# DynAMOS manual

## RELEASE_0-5-3

**The DynAMOS Team**

**DynAMOS manual: RELEASE_0-5-3**

by The DynAMOS Team

Published 2007-05-13

This is the documentation of DynAMOS, a dynamic kernel updating system for commodity operating system kernels.

# Table of Contents

# List of Figures

# Chapter 1. Introduction

## 1.1. What Is It?

DynAMOS is a dynamic kernel updating system for commodity operating system kernels. It can safely, unobtrusively, and adaptively apply reversible updates of non-quiescent kernel subsystems and datatypes. No kernel source code modifications or system reboot are required.

## 1.2. Example Updates

Dynamic updates DynAMOS is able to successfully carry out include:

- Extending the Linux 2.2 kernel process scheduler to support unobtrusive, fine-grain cycle stealing.
- Introducing adaptive memory paging for efficient gang-scheduling in a Linux 2.4 cluster.
- Adaptively updating the Linux pipefs implementation during large data transfers.
- Introducing process checkpointing in Linux 2.4.
- Applying security fixes provided by the OpenWall project.
- Injecting performance monitoring functionality in kernel functions.
- Updating DynAMOS itself.

# Chapter 2. Approach

DynAMOS is founded on a new dynamic instrumentation technique called *adaptive function cloning*. Dynamic instrumentation is the technique of inserting code instruments directly in the memory image of a running system.

## 2.1. Related Work

In support of dynamic updates, some operating systems have been designed from scratch to be adaptable or hot-swappable. Examples include *SPIN*, the *Exokernel* and *K42*. These solutions require significant changes in the way the operating system and applications are crafted. They cannot be generally applied to commodity operating systems like Linux without kernel source code modifications.

Dynamic instrumentation systems like *KernInst* and *GILK* made it possible to instrument kernel code in fixed (e.g. SPARC) and variable (e.g. i386) instruction-length architectures respectively. However, they have not addressed the issue of dynamic software updates in general. They focus on performance profiling, including the tools *DTrace* and *Pin*.

Other dynamic software updating systems include *DYMOS*, and *Ginseng*, but they are limited to userspace updates. These systems recompile user programs enabling them to be dynamically updateable. They cannot be dynamically applied to a running system.

Binary rewriters like *ATOM* and *EEL* also do not support dynamic software updates.

## 2.2. Architecture

DynAMOS is developed in C and assembly using the GNU toolchain, currently for the i386 architecture for Linux kernels 2.2-2.6. The mechanisms it employs can also be generally applied to fixed instruction-length architectures. It has been designed with portability in mind and can be enhanced with support for other commodity operating systems besides Linux, such as FreeBSD, MacOS, Solaris, and AIX.

The system consists of a kernel module providing the dynamic updating framework and a userspace collection of tools that can build, apply and manage dynamic kernel updates using the framework.

## 2.3. Adaptive Function Cloning

Unlike existing dynamic instrumentation systems, DynAMOS does not apply instruments at the basic

block level, but instead updates complete functions. It uses an execution flow redirection technique that permits concurrent execution of multiple versions of a function. Updates can be autonomously applied by the kernel based on rules defined by the user. An adaptation handler can be developed that determines prior to invocation of a function the appropriate version of the function that should be called. This capability of adaptively switching between multiple function versions makes DynAMOS the first dynamically applied *adaptive* kernel updating system.

## 2.4. Safe Execution Flow Redirection

Execution flow redirection is accomplished by installing a trampoline in the beginning of a function. It is guaranteed that installation is safe from sleeping processes.

**Note:** There are plans to support safe execution flow redirection in multi-processor systems.

## 2.5. Version Manager

The list of possible updates is maintained in a version database inside the kernel module. This version manager tracks the list of possible different versions of a particular function. All updates must first be registered with the version manager before they can be dynamically applied.

By design, the original version of a function (the one that is active since bootup) must be the first version of a function that is registered with the version manager. This action installs the execution flow redirection mechanism. Alternate versions can only be registered after execution flow can be redirected away from the original.

---

### Caution

Registering an alternate version of a function first, will result in the execution flow redirection mechanism applied to the alternate function instead of the original. This would render the redirection ineffective.

---

## 2.6. Safe Removal

Unregistering alternate functions from the version manager is safeguarded by a check for quiescence. Before a version of a function is removed, it is verified that it is not in use by the program counter or stack of any process.

## 2.7. Function Image Relocation

The adaptive function cloning technique relocates function images to make them adaptively updateable. A series of checks safeguard from and can handle potentially unsafe relocations, such as:

- Backward branches.

- Data-in-code.

- Outbound branches.

- Indirect outbound branches.

- Multiple entrypoints.

## 2.8. Symbol Resolution

The original kernel image object file, such as `vmlinux` in Linux, is consulted when it is desired to update symbols (variables or functions) that were not exported in the original kernel source.

## 2.9. Datatype Updates

For some types of updates it is necessary to update the datatype of a variable. When a new field is added, there isn't any reserved room in the data structure to host the field. DynAMOS uses *shadow* variables in support of such datatype updates. On variable creation a new shadow variable is created. The memory address of the variable is used to map into it's shadow using a hash table. When the new datatype is freed, it's shadow is also freed. A benefit of this technique is that only the functions that must use the new field need to be updated, instead of all functions that use the old datatype.

## 2.10. Constructing Updates

Constructing kernel updates is currently a manual process. There are plans to introduce a semi-automatic tool that, given as input a diff file, will automatically produce alternate versions of the functions that need to be updated.

Updates to functions are constructed by duplicating in source code the originals, applying modifications in the source, naming the functions differently (e.g. append the postfix "`_v2`"), and recompiling them with the compiler and kernel flags used to originally compile the kernel.

Updates to variables are constructed as a function containing logic that must be executed to update the variables. This function is executed once when the update is applied.

# Chapter 3. Installation

## 3.1. Availability

DynAMOS is available for Linux 2.2-2.6 on the i386 architecture. There are plans to port the system to FreeBSD (http://bugzilla.mkgnu.net/show_bug.cgi?id=121) and OpenSolaris (http://bugzilla.mkgnu.net/show_bug.cgi?id=901). It's webpage (http://www.mkgnu.net/?q=dynamos) contains the most up to date information on the project, including the latest release and manual.

A users mailing list is available for subscription (http://lists.mkgnu.net/mailman/listinfo/dynamos-users), or simply for sending email (mailto:dynamos-users@lists.mkgnu.net).

## 3.2. Installation

DynAMOS is available in the form of Debian and RPM packages, but is not yet available in source code form. The provided packages are:

- `dynamos-framework`: The DynAMOS system.
- `dynamos-doc`: Documentation including this manual.

# Chapter 4. Framework

This section provides instructions on using the DynAMOS framework to dynamically apply updates.

## 4.1. Starting

The framework kernel module must first be loaded. This is accomplished as shown in Figure 4-1.

**Figure 4-1. Starting the framework.**

```
bash$ su -
Password:
bash# /etc/init.d/dynamos start
```

## 4.2. Stopping

Stopping the framework when updates are already applied could crash the system. Before stopping the framework it is recommended that any applied updates are properly deactivated.

Stopping the framework is accomplished as shown in Figure 4-2. Function updates that are already in effect will be automatically reversed, but not in any particular order that guarantees safe deactivation of the update.

**Figure 4-2. Stopping the framework.**

```
bash$ su -
Password:
bash# /etc/init.d/dynamos stop
```

## 4.3. Control Tool

The framework is managed with the dynamos_control tool. Additional programs and scripts are built on top of calls to this tool.

# 4.4. Control Commands

One way of controlling the framework is by writing shell scripts containing special macros. These scripts are supplied as an argument to the tool `dynamos_run_commands`. This tool will translate the macros and produce a new script with the postfix "`.translated.sh`" containing the actual calls to the *Control Tool*.

> **Note:** The current method of issuing control commands is weak. It does not support retrieving the return value of the control commands to react appropriately (e.g. abort the update on error). It also lacks a macro of defining a group of function updates that should be applied atomically. There are plans to replace this method with a Perl-based library of calls. Control programs will then be written in a more reliable way and in a more powerful language.

The list of the existing control command macros follows:

- DYNREPLACE_REGISTER_INTUITIVELY(update_name, id, function_name)

  This macro is used to register updates with the version manager.

  By convention, the `id` is a number indicating the number of parameters the function accepts. Multiple updates may be registered in the version manager with the same `update_name` but different `id`s. Such updates would correspond to different routines. And for each `<update_name, id>` pair, multiple `function_name`s could be registered as the alternate versions of this update. `update_name`s must be enclosed in double(") quotes.

- DYNREPLACE_DEREGISTER_FUNCTION(update_name, id, version)

  This macro is used to unregister specific versions of updates.

  Attempting to unregister version 0 will unregister all versions for the particular `<update_name, id>` pair.

- DYNREPLACE_ACTIVATE_FUNCTION(update_name, id, version)

  This macro is used to activate specific versions of updates.

  By convention, the original version of an update is version number 1. To deactivate an update, activate version number 1.

- DYNREPLACE_SET_PREACTIVATION_HOOK_BY_NAME(update_name, id, version, function_name)

  This macro is used to set a hook that will be executed before a particular version of a specific `<update_name, id>` pair is activated.

- DYNREPLACE_SET_POSTACTIVATION_HOOK_BY_NAME(update_name, id, version, function_name)

  This macro is used to set a hook that will be executed after a particular version of a specific `<update_name, id>` pair is activated.

- DYNREPLACE_SET_PREREMOVAL_HOOK_BY_NAME(update_name, id, version, function_name)

  This macro is used to set a hook that will be executed before a `<update_name, id>` pair is removed from the version manager.

- DYNREPLACE_SET_POSTREMOVAL_HOOK_BY_NAME(update_name, id, version, function_name)

  This macro is used to set a hook that will be executed after a `<update_name, id>` pair is removed from the version manager.

- DYNREPLACE_SET_RULE_EVALUATION_FUNCTION_BY_NAME(update_name, id, version, function_name)

  This macro is used to define an adaptation handler. There can be only one adaptation handler per `<update_name, id>` pair.

- DYNREPLACE_SET_RULE_EVALUATION_FUNCTION_BY_ADDRESS(update_name, id, version, memory_address)

  This macro is also used to define an adaptation handler. Supplying a `memory_address` of `0` will remove an existing adaptation handler.

- DYNREPLACE_CALL(function_name)

  This macro is used to invoke an initialization function.

Figure 4-3 shows parts of actual control commands used to enable the Linger-Longer system, adaptive updating of the pipefs implementation, and enable EPCKPT process checkpointing.

**Figure 4-3. Example control commands.**

```
# Linger-Longer -- Update the kswapd thread
DYNREPLACE_REGISTER_INTUITIVELY("interruptible_sleep_on", 1, interruptible_sleep_on)
DYNREPLACE_REGISTER_INTUITIVELY("interruptible_sleep_on", 1, interruptible_sleep_on_v2)
DYNREPLACE_ACTIVATE_FUNCTION("interruptible_sleep_on", 1, 2)

DYNREPLACE_REGISTER_INTUITIVELY("kswapd", 1, kswapd)
DYNREPLACE_REGISTER_INTUITIVELY("kswapd", 1, kswapd_ll)
DYNREPLACE_SET_PREACTIVATION_HOOK_BY_NAME("kswapd", 1, 1, kswapd_pre_activation_hook)
DYNREPLACE_SET_POSTACTIVATION_HOOK_BY_NAME("kswapd", 1, 1, kswapd_post_activation_hook)
DYNREPLACE_SET_PREACTIVATION_HOOK_BY_NAME("kswapd", 1, 2, kswapd_ll_pre_activation_hook)
DYNREPLACE_SET_POSTACTIVATION_HOOK_BY_NAME("kswapd", 1, 2, kswapd_ll_post_activation_hook
DYNREPLACE_ACTIVATE_FUNCTION("kswapd", 1, 2)

# pipefs -- create an adaptation handler
DYNREPLACE_REGISTER_INTUITIVELY("pipe_read", 4, pipe_read)
DYNREPLACE_REGISTER_INTUITIVELY("pipe_read", 4, pipe_read_v2)
DYNREPLACE_REGISTER_INTUITIVELY("pipe_read", 4, pipe_read_v3)
DYNREPLACE_ACTIVATE_FUNCTION("pipe_read", 4, 2)
DYNREPLACE_SET_RULE_EVALUATION_FUNCTION_BY_NAME("pipe_read", 4, pipe_adaptation_handler_

# Disable the adaptation handler
DYNREPLACE_SET_RULE_EVALUATION_FUNCTION_BY_ADDRESS("pipe_write", 4, 0)
DYNREPLACE_SET_RULE_EVALUATION_FUNCTION_BY_ADDRESS("pipe_read", 4, 0)

# Unregister some functions
DYNREPLACE_DEREGISTER_FUNCTION("pipe_write",  4, 0)
DYNREPLACE_DEREGISTER_FUNCTION("pipe_read",  4, 0)
DYNREPLACE_DEREGISTER_FUNCTION("pipe_release",  3, 0)

# EPCKPT -- Set a preremoval hook
DYNREPLACE_SET_PREREMOVAL_HOOK_BY_NAME("restart_binary", 2, epckpt_cleanup)

# Invoke an initialization function
DYNREPLACE_CALL(epckpt_init)
```

# 4.5. Writing Adaptation Handlers

An API is available for writing adaptation handlers. Calls to activate a different version of a function, or query the framework for presence of a specific version of a function are available. Part of the function signature of the adaptation handler is defined by the framework, and the remaining arguments could match the original arguments of the function on which the adaptation handler is applied.

Figure 4-4 shows the signature of `pipe_write`, the producer function of `pipefs` in Linux 2.4. Figure 4-5 shows an example adaptation handler written for `pipe_write`. The arguments supplied to the original `pipe_write` are still accessible on the stack and can be used to determine which version of `pipe_write` should run next. In this (simplified) example, when more than 64K of data are written through a pipe the second version of `pipe_write` is used.

**Figure 4-4. Function signature of `pipe_write`.**

```
static ssize_t
pipe_write(struct file *filp, const char *buf,
           size_t count, loff_t *ppos)
```

**Figure 4-5. Example adaptation handler for `pipe_write`.**

```
long pipe_write_total_count = 0;

void
pipe_write_adaptation_handler(dynreplace_version_table_entry_t *entry,
                              redirection_state_t redir_state,
                              rule_evaluation_call_state_t rec_state,
                              struct file *filp, char *buf,
                              size_t count, loff_t *ppos)
{
  pipe_write_total_count += count;
  dynreplace_version_table_lock();

  if (pipe_write_total_count > 64 * 1024)
    dynreplace_activate_function(&entry->unique, 2);
  else
    dynreplace_activate_function(&entry->unique, 1);

  dynreplace_version_table_unlock();
}
```

# 4.6. Updating Datatypes

Datatype mappings are maintained in a separate hash table per datatype. The table must be initialized before applying an update, and freed when reversing the update, as shown in Figure 4-6. Figure 4-7 shows a set of new fields that should be added in the existing datatype definition of `struct task_struct`, the process control block in Linux 2.4. All these fields are grouped in a new datatype definition called `struct new_task_struct`, instead of extending the existing definition.

**Figure 4-6. Managing a datatype mapping table.**

```
/* Mapping table for the updated datatype of struct task_struct. */
```

```
dynreplace_access_t access_new_task_struct;

void epckpt_init()
{
  dynreplace_access_init(&access_new_task_struct);
}

void epckpt_cleanup()
{
  dynreplace_access_cleanup(&access_new_task_struct);
}
```

**Figure 4-7. Definition of `struct new_task_struct`.**

```
struct new_task_struct {
int collect_ckpt_data:1;
struct mmap_list *mmap_list;
struct shmem_list *shmem_list;
struct sem_list *sem_list;
};
```

A datatype definition, like the one in Figure 4-7, may define the initial values fields will need to be initialized to. This initialization must be carried out manually. Shadow variables must be instantiated when the original datatype is instantiated, and a mapping between the original and its shadow must be established. Figure 4-8 shows an example creating a shadow variable instance on `do_fork` which will hold the new fields of `struct new_task_struct` as defined in Figure 4-7. A shadow instance is created, and a reference to the shadow is obtained. The reference is used to access the new fields.

**Figure 4-8. Creating a shadow variable in `do_fork`.**

```
int do_fork_v2(unsigned long clone_flags, unsigned long stack_start,
       struct pt_regs *regs, unsigned long stack_size)
{
  ...
  struct task_struct *p;
  void *new_p;

  ...

  dynreplace_access_lock(&access_new_task_struct);

  /* Create a shadow instance for this task_struct */
  dynreplace_access_create(&access_new_task_struct,
                           (void *)p, sizeof(struct new_task_struct));

  /* Obtain a reference to the shadow instance */
  new_p = dynreplace_access_find(&access_new_task_struct, (void *)p);

  /* Initialize the instance */
```

```
  if (new_p != NULL) {
    /* Access the various fields */
    dynreplace_access_field(struct new_task_struct, new_p, collect_ckpt_data) = 1;
    dynreplace_access_field(struct new_task_struct, new_p, mmap_list) = NULL;
    dynreplace_access_field(struct new_task_struct, new_p, shmem_list) = NULL;
    dynreplace_access_field(struct new_task_struct, new_p, sem_list) = NULL;

    /* Use the new fields to change control flow as required by the update. */
    if (dynreplace_access_field(struct new_task_struct, new_p, collect_ckpt_data)) {
      /* Perform process checkpointing bookkeeping */
      ...

    }
  }

  dynreplace_access_unlock(&access_new_task_struct);

 ...

}
```

Shadow variable instances must be freed when the original datatype is freed. Figure 4-9 shows an example freeing the shadow variable instance on do_exit.

**Figure 4-9. Freeing a shadow variable in do_exit.**

```
ATTRIB_NORET NORET_TYPE void do_exit_v2(long code)
{
  struct task_struct *tsk = current;

 ...

  {
    void *new_current;

    dynreplace_access_lock(&access_new_task_struct);

    new_current = dynreplace_access_find(&access_new_task_struct, (void *)current);
    if (new_current != NULL)
      dynreplace_access_remove(&access_new_task_struct, (void *)current);

    dynreplace_access_unlock(&access_new_task_struct);
  }

 ...

}
```

On shadow variable instance creation and deletion the code that accesses the new fields and the code that frees the shadow variable is written in a manner that permits existing kernel code paths to execute without dereferencing a NULL pointer. For example, a process executing `do_exit` that has not had a shadow variable instance created (because when the process was created the update had not been applied) will still execute normally. Similarly, code in other functions that uses the new fields should not be executed if there is no shadow instance available. In essence, it is possible to have multiple versions of a function concurrently executing safely.

# 4.7. Control Center

A GUI tool that controls the framework will be available.

# Chapter 5. Examples

DynAMOS is distributed with examples of kernel updates in source code form for Linux 2.4. The source code can be compiled and prepared to be applied as an update. The updated functionality can be loaded in the kernel and activated using the DynAMOS framework. The following examples demonstrate how the collection of tools offered by DynAMOS are used together from start to finish in dynamically applying a kernel update.

The general format of running these examples is shown in Figure 5-1.

**Figure 5-1. Running the example updates.**

```
# DynAMOS must run with administrator privileges.
bash$ su -
Password:
bash#

# Start the framework
bash# /etc/init.d/dynamos start

# Check if everything started correctly
bash# dmesg -c

# Build an example's source
bash# cd get_pid
bash# make

# Load the example's module
bash# insmod final_dynreplace_file.o

# Activate the update
bash# ./activate.sh

# Verify things work
bash# dmesg -c
bash# cat /dev/dynamos

# Deactivate the update
bash# ./deactivate.sh

# Verify the update is not effective
bash# dmesg -c
bash# cat /dev/dynamos

# Unload the example's module
bash# rmmod final_dynreplace_file

# Stop the framework
bash# /etc/init.d/dynamos stop
```

```
# Check if everything stopped correctly
bash# dmesg -c
```

# 5.1. Updating `get_pid`

`get_pid()` is the Linux process allocation routine. It returns the next available process id that can be used by a newly created process. When applied, this update will report in the kernel logs on process creation the id of the process that calls `get_pid()` and the pid it returns, as shown in Figure 5-2.

**Figure 5-2. Expected output of `get_pid()` update.**

```
This is get_pid_v2 from 3012 and will return pid 3018.
```

# 5.2. Updating The Process Scheduler

`schedule()` is the Linux process scheduler. When applied, this update will report an informative message in the kernel logs after every 5000 invocations of the scheduler, as shown in Figure 5-3.

**Figure 5-3. Expected output of `schedule()` update.**

```
schedule_v2 called for 5000 times (25000 times total), this time from 3174.
```

# Appendix A. FAQ

This FAQ includes questions not covered elsewhere in this manual.

# Glossary

# A

### ATOM

*ATOM* is a library for modifying object files.

# D

### DTrace

*DTrace* (http://www.sun.com/bigadmin/content/dtrace/) is a comprehensive dynamic tracing framework for the Solaris Operating Environment. DTrace provides a powerful infrastructure to permit administrators, developers, and service personnel to concisely answer arbitrary questions about the behavior of the operating system and user programs.

### DYMOS

*DYMOS* (http://www.cis.upenn.edu/~lee/mydissertation.doc) is a dynamic modification system.

# E

### EEL

*EEL* (http://www.cs.wisc.edu/~larus/eel.html) is a C++ library that hides much of the complexity and system-specific detail of editing executables. EEL provides abstractions that allow a tool to analyze and modify executable programs without being concerned with particular instruction sets, executable file formats, or consequences of deleting existing code and adding foreign code. EEL greatly simplifies the construction of program measurement, protection, translation, and debugging tools. EEL differs from other systems in two major ways: it can edit fully-linked executables, not just object files, and it emphasizes portability across a wide range of systems.

### Exokernel

*Exokernel* (http://pdos.csail.mit.edu/exo.html) is the name of an operating system kernel developed by the Parallel and Distributed Operating Systems group at MIT, and of a class of similar operating systems. An exokernel eliminates the notion that an operating system should provide abstractions on which applications are built. Instead, it concentrates solely on securely multiplexing the raw hardware: from basic hardware primitives, application-level libraries and servers can directly implement traditional operating system abstractions, specialized for appropriateness and speed.

# G

### GILK

*GILK* (http://www.doc.ic.ac.uk/~djp1/gilk.html) is a dynamic instrumentation tool for the Linux 2.2 kernel.

### Ginseng

*Ginseng* (http://www.cs.umd.edu/projects/dsu/) is a dynamic updating system for userspace programs.

# K42

### K42

*K42* (http://www.research.ibm.com/K42/) is a high performance, open source, general-purpose research operating system kernel for cache-coherent multiprocessors.

### KernInst

*KernInst* (http://www.paradyn.org/html/kerninst.html) is a framework for dynamically splicing code into a running kernel, almost anywhere, anytime.

# P

## Pin

*Pin* (http://rogue.colorado.edu/pin/) is a tool for the dynamic instrumentation of programs. Pin was designed to provide functionality similar to the popular ATOM toolkit for Compaq's Tru64 Unix on Alpha, i.e. arbitrary code (written in C or C++) can be injected at arbitrary places in the executable. Unlike ATOM, Pin does not instrument an executable statically by rewriting it, but rather adds the code dynamically while the executable is running. This also makes it possible to attach Pin to an already running process.

# S

## SPIN

*SPIN* (http://www-spin.cs.washington.edu/) is an operating system that blurs the distinction between kernels and applications. Applications traditionally live in user-level address spaces, separated from kernel resources and services by an expensive protection boundary. With SPIN, applications can specialize the kernel by dynamically linking new code into the running system. Kernel extensions can add new kernel services, replace default policies, or simply migrate application functionality into the kernel address space. Sensitive kernel interfaces are secured via a restricted linker and the type-safe properties of the Modula-3 programming language. The result is a flexible operating system that helps applications run fast but doesn't crash.