Rebootless Security Patches for the Linux Kernel

Caglar Ünver
30.05.2014

Motivation

Why do we care about updates on the fly

- More than 90% of the attacks exploit known security vulnerabilities
- Important bugfixes and security updates roughly every month
- Delaying the updates: a great security risk
- Reboots: Service outage, administrator supervision needed (sysadmins working on weekends)

Challenges

- Commodity kernels do not have well defined boundaries between their modules and components
- Some modules are always busy

Outline

1. Classification of Kernel Updates
   - Updating Code Only
   - Updating Code and Existing Data

2. DynAMOS - The Basic Approach
   - Quiescence Detection
   - Binary Rewriting
   - Redirection Table

3. LUCOS - Using Virtualization for Live Updates
   - State Transfer

4. Ksplice - Hot Updates at Object Code Level
   - Pre-post Differencing and Run-pre Matching

5. Conclusion and Discussion

1. Classification of Kernel Updates
   - Updating Code Only
   - Updating Code and Existing Data

2. DynAMOS - The Basic Approach
   - Quiescence Detection
   - Binary Rewriting
   - Redirection Table

3. LUCOS - Using Virtualization for Live Updates
   - State Transfer

4. Ksplice - Hot Updates at Object Code Level
   - Pre-post Differencing and Run-pre Matching

5. Conclusion and Discussion

Classification of Kernel Updates (1)

Updates that modify the code only

- Keeps the existing data structures unchanged
- May introduce new data structures, global variables
- Easy to patch, if there are no semantic changes

Classification of Kernel Updates (2)

Updates that modify the code and existing data

- Existing data structures will be changed
- State transfer from the old to the new data needed
- What if the semantic of the patched code is changed?

Classification of Kernel Updates (3)

Changing the semantic of the code

```
void foo()
{
...
 do
 {
    ...
    unlock(semaphore);
    ...
    lock(semaphore);
    ...
 } while(someVar)
return;
}
```

→

```
void foo()
{
...
 do
 {
    ...
    lock(semaphore);
    ...
    unlock(semaphore);
    ...
 } while(someVar)
return;
}
```

1. Classification of Kernel Updates
- Updating Code Only
- Updating Code and Existing Data

2. DynAMOS - The Basic Approach
- Quiescence Detection
- Binary Rewriting
- Redirection Table

3. LUCOS - Using Virtualization for Live Updates
- State Transfer

4. Ksplice - Hot Updates at Object Code Level
- Pre-post Differencing and Run-pre Matching

5. Conclusion and Discussion

DynAMOS (1)

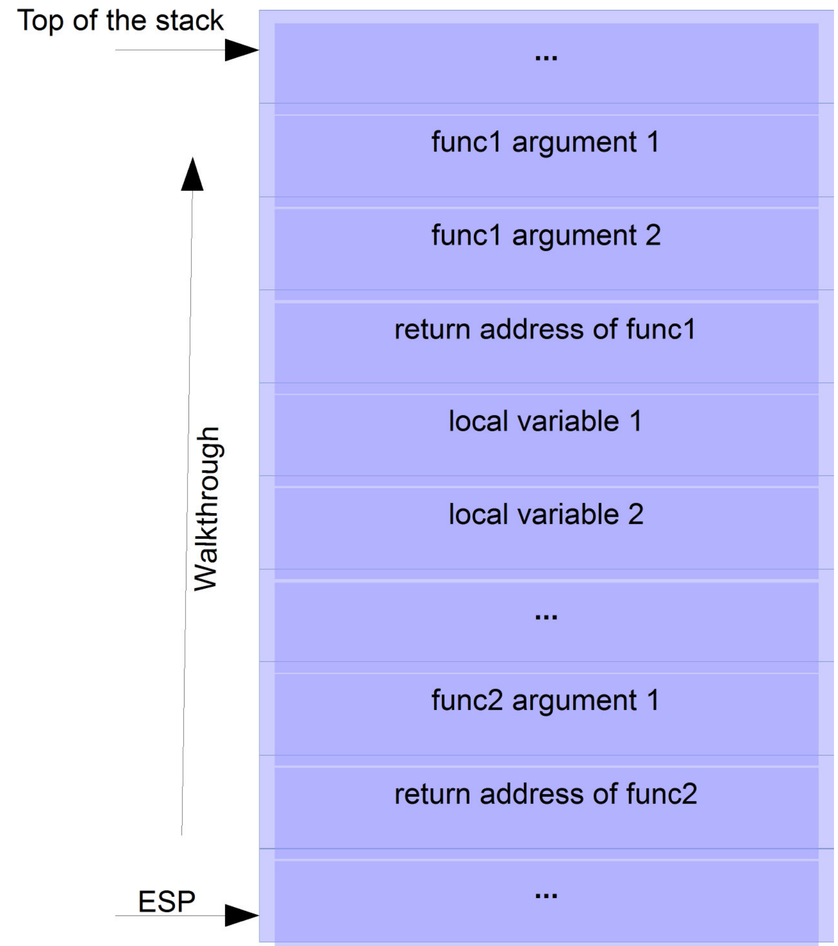
Quiescence

- If no parts of the resource are in use, either by sleeping processes or partially-completed transactions
- No function can be idle on the stack.
- Updating modules in quiescence state is easier
- Some processes never reach quiescence state (e.g. Process scheduler)

DynAMOS (2)

Quiescence Detection

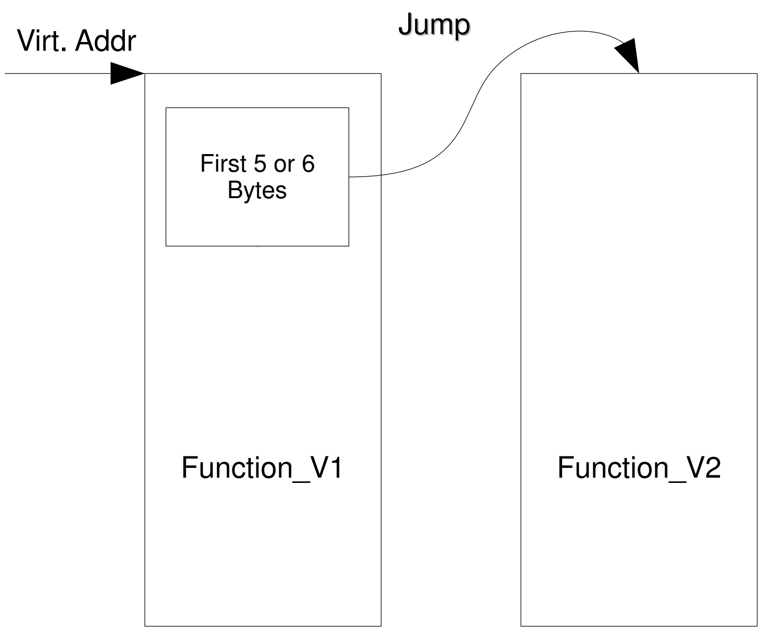
- Function Usage Counters (but not sufficient e.g. do_exit)
- Stack-walkthrough Method (Has side effects)



DynAMOS (3)

Binary Rewriting

- Adds jump instruction at the top of the function
- Make sure that no thread context or interrupt context is executing in the first 5 or 6 bytes of the function



DynAMOS (4)

State Tansfer is needed:

- Existing data structures changed
- Semantic of the function changed
- Updated unit not in quiescence state

1. Classification of Kernel Updates
   - Updating Code Only
   - Updating Code and Existing Data

2. DynAMOS - The Basic Approach
   - Quiescence Detection
   - Binary Rewriting
   - Redirection Table

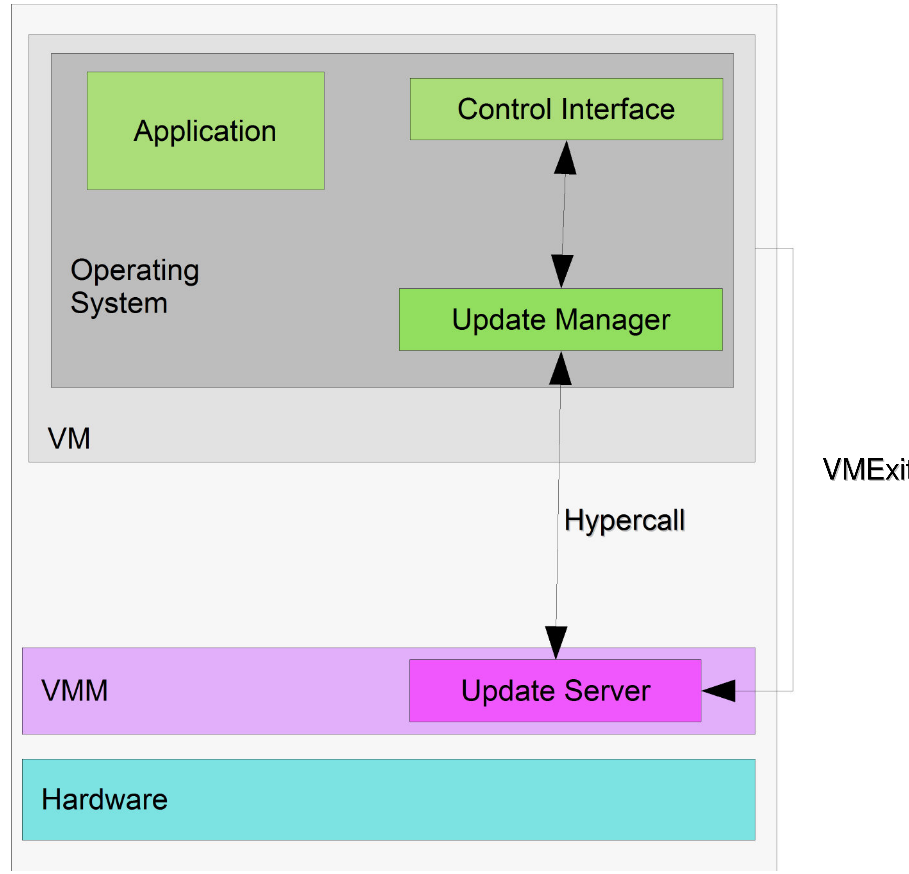
3. LUCOS - Using Virtualization for Live Updates
   - State Transfer

4. Ksplice - Hot Updates at Object Code Level
   - Pre-post Differencing and Run-pre Matching

5. Conclusion and Discussion

LUCOS (1)

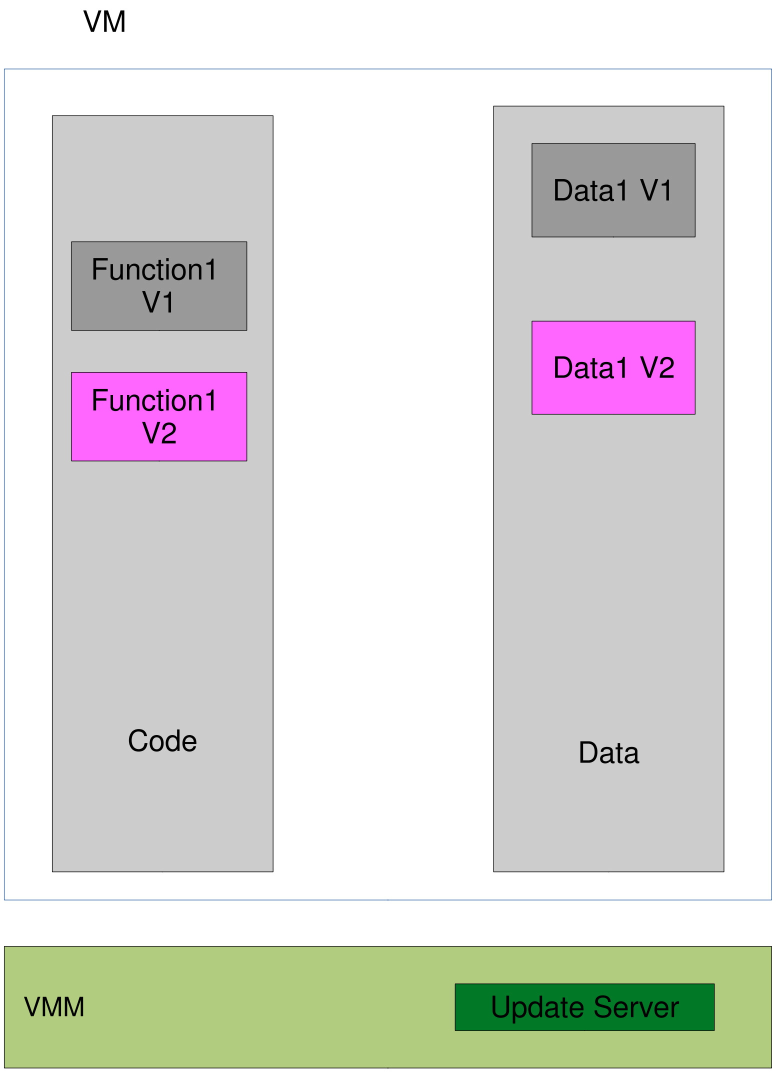
- Virtual Machine Monitor(VMM) controls system resources
- VMM intercepts and emulates memory and I/O accesses



LUCOS (2)

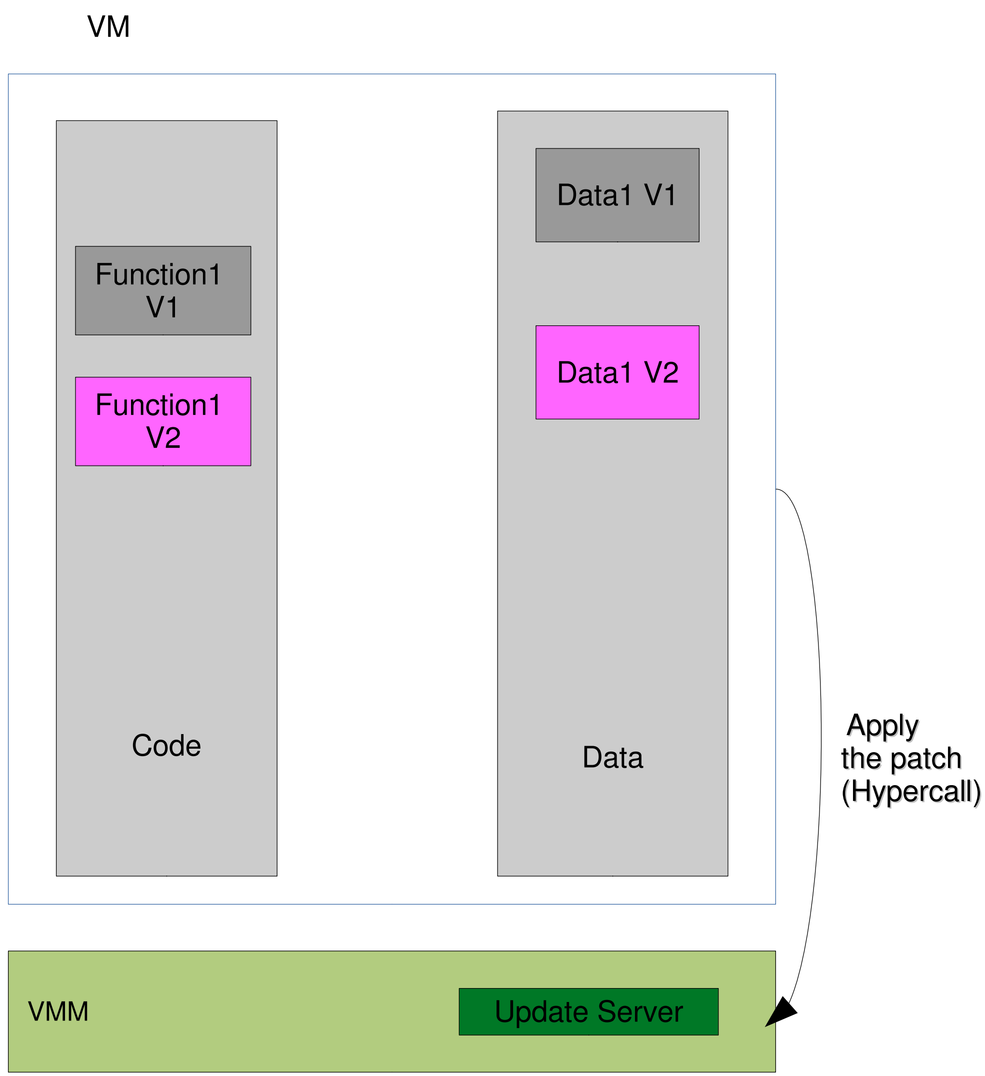
- Quiescence state is not a prerequisite
- Manual patch creation
- Patch files: Code + data structures as loadable kernel modules

LUCOS (3)



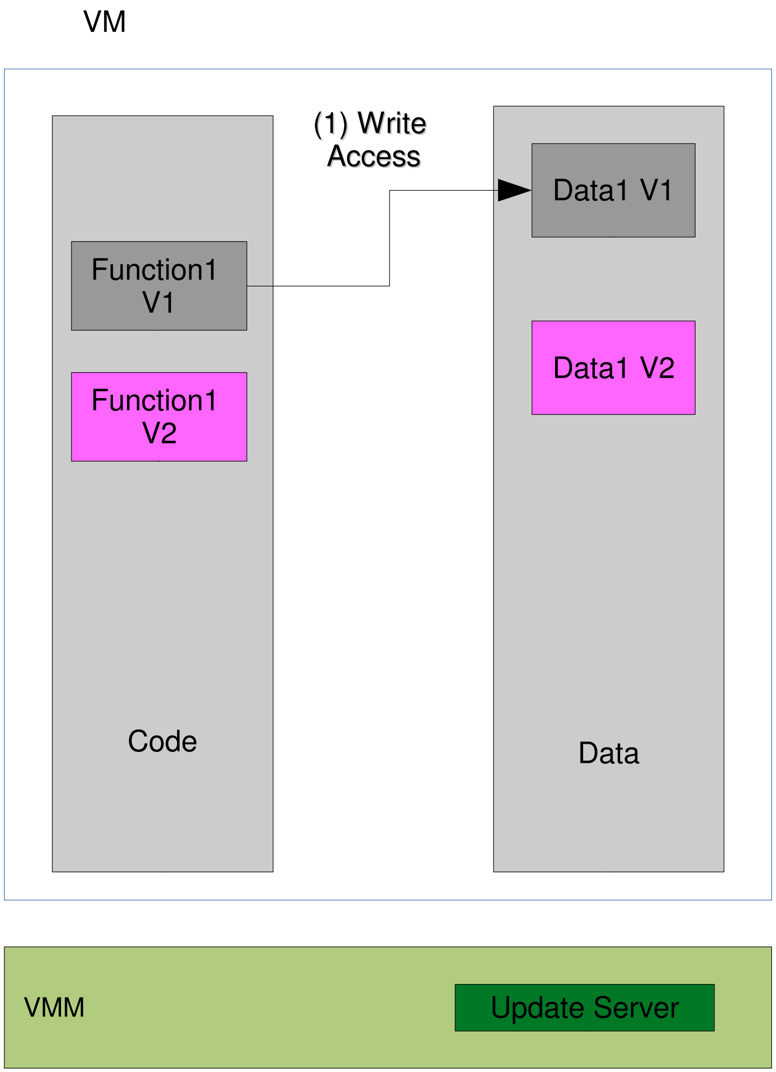
- Update Manager loads kernel modules for the patched function(s) and data structure(s)

LUCOS (4)



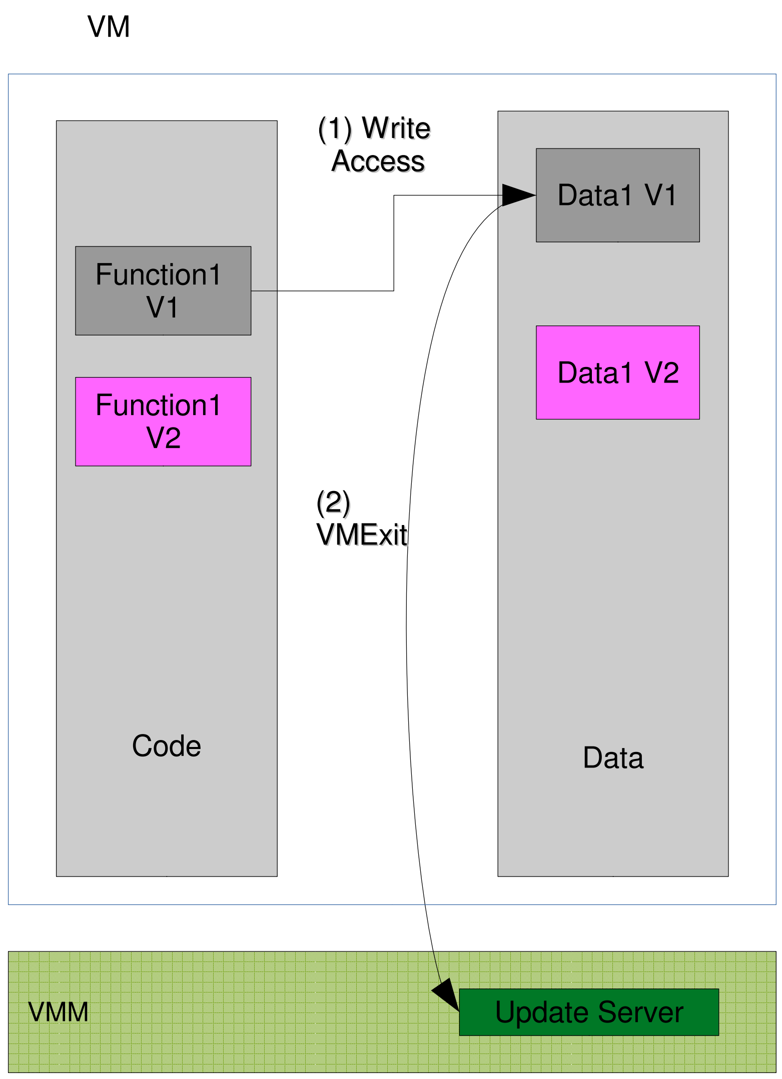
- Update Manager iterates all kernel threads and makes sure that none of them is executing in the first 5 bytes of the function
- Update Manager inspects kernel call stacks for counting threads executing in the patch code
- Control is passed to Update Server via hypercall
- Update Server applies binary rewriting for inserting jump and for replacing return address of the function

LUCOS (5)



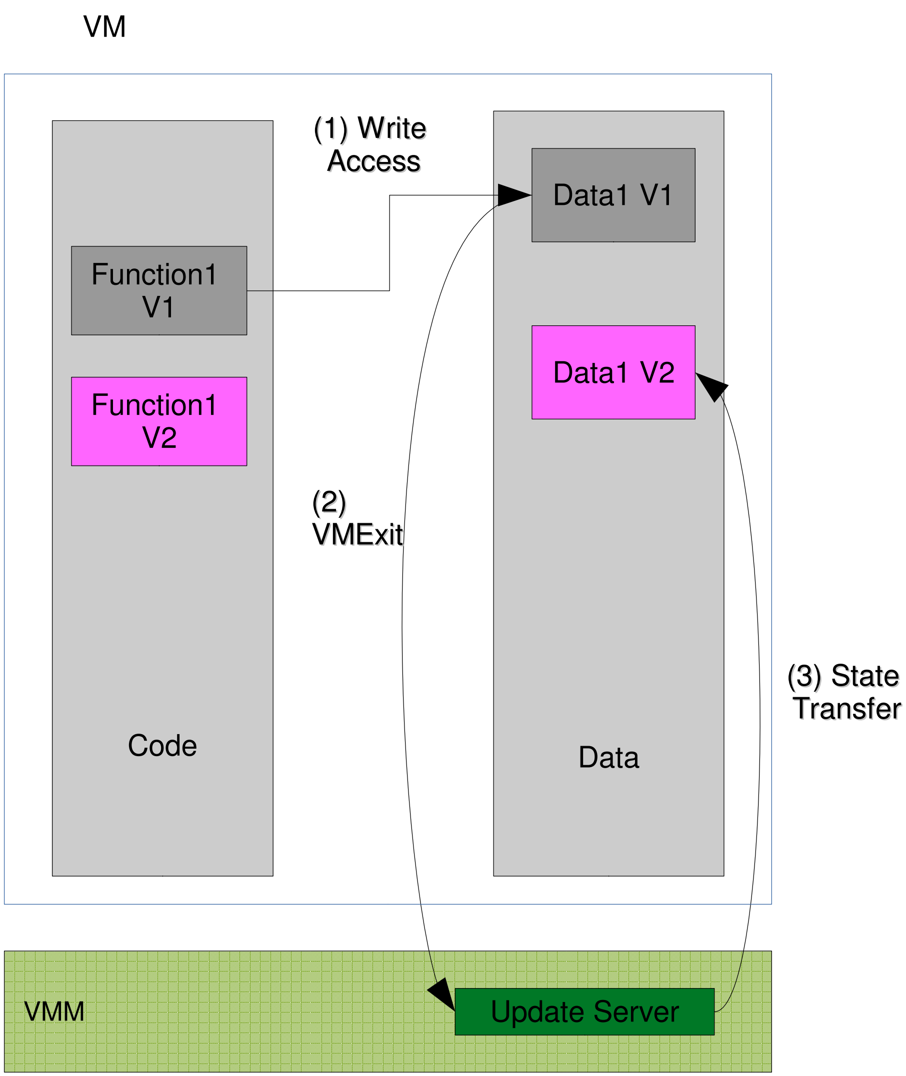
- Memory virtualization techniques provided by x86 architecture – Shadow paging & NPT/EPT
- Update Server resumes the VM
- Old function accesses to old data

LUCOS (6)



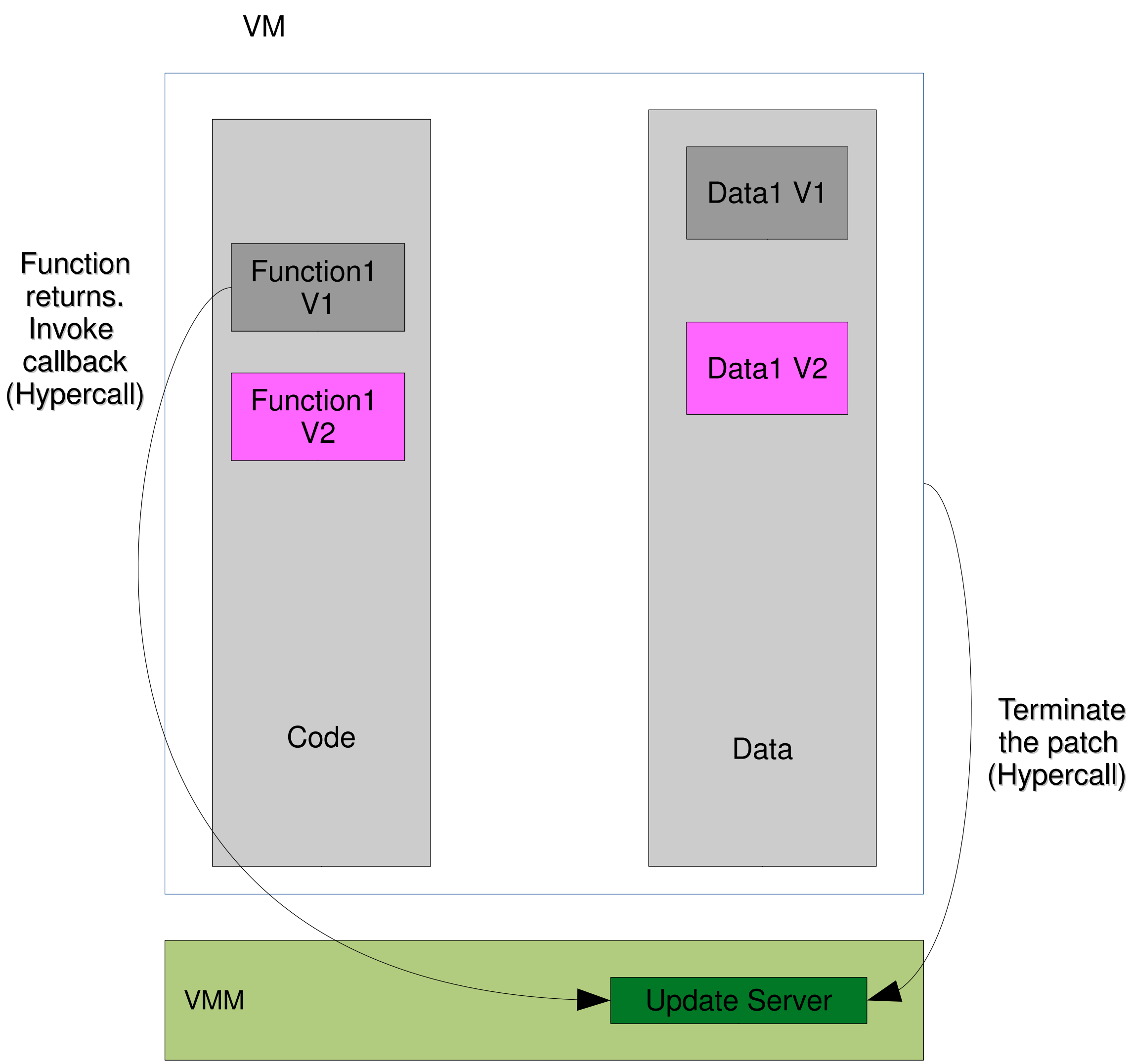
- Memory access intercepted
- Update Server checks if VM is accessing to either versions of the data

LUCOS (7)



- Update Server invokes state transfer function to maintain data consistency

LUCOS (8)



- Usage information of the old function and data is updated via callbacks
- Callbacks are invoked in the context of VMM
- Update Server terminates the patch when the old function and data is not in use

1. Classification of Kernel Updates
   - Updating Code Only
   - Updating Code and Existing Data

2. DynAMOS - The Basic Approach
   - Quiescence Detection
   - Binary Rewriting
   - Redirection Table

3. LUCOS - Using Virtualization for Live Updates
   - State Transfer

4. Ksplice - Hot Updates at Object Code Level
   - Pre-post Differencing and Run-pre Matching

5. Conclusion and Discussion

Ksplice (1)

- Ksplice Inc. :Created by four MIT students based on a master's thesis
- Provides prebuilt and tested updates for the Red Hat, CentOS, Debian, Ubuntu and Fedora Linux distributions
- Acquired by Oracle on 21 July 2011
- Used by over 700 customers running more than 100,000 production systems at that time

Ksplice (2)

- Creating patches manually: quite complex and error prone
- Automatic patch creation
- Analysis at the Executable and Linkable Format (ELF) object code layer
  - Doesn't matter if it's C or Assembly code
  - Inlined functions detected
- Most of the Linux security patches do not make semantical changes to data structures

Ksplice (3)

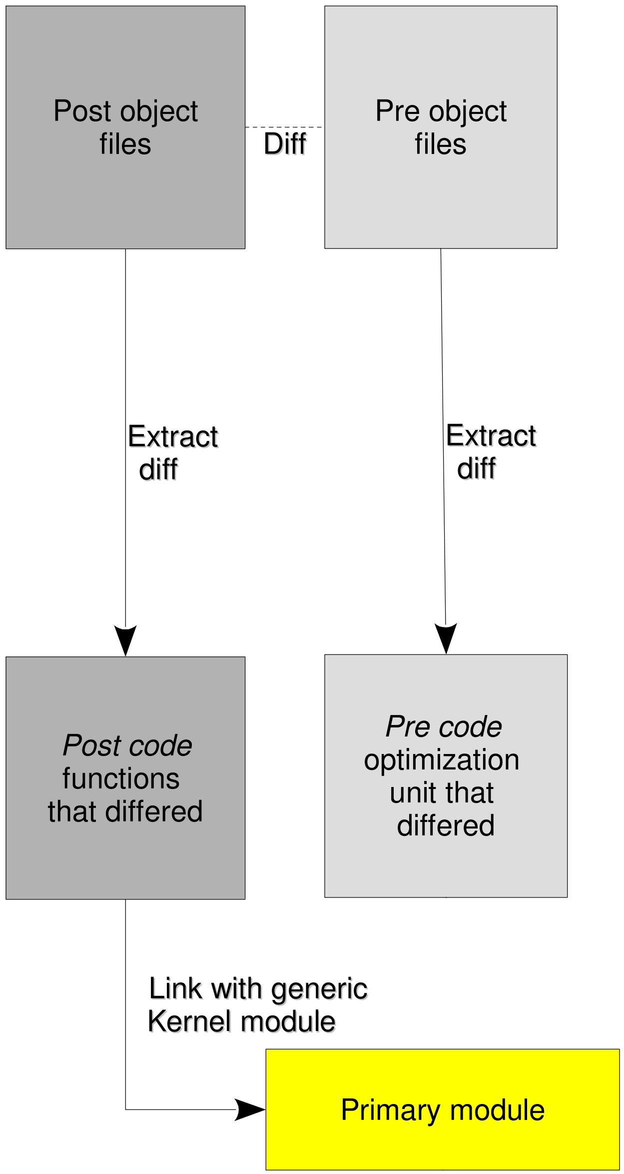
- Input:
  - Original source (*pre source*) of the running kernel (buggy).
  - The code in the running kernel (*run code*) (buggy).
  - Source of the patched kernel (*post source*).
- Preparation
  - Compile the *pre source* and *post source* using `-ffunction-sections` and `-fdata-sections` compiler options (gcc)
  - *Pre* and *post* object files created

Ksplice (4)
Pre-post Differencing and Run-pre Matching

Steps:

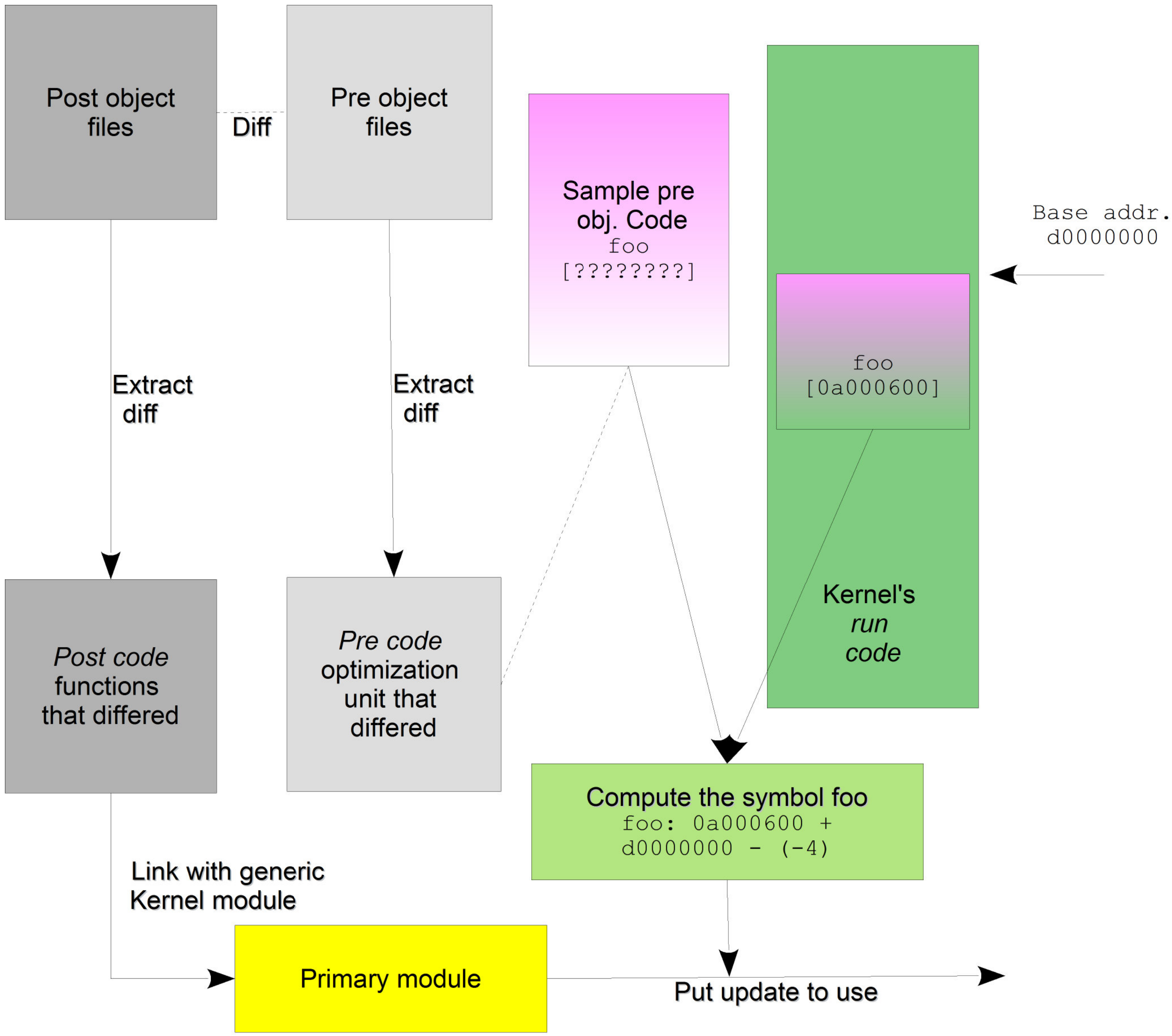
- Compare the *pre* and *post* object files
- Detect and replace kernel functions that have been changed
- Calculate symbols
- Detect quiescence state
- Patch

Ksplice (5) pre-post differencing



- Primary module has unresolved symbols

Ksplice (6) run-pre matching



- Reversing what the Linker did
- Symbol tables of Linux Kernel is not used
- Allows accessing to every symbol in the kernel
- Actually, Linux Kernels without any symbol tables can be patched.

Ksplice (7)

Quiescence Detection:

- Calls `stop_machine` for detecting quiescence state: Makes patching atomic. Causes 0.7 milliseconds delay
- Stack-walkthrough used for quiescence detection
- If check failed: wait couple of seconds, check again
- Using Ksplice for customer support: Diagnostic tool sends the report to oracle. Oracle prepares a bugfix as a ksplice patch.

1. Classification of Kernel Updates
    - Updating Code Only
    - Updating Code and Existing Data

2. DynAMOS - The Basic Approach
    - Quiescence Detection
    - Binary Rewriting
    - Redirection Table

3. LUCOS - Using Virtualization for Live Updates
    - State Transfer

4. Ksplice - Hot Updates at Object Code Level
    - Pre-post Differencing and Run-pre Matching

5. Conclusion and Discussion

Conclusion

- Binary rewriting and stack-walkthrough is used in such frameworks.
- Keeping the old code consistent with the new code is complex and expensive (state transfers, callbacks).
- LUCOS exploits virtualization technologies, whereas Ksplice operates on object code level.
- Ksplice: not limited to C or Assembly code. But compiler and linker dependent.
- Ksplice: Minimum programmer involvement. %88 of the security patches from May 2005 to May 2008 can be applied automatically.

Discussion

- How reliable are usage counters in LUCOS
- Applying LUCOS in multi-core platforms
- Applying Ksplice and LUCOS on real time operating systems
- Patch rollback