At this stage, the Update Server can either find out the exact memory location and emulate the instruction that has caused the VMExit or disable the protection of the page and set a single step flag in x86. LUCOS uses the second method and compares the contents of the two versions of the monitored data structure after the single step debug exception and invokes the state transfer function to guarantee the consistency of the two versions of the data.

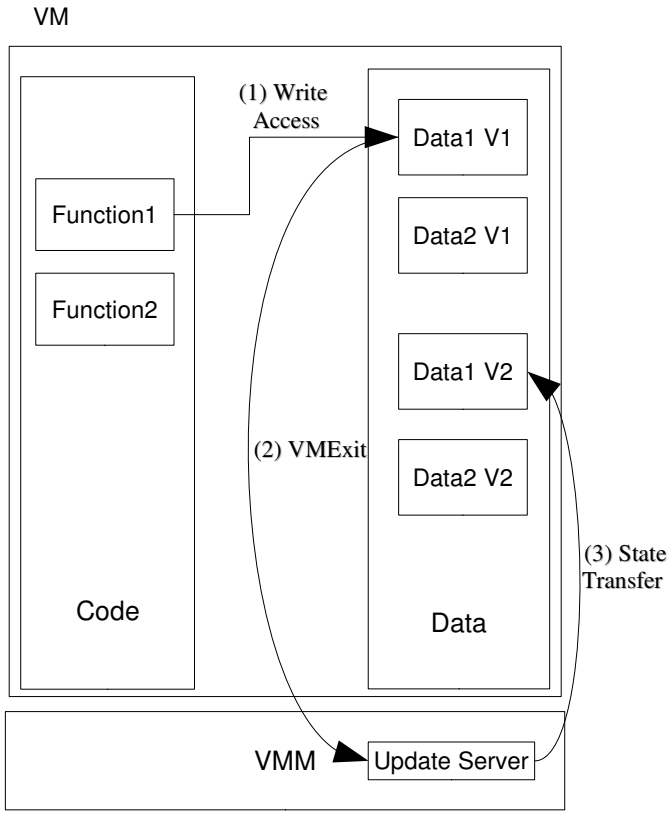

Figure 5: Live Updating Data Changes While Keeping Data Versions Consistent

As demonstrated in Figure 5, the state transfer function needs to be called in case of a write access to the protected data structures intercepted by the VMM.

3.2.3 *Terminating the Patch*

As the last step of the live update, the Update Server needs to eventually detect that the original data structures are no longer in use and disable the monitoring of the pages for the corresponding data structures in VMM. For doing that, LUCOS uses *stack walk-through method* [14]. In case the corresponding function is in use, the update manager replaces the return address of the original function with the address of a stub function, which in conclusion enables LUCOS to detect if an updated function is still in use. At the end of the patching process, functions manipulating the data structures and data structures manipulated by functions will become inactive and the thread callbacks and data callbacks will be invoked through the replaced stub functions respectively. Eventually, the update server does some patch cleanup and sets an update finished flag in order to inform the update manager.

3.3 Architecture of Ksplice

In contrast to Lucos, Ksplice does not require virtualization technologies for applying hot patches. Ksplice operates at object code level and its novelty lies in its effectiveness in creating hot patches through *pre-post* differencing and *run-pre* matching for quiescent threads[3], besides supporting DynAMOS methods for updating non-quiescence kernel threads.

Before applying a patch using Ksplice, a programmer needs to check if the patch is making semantic changes to the kernel's persistent data structures. As mentioned, Ksplice supports automatic patch construction for the updates that do not make significant changes on kernel data structures.

3.3.1 *Pre-Post Differencing*

In order to create replacement code, Ksplice first analyzes object code level differences. For doing that, Ksplice makes two kernel builds, one with the original kernel code, other with the patched kernel code. As depicted in Figure 6, the first build with the original kernel source will create object files, which is called *pre* object files and the second build will create the so called *post* object files. Unlike the source code changes, analyzing object code differences between the *pre* and *post* object files enables us to understand which functions were changed by the patch. In order to prevent unnecessary differences and location detection difficulties for finding out function and data structures in its executable text, kernel builds are done with the `-ffunction-sections` and `-fdata-sections` compiler options. Using these options forces the compiler to place each function or data item in its own section in the output file. By using these options, most of the kernel functions that have not been changed by the patch will have same *pre* and *post* object codes. For various reasons, still some of the object files may differ even if there is no affected part by the patch in the corresponding function. Ksplice safely replaces these functions too even if it is unnecessary. After the *pre-post* differencing step, Ksplice detects the changed functions and puts them into a kernel module, named as *primary module*, for loading them into the kernel.

3.3.2 Run-pre matching

At this point, the symbols referenced by the *primary module's* relocations are yet missing. Ksplice developers denoted that using the kernel's symbol table for resolving the symbols might be problematic in case of non unique or nonexistent symbol names. Relying on source code info is also not enough to extract required information from the symbol table.

Another problem arises when detecting the safety of the update systems, such as detecting locations where a non-inlined function is inlined. In that case, using source code comparisons between the old and the new versions of the function is also not sufficient since compilers tend to also inline functions that do not have the *inline* keyword.

As illustrated in Figure 6, in order to solve these issues, Ksplice compares the running kernel code with the compiled original kernel code or so called *pre* code. This process is called *run-pre* matching. As a result of that comparison, the inlined functions can be detected and, moreover, the symbols can be resolved.

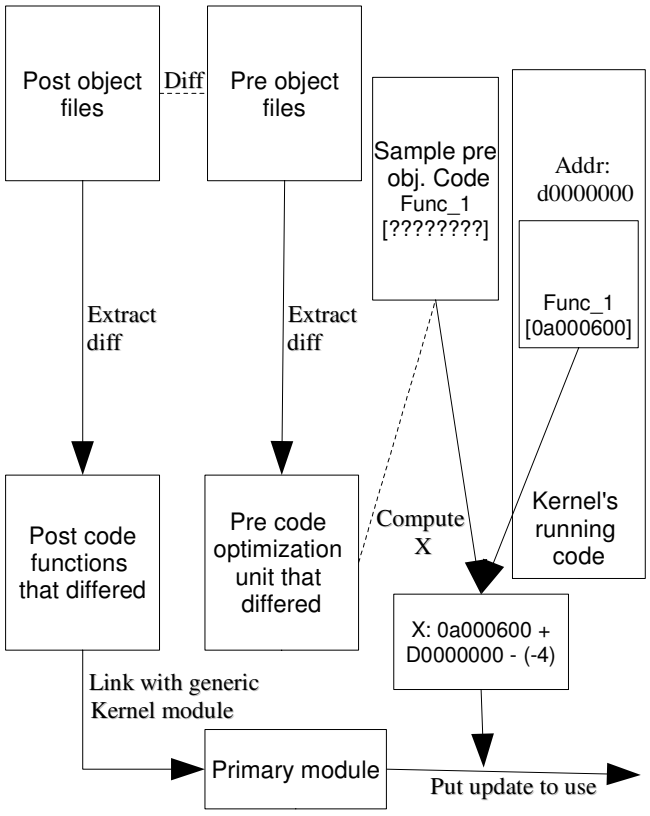

Figure 6: Illustration of pre-post differencing (left) and run-pre matching (right)

3.4 Performing live updates using Ksplice

As stated before, Ksplice needs to detect a safe time to apply live updates. For that purpose, Ksplice calls the `stop_machine` function provided by Linux in order to stop all of the CPUs and run the desired function on a single CPU. In that particular time, Ksplice applies *stack walk-through method* and if every function being replaced is in quiescent state, the safety condition is met and Ksplice can proceed with applying the patch. In that case, the replacement code will be loaded to the kernel and jump instructions will be inserted into the old functions by applying *binary rewriting*.

Calling `stop_machine`, checking safety conditions and updating the kernel takes about 0.7 milliseconds to execute. If the safety conditions are not met, after a short time, Ksplice tries to call `stop_machine` again. After multiple failures to reach a safe state, Ksplice aborts the update and reports an error to the user.

Ksplice also supports custom code for modifying data structures during the function replacement operation, as required by the patches making semantic changes to data.

4. EVALUATION

4.1 Performance

In [1] it is stated that the Xen-Linux of LUCOS incurs a less than 1% performance loss in comparison to Xen-Linux. DynAMOS reports an overhead on the kernel functions in the range of 1-8% after applying its updates [2]. Ksplice claims a minimal performance impact for applying patches [3]. The execution of the whole system will be delayed for 0.7 milliseconds whenever Ksplice calls the `stop_machine` function for quiescence detection, which likely violates strict timing constraints of the real time systems. No information is given about the performance impacts of executing LUCOS patches on real time operating systems running in the guest mode.

4.2 Patch Rollback

Any update to the operating system should be transactional to avoid corrupting the whole system. Both LUCOS and Ksplice do support rolling back the patches. Rollbacks are patches using the original code left after the committed patches. For restoring the code changes, the first 5 bytes of the functions should be restored. In case of data changes, the state transfer functions are reused for maintaining the consistency.

4.3 Known Issues

The stack inspection method used by LUCOS, DynAMOS and Ksplice can cause some performance problems especially when the system is under heavy workload.

For supporting rollback patches, keeping the original code in memory can lead to some performance and resource overhead.

LUCOS, DynAMOS and Ksplice do not offer any particular method for verifying the correctness of the patches. The verification of the patches is left to the programmers and testers. It is emphasized in [1] that the patch cannot be rolled back if the kernel collapses when executing a buggy patched function.

5. DISCUSSION AND RELATED WORK

Dynamic software updating techniques cannot be applied to the existing operating systems. Therefore, new techniques have been implemented.

Ksplice can automatically update kernel modules if they are in quiescence state. On the other hand, LUCOS and DynAMOS are not limited to the modules having quiescent state [1] [2] [3]. K42 [9] [10] has short living and non blocking kernel threads.

Therefore detecting quiescence is easier. Unlike K42, quiescent detection of LUCOS, DynAMOS and Ksplice requires special techniques such usage counters of the functions and stack inspection of the processes.

DynAMOS identifies non-quiescent functions by their utilization of synchronization primitives. For enabling synchronized updates of these functions, DynAMOS employs an algorithm called adaptive function cloning [2]. This algorithm is applied in three phases, which includes creating different versions of the original function, implementing usage counters for these versions and using global flags for determining which version to be used. For managing execution flow diversions between different versions of the functions, DynAMOS installs an execution flow redirection handler [2]. LUCOS realizes the functionality of that approach through its state transfer functions, whereas it classifies that operation as updates affecting code and data. Since the virtualization technologies allow LUCOS to intercept and alter the function calls accessing data structures via VMExits, the required logic for function redirection will be triggered in the host mode. Therefore, unlike DynAMOS, creating global flags or execution flow redirection handlers in guest mode is not needed.

For redirecting function calls, *binary rewriting* is used by LUCOS, DynAMOS and Ksplice to patch the original function with a jump instruction [1] [2] [3]. DynAMOS redirects the function calls to an execution flow redirection handler, whereas LUCOS and Ksplice link the calls directly to the new version of the function.

For data-type updates, whereas LUCOS exploits virtualization technologies for monitoring accesses to data structures and maintaining coherence of data structures, DynAMOS uses their newly developed technique, which uses shadow data structures. Unlike LUCOS, DynAMOS does not transfer the state of the old data structures to the newly defined ones. Instead, for each new data type, DynAMOS maintains a shadow data structure. DynAMOS creates a shadow variable upon creation of variables of the new data type, which contains only the new fields of the new data types [2]. A hash table is used for mapping the memory address of the variable into its shadow. Ksplice is also able to utilize shadow data structures described in [2].

Dynamic updates in K42 are limited to K42 object classes [9] [10]. Low level code cannot be updated dynamically. On the other hand, LUCOS and DynAMOS can update the low level code and exception handling code.

LUCOS, DynAMOS and K42 do not support automatic patch construction [1] [2] [3] [9] [10]. Updates are prepared by the programmer at the source code layer. Constructing updates manually requires analyzing the patch and writing update code for that patch, which is a quite complex and error prone process.

According to the evaluation reported in [3], 56 of 64 x86-32 Kernel security patches released from May 2005 to May 2008 can be applied by Ksplice without a programmer involvement. On the other hand, Ksplice requires a programmer check on the updates for determining if they make semantic changes to data structures. Performing this check takes a few seconds to a few minutes for most security patches. Ksplice is available as a part of Oracle Linux [7].

New developments in virtualization technologies allow a hypervisor to selectively monitor system resources with decreased numbers of VMExits and without suffering from increased hypervisor overhead and performance losses. Therefore, patching real-time Linux systems using virtualization technologies may be a convenient method for mission critical systems.

6. CONCLUSION

We have evaluated different frameworks for applying updates to a running kernel without rebooting it.

LUCOS is a prototype for updating a running Linux kernel without disrupting its services and without causing extra performance overhead on the VM by exploiting the virtualization technologies [1].

Ksplice has proven its practicality for creating patches with minimum programmer involvement. It also introduces two new object code analyzing techniques, *pre-post* differencing and *run-pre* matching. Ksplice is not limited to security patches. Another usage of Ksplice is to support Linux users for applying specific patches that are developed for the local issues and bugs on their servers.

7. REFERENCES

[1] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang and Pen-Chung Yew. Live Updating Operating Systems Using Virtualization. *VEE '06 Proceedings of the 2nd international conference on Virtual execution environments.*

[2] Kristis Makris and Kyung Dong Ryu. Dynamic and Adaptive Updates of NonQuiescent Subsystems in Commodity Operating System Kernels. *EuroSys '07 Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* 2007.

[3] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. *EuroSys '09 Proceedings of the 4th ACM European conference on Computer systems.*

[4] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*. 10(2):53–65, 1993.

[5] Intel Corporation. Intel vanderpool technology for IA-32 processors (VT-x) preliminary specification. http://www.intel.com/technology/computing/vptech/.

[6] Jeff Arnold and M. Frans Kaashoek. Ksplice evaluation full data: kernel versions, commit ids, and new code, 2008. http://www.ksplice.com/cve-evaluation-2008.

[7] Oracle Corporation. Ksplice: Zero Downtime Updates for Oracle Linux. http://www.oracle.com/us/technologies/linux/ksplice-datasheet-487388.pdf

[8] Nexcess Technologies Inc. . Nexcess Adopts Ksplice Uptrack “Rebootless” Technology. http://blog.nexcess.net/2010/11/30/nexcess-adopts-ksplice-uptrack-rebootless-technology/

[9] Andrew Baumann and Gernot Heiser. Providing dynamic update in an operating system. *ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference.*

[10] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger and Gernot Heiser. Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. *ATC'07 2007*

USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference.

[11] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 ACM SIGCOMM Conference*, pages 193–204, 2004.

[12] Intel Corporation. Hardware-assisted virtualization technology. http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/hardware-assist-virtualization-technology.html

[13] Advanced Micro Devices Corporation. AMD Virtualization. http://www.amd.com/en-us/solutions/servers/virtualization

[14] Paul Burstein, Gautam Altekar, Ilya Bagrak and Andrew Schultz. OPUS: Online Patches and Updates for Security. In *Proceedings of 14th USENIX Security Symposium*, Baltimore, MD USA, 2005.