

UpStare manual

RELEASE_0-12-8

Kristis Makris <mkgnu@mkgnu.net>

UpStare manual: RELEASE_0-12-8

by Kristis Makris <mkgnu@mkgnu.net>

Published 2011-01-14

This is the documentation of UpStare, a system that can apply immediate dynamic software updates in multi-threaded userspace applications using stack reconstruction.

Table of Contents

| | |
|--|-----------|
| 1. About | 1 |
| 1.1. Copyright Information | 1 |
| 1.2. Disclaimer | 2 |
| 1.3. Document Conventions | 2 |
| 2. Introduction | 4 |
| 2.1. What Is It? | 4 |
| 2.2. Example Updates | 4 |
| 3. Installation | 6 |
| 3.1. Availability | 6 |
| 3.2. Installation..... | 6 |
| 4. Preparing Updateable Programs | 7 |
| 4.1. Invoking Wrapper Build Programs | 7 |
| 4.2. Exporting Dynamic Symbols | 8 |
| 4.3. Additional API Calls | 8 |
| 4.4. Understanding The Compiler And Its Options | 10 |
| 5. Preparing Dynamic Software Updates | 13 |
| 5.1. Preparing An Updateable Original Version | 13 |
| 5.2. Preparing An Updateable New Version | 13 |
| 5.3. Describing Dynamic Software Updates | 13 |
| 5.3.1. Describing Function Updates | 14 |
| 5.3.2. Describing Datatype Updates | 16 |
| 5.3.3. Describing Execution Continuation | 17 |
| 5.3.4. Describing Update Constraints | 21 |
| 5.4. Running The Patch Generator | 23 |
| 5.5. Compiling The Dynamic Software Update Patch | 27 |
| 6. Applying Dynamic Software Updates | 29 |
| 7. System Internals | 30 |
| 7.1. Function Call Indirection | 30 |
| 7.2. Thread Entry-Points | 30 |
| 7.3. Signal Handlers | 30 |
| 7.4. Update Points | 30 |
| 7.5. Exported Local Variables | 31 |
| 7.6. Multi-Threaded Updates | 31 |
| 7.7. Multi-Process Updates | 32 |
| 7.8. Blocking System Calls | 32 |
| 7.9. Dynamic Stack Tracing | 33 |

List of Figures

| | |
|--|----|
| 3-1. Forcing installation of RPM packages. | 6 |
| 3-2. Forcing installation of Debian packages. | 6 |
| 4-1. Original <code>Makefile</code> for PostgreSQL 7.4.16. | 7 |
| 4-2. <code>Makefile</code> modifications for an updateable PostgreSQL 7.4.16. | 7 |
| 4-3. Test program that attempts to be updated inside a signal handler. | 9 |
| 4-4. Describing a compiler configuration. | 11 |
| 5-1. Describing function updates for vsFTPD from 2.0.4 to 2.0.5. | 14 |
| 5-2. Transforming datatypes for vsFTPD from 2.0.4 to 2.0.5. | 16 |
| 5-3. Viewing the execution continuation points of <code>vsf_standalone_main</code> in vsFTPD 2.0.4. | 18 |
| 5-4. Describing execution continuation when updating from Bubblesort to Heapsort. | 19 |
| 5-5. Describing update constraints. | 21 |
| 5-6. Preparing a dynamic software update patch for vsFTPD from 2.0.4 to 2.0.5. | 23 |
| 5-7. Patch generator report to update vsFTPD from 2.0.4 to 2.0.5. | 23 |
| 5-8. Patch generator warnings for vsFTPD from 2.0.4 to 2.0.5. | 26 |
| 5-9. Compiling a dynamic software update patch for vsFTPD from 2.0.4 to 2.0.5. | 28 |
| 6-1. Applying a dynamic software update for vsFTPD from 2.0.4 to 2.0.5. | 29 |

Chapter 1. About

1.1. Copyright Information

This system, including this document, is Copyright (C) 2006 by Kristis Makris <mkgnu@mkgnu.net> (mailto:mkgnu@mkgnu.net). In binary form produced and digitally signed specifically by Kristis Makris, it is freely distributed for personal and academic/research use. It is NOT distributed for commercial use. No other binary forms can be distributed.

The source code and documentation, either in electronic or hardcopy format, may not be used for any purpose whatsoever without the written permission of Kristis Makris.

The CIL framework used by this system (but NOT the CIL modules written specifically for THIS system) is under a different license (found in src/multi_threaded_updates/cil/LICENSE). That license is reproduced below:

```
Copyright (c) 2001-2007,  
George C. Necula    <necula@cs.berkeley.edu>  
Scott McPeak      <smcpeak@cs.berkeley.edu>  
Wes Weimer       <weimer@cs.berkeley.edu>  
Ben Liblit       <liblit@cs.wisc.edu>  
Matt Harren      <matth@cs.berkeley.edu>  
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of the contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

(See <http://www.opensource.org/licenses/bsd-license.php>)

1.2. Disclaimer

No liability for the contents of this document can be accepted. Follow the instructions herein at your own risk.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.3. Document Conventions

This document uses the following conventions:

| Descriptions | Appearance |
|---|---|
| Warning | <div style="border: 2px solid black; padding: 5px; display: inline-block;"> <p style="text-align: right; margin: 0;">Caution</p> <p>Don't run with scissors!</p> </div> |
| Hint | Tip: Would you like a breath mint? |
| Note | Note: Dear John... |
| Information requiring special attention | <div style="border: 2px solid black; padding: 5px; display: inline-block;"> <p style="text-align: right; margin: 0;">Warning</p> <p>Read this or the cat gets it.</p> </div> |
| File or directory name | <code>filename</code> |
| Command to be typed | command |
| Application name | application |

Descriptions

Normal user's prompt under bash shell
Root user's prompt under bash shell
Environment variables
Code example

Appearance

bash\$
bash#
VARIABLE
<para>
Beginning and end of paragraph
</para>

This documentation is maintained in DocBook 4.2 SGML format.

Chapter 2. Introduction

2.1. What Is It?

UpStare is a system that can apply immediate dynamic software updates in multi-threaded programs. It provides a compiler that applies high-level, source-to-source transformations that make programs dynamically updateable. It is implemented in OCaml using the CIL v1.3.6 framework, C, and Perl, is architecture and operating system independent, and has been successfully deployed on various UNIX-like systems. Programmers only need to replace in their build process (e.g. Makefiles) calls to an existing compiler like `gcc` to the UpStare compiler (`hcucc.pl`).

UpStare can update applications while they are live and running without stopping the application at all. It achieves this goal by reconstructing the process stack and updating the program state in a single step. This eliminates the possibility of an old version of the code to be executed on a newer version of a datatype representation (a type-safety error). The updates are applied immediately, even in multi-threaded programs. It is not necessary to wait indefinitely for a quiescent program state, such as waiting for a program thread to receive data from a network socket before an update can begin. A running algorithm can be updated midstream its execution and resumed from a different point (not necessarily the beginning) of another algorithm that is behaviorally equivalent: an algorithm that aims to produce the same results while reusing existing progress.

2.2. Example Updates

Example updates that UpStare has been able to successfully carry out include:

- **Recursion:** Updating an application recursively computing Fibonacci numbers, while nested deep in the stack, to report additional information when the recursion unwinds.
- **Network Sockets:** Updating a server application while serving multiple clients without closing the network socket.
- **Multi-threaded Applications:** Updating the main function body executed by multiple threads of an application. Also, updating in a producer/consumer multi-threaded application only the consumer threads while the producer threads remained unmodified.
- **Multi-nested Long-Lived Loops:** Updating in the middle of executing Bubblesort, a multi-nested long-lived loop, to continue executing from the middle of a different multi-nested long-lived loop implementing Selectionsort while reusing the existing program state. Additionally, updating from Bubblesort to Heapsort, which is a drastically different sorting algorithm executing over different loop iterators.
- **vsFTPd:** Applying 13 updates spanning 5.5 years of the multi-process (forked processes do not communicate) Very Secure FTP Daemon (about 12,000 lines of code).

- **PostgreSQL:** Updating the multi-process (forked processes communicate) PostgreSQL 7.4.16 database server (more than 200,000 lines of code).

Chapter 3. Installation

3.1. Availability

UpStare is available in binary format for UNIX-like systems and has been verified to work on Linux 2.4-2.6 on i386, Solaris 5.10 on SPARC, and Mac OS X on PowerPC, using the compilers `gcc-2.95`, `gcc-3.3`, `gcc-3.4`, `gcc-4.1`, `icc-8.0`, `icc-9.1`, and `icc-10.0`. The project webpage (<http://freshmeat.net/projects/upstare>) 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/upstare-users>), or simply for sending email (<mailto:upstare-users@lists.mkgnu.net>).

3.2. Installation

UpStare is available in the form of Debian and RPM packages. The provided packages are:

- `upstare-system`: The UpStare system. It includes the updateable compiler, runtime system, patch generator, and the tool that applies updates.
- `upstare-doc`: Documentation, including this manual.
- `upstare-tests`: The system tests which provide a large collection of example dynamic software updates.

Tip: If you believe your system meets the package dependencies, but installing packages fails due to missing dependencies, installation of the packages is still possible. Installation of RPM packages can be forced as shown in Figure 3-1, and installation of Debian packages can be forced as shown in Figure 3-2.

Figure 3-1. Forcing installation of RPM packages.

```
bash$ rpm -ivh --force --nodeps <RPM_PACKAGE_NAME>
```

Figure 3-2. Forcing installation of Debian packages.

```
bash$ dpkg -i --force-depends <DEB_PACKAGE_NAME>
```

Chapter 4. Preparing Updateable Programs

Preparing updateable programs requires no source code modifications in existing programs and minimal modifications to a program's build process (e.g. `Makefile`). Two types of modifications are necessary:

- Calls to an existing compiler, like `gcc`, must be changed to calls to the UpStare compiler, `hcucc.pl`.
- The flags supplied to a linker, like `ld`, must be modified to allow dynamically loaded shared objects (dynamic software update patches) to use the symbols of the built binary.

Figure 4-1 shows the original `Makefile` used for the PostgreSQL 7.4.16 database server and Figure 4-2 shows the modifications needed to this `Makefile` to prepare the PostgreSQL source code to become updateable. Section 4.1 and Section 4.2 explain the modifications applied to this `Makefile`.

Figure 4-1. Original `Makefile` for PostgreSQL 7.4.16.

```
...
CC = gcc
LD = ld
AR = ar
RANLIB = ranlib
LORDER = lorder
...
LDFLAGS =
```

Figure 4-2. `Makefile` modifications for an updateable PostgreSQL 7.4.16.

```
...
CC = hcucc.pl --merge --keepmerged --update-version=7416 --base-version
LD = hcucc.pl --merge --keepmerged --update-version=7416
AR = hcucil.pl --merge --mode=AR
RANLIB = echo
LORDER = echo
...
LDFLAGS = -Wl,--export-dynamic
```

4.1. Invoking Wrapper Build Programs

The build process is modified to invoke wrapper programs, such as a wrapper compiler, a wrapper linker, etc. These wrapper programs prepare updateable source code, but with a twist: the `.o` object files produced do not contain binary code, but text source code. During the linking process, a wrapper linker

performs whole-program merging and prepares a `_comb.c` file containing the source code of all object files. This final source code is then prepared to be updateable. The implication of this whole-program merging indirection is that some utilities used during the build process, besides the compiler and linker, may also need to also be wrapped.

In Figure 4-2, the "CC" variable, indicating the compiler used, needs to invoke the UpStare `hcucc.pl` compiler. The arguments "`--merge --keepmerged`" are necessary to perform whole-program merging. The "`--update-version=7416`" argument is supplied to indicate the initial version(7.4.16) of the updateable program. The "`--base-version`" argument indicates that this is the original program that will be started for the first time; future versions will be dynamically applied. The "LD" variable, indicating the linker used, is redefined similar to "CC", but does not need the "`--base-version`" argument. The "AR" variable, indicating the library archiving utility used, is redefined in a similar way to function in a wrapper mode. Finally, other variables, such as "RANLIB" indicating the tool used to generate an archive index, and "LORDER" indicating the tool used to list dependencies of object files, are also modified. These variables are defining tools that are taking as input binary `.o` object files, and are redefined to not attempt to process binary input.

In this example, whole-program merging produces the file `postgresql-7.4.16/src/backend/postgres_comb.c` which contains the source code of all linked objects and the final object file is called `postgresql-7.4.16/src/backend/postgres_comb.o`. The final binary file retains its original name `postgresql-7.4.16/src/backend/postgres`.

4.2. Exporting Dynamic Symbols

In Figure 4-2, the "LDFLAGS" variable, indicating the flags passed to the linker, is redefined to pass the "`-Wl,--export-dynamic`" argument, which allows a dynamically loaded shared object (a dynamic software update patch) to use the symbols in the built binary.

4.3. Additional API Calls

Additional API calls are available for special preparation of source code to be updateable. These API calls are commonly used to develop test programs that can demonstrate the capabilities of UpStare, accurately test new features, and conduct experiments. Dynamic software updates of real-world programs do not rely on these API calls.

The API calls require including the file `hcu_headers.h`, and they are:

- `HCU_STATIC_REQUEST_UPDATE_IMMEDIATE()`

Insertion of this function call in a program forces a dynamic software update to begin when this call is issued. This is different than applying a dynamic software update when the `hcuapply` tool is invoked, because there is no guarantee of which code the program may be executing when an update is

requested with `hcuapply`. It is a way of initiating a dynamic software update precisely when a particular piece of code is executing during experimentation.

Additionally, this function call means apply a dynamic software update using the updating model of stack reconstruction. An alternative option is to apply a dynamic software update using the updating model of update on function entry using the `HCU_STATIC_REQUEST_UPDATE_LAZY()` call.

- `HCU_STATIC_REQUEST_UPDATE_LAZY()`

Insertion of this function call in a program means apply a dynamic software update lazily: update datatypes, if possible, and update functions on function entry. Do not apply stack reconstruction.

- `HCU_STATIC_UPDATE_FILE(filename, version)`

If the dynamic software update will be initiated with any of the `HCU_STATIC_REQUEST_UPDATE_*` calls, this call is used to inform the dynamic software update runtime of the filename the dynamic software update patch is stored in, and the update version.

- `HCU_MANUAL_UPDATE_POINT()`

This function call manually inserts an update point. This is not necessarily a point where an update will begin.

Figure 4-3 shows an example using these additional API calls to test if dynamic software updates are refused if a signal handler is executing. The `HCU_STATIC_UPDATE_FILE()` macro is used in the top of the program to define the name of the dynamic software update patch file.

`HCU_STATIC_REQUEST_UPDATE_IMMEDIATE()` is called to request an update inside `functionA()` which is called inside a signal handler. `HCU_MANUAL_UPDATE_POINT()` is placed right after this call to ensure an update point will be encountered and the dynamic software update will be attempted. In this example, the runtime detects the request for an update is issued while a signal handler is executing and refuses to apply the update at that point. Of course, the update request will be served as soon as the signal handler exits. But in this example no other update point will be encountered and the update will not be applied at all.

Figure 4-3. Test program that attempts to be updated inside a signal handler.

```
#include <stdio.h>
#include <signal.h>
#include "hcu_headers.h"
HCU_STATIC_UPDATE_FILE("./libsignal_handler_v1.c_v1_to_v2.hcupatch.so.2", 2);

void functionA()
```

```

{
    printf("functionA_v1 entered\n");
    HCU_STATIC_REQUEST_UPDATE_IMMEDIATE();
    HCU_MANUAL_UPDATE_POINT();
    printf("functionA_v1 exited\n");
}

void signal_handler(int s)
{
    printf("signal_handler_v1 executed\n");
    functionA();
}

int main()
{
    if (signal(SIGUSR1, signal_handler) == SIG_ERR) {
        perror("There was an error setting the signal handler:");
        exit(-1);
    }
    kill(getpid(), SIGUSR1);
    printf("Signal handler test exiting\n");

    return 0;
}

```

4.4. Understanding The Compiler And Its Options

The compiler of updateable programs `hcucc.pl` is a wrapper that invokes the compiler driver `hcubasic.pl` with various options. By default, `hcucc.pl` enables all of the features needed to build updateable programs. However it is possible to invoke the compiler driver by enabling only some of these features, which is useful in preparing updateable programs with different capabilities. Enabling only some features can be useful in measuring the performance of updateable programs.

Tip: Note that a lot of the compiler features offer additional options. For a complete list of compiler options run `hcubasic.pl --help`.

The mandatory compiler features that must be enabled to prepare updateable programs are:

- **Enabling common functionality for updating**, enabled with the parameter `--doCommonUpdate`.
- **Initializing the dynamic update runtime environment**, enabled with the parameter `--doInitializeRuntime`.
- **Automatically inserting update points**, enabled with the parameter `--doInsertUpdatePoints`.

- **Identifying thread entry-points, signal-handlers, and functions passed as parameters to libraries**, which need special support to be updated properly, enabled with the parameter `--doWrapCalls`.

In addition to the mandatory compiler features, the following features are enabled by default. These features are optional:

- **Converting blocking system calls to non-blocking**, enabled with the parameter `--doBlockingSystemCallConversion`.
- **Supporting updates of multi-threaded and multi-process programs**, enabled with the parameter `--doMultiThreadedUpdates`.
- **Preparing programs to be updateable using stack reconstruction**, enabled with the parameter `--doStackReconstruction`.
- **Applying dynamic stack tracing** to enforce runtime safety constraints, enabled with the parameter `--doDynamicSafety`.

Other optional features useful in debugging the compiler or further directing the compiler are:

- **Marking some functions as non-updateable**, enabled with the parameter `--doConfiguration --compiler-configuration-file=<filename>`. It requires the compiler configuration file supplied to define a variable of the datatype:
 - `hcu_compiler_configuration_t`

This datatype defines a structure with one variable: an array of function names that should not be instrumented for stack reconstruction.

Figure 4-4 shows an example compiler configuration file.

Figure 4-4. Describing a compiler configuration.

```
#include "hcu_compiler_configuration.h"

hcu_compiler_configuration_t compiler_configuration = {

    /* non_updateable_functions */
    {
        "cash_cmp",
        "on_exit_reset",
        "smgr_redo",
        "b64_dec_len",
        ...
        "SetDefaultClientEncoding",
        "Int_yy_init_buffer",
    }
}
```

```
"restriction_is_or_clause"  
}  
};
```

Warning

Marking some functions as non-updateable can lead to breaking stack-reconstruction if a callee function returns to a non-updateable function. This feature should only be used with a clear understanding of its implications.

- **Trace program execution** at the source-code level, enabled with the parameter `--doExecutionTrace`. This option can be useful in debugging the compiler.
- **Print a control-flow graph** in dot (Graphviz) format, enabled with the parameter `--doPrintDot`. This option can be useful in understanding the control flow of a program.

Chapter 5. Preparing Dynamic Software Updates

Preparing dynamic software updates requires the complete source code of the original, updateable program, and the complete source code of the newer version of program. Using the original and new versions, a dynamic software update patch can be prepared using the `hcu_build_patch.sh` patch generator.

The next sections describe in detail how a dynamic software update patch can be prepared. Both the original and newer source code of a program need to be first compiled as if they were being prepared to be updateable (Section 5.1, Section 5.2). This will result in producing whole-program merged versions of the source code, which are needed by the patch generator. A file describing possible execution continuation mappings (Section 5.3.3) from the old version to the new version must also be prepared by the user. The patch generator (Section 5.4) produces the dynamic software update patch in source code, and the patch is then compiled (Section 5.5) to a binary dynamic software update patch as dynamically loadable shared object.

5.1. Preparing An Updateable Original Version

The original version of the program is prepared as described in Chapter 4. In this example, preparing an updateable vsFTPd version 2.0.4 produces the file `vsftpd-2.0.4/vsftpd_comb.c`.

5.2. Preparing An Updateable New Version

The new version of the program is also prepared as described in Chapter 4. This updateable version of the program will never be run. The program must be prepared as updateable to produce the whole-program merged source code of the program, which will be supplied to the patch generator. In this example, preparing an updateable vsFTPd version 2.0.5 produces the file `vsftpd-2.0.5/vsftpd_comb.c`.

5.3. Describing Dynamic Software Updates

For every dynamic software update patch that will be produced, a file must be provided that describes how the update should be applied. This file describes:

- Which functions will be updated.
- Which global variables and which datatypes will be updated. Note that a datatype update affects functions that use that datatype.
- How program execution should continue after an update is applied.

- Runtime update constraints.

These four items are described next.

5.3.1. Describing Function Updates

The functions that should be updated need to be described to the dynamic software update runtime. Figure 5-1 shows an example describing the updated functions when updating vsFTPD from version 2.0.4 to version 2.0.5.

Figure 5-1. Describing function updates for vsFTPD from 2.0.4 to 2.0.5.

```
#include "hcu_mappings.h"

hcu_mapping_update_description_t mapping_updates_v2[] = { { "main", "main" } };
hcu_mapping_function_update_description_t mapping_function_updates_v2[] = {
    { "str_locate_text_reverse", "str_locate_text_reverse", 0 },
    { "emit_greeting", "emit_greeting", 0 },
    { "handle_login", "handle_login", 0 },
    { "str_locate_chars", "str_locate_chars", 0 },
    { "vsf_privop_do_login", "vsf_privop_do_login", 0 },
    { "vsf_remove_uwtmp", "vsf_remove_uwtmp", 0 },
    { "handle_retr", "handle_retr", 0 },
    { "vsf_sysutil_connect_timeout", "vsf_sysutil_connect_timeout", 0 },
    { "handle_upload_common", "handle_upload_common", 0 },
    { "handle_user_command", "handle_user_command", 0 },
    { "main", "main", 1 },
    { "handle_mdtm", "handle_mdtm", 0 },
    { "handle_size", "handle_size", 0 },
    { "vsf_insert_uwtmp", "vsf_insert_uwtmp", 0 },
    { "handle_feat", "handle_feat", 0 },
    { "str_locate_text", "str_locate_text", 0 },
    { "get_unique_filename", "get_unique_filename", 0 },
    { "vsf_sysutil_chroot", "vsf_sysutil_chroot", 0 },
    { "vsf_sysdep_check_auth", "vsf_sysdep_check_auth", 0 },
    { "vsf_sysutil_tzset", "vsf_sysutil_tzset", 0 },
    { "handle_pass_command", "handle_pas_command", 0 },
    { "vsf_ls_populate_dir_list", "vsf_ls_populate_dir_list", 0 },
    { "calc_num_send", "calc_num_send", 0 },
    { "handle_stat", "handle_stat", 0 },
    { "vsf_privop_do_file_chown", "vsf_privop_do_file_chown", 0 }
};
```

The description file defines two variables of two key datatypes:

- `hcu_mapping_update_description_t`

This datatype defines an array of threads that should be updated. One variable declaration of this datatype is required.

Since vsFTPD is a single-threaded program, this array contains only one entry. The entry requests that the thread called `main` (the thread whose entrypoint function is the function called `main()`) will have its stack reconstructed, when updated, all the way up to the function called `main()`.

Tip: It would be possible to request for this thread to be have its stack partially reconstructed: to unwind up to one of the callees of the `main()` function instead of unwinding all the way up to the `main()` function. This could be useful as an optimization that minimizes the updating latency in a deeply recursive program. But for most programs unwinding the stack up to the thread entrypoint would have little impact in the total updating latency.

- `hcu_mapping_function_update_description_t`

This datatype defines an array of functions that should be updated. One variable declaration of this datatype is required.

Each definition of a function update consists of three fields. The name of the function that will be updated (from the original source code), the name of the function that will take its place (from the new source code), and a flag indicating whether the original function was a thread entrypoint.

Thread entrypoints are the `main()` function, functions that are passed as arguments to a `pthread_create()`, and signal handlers defined with `signal()` and `sigaction()`.

Since vsFTPD is a single-threaded program, the entry to update the `main()` function is flagged as a thread entrypoint.

Warning

But wait! How did a user produce the list of function updates?

The patch generator, described in Section 5.4, can produce the list of modified functions when invoked with empty variable definitions of the two required datatypes `hcu_mapping_update_description_t` and `hcu_mapping_function_update_description_t`. It is expected that a user will first run the patch generator to produce the list of function updates, and then manually produce the update description file.

But why? Isn't the patch generator capable of producing the entire update description file?

The patch generator can produce the entire update description file, but that would be presumptuous. Producing the entire list of function updates in the file would guarantee that a program is *representation consistent*: the running version matches the source code. However, the user may not desire an update to be representation consistent for various reasons. A user may want to apply an update to only a small collection of functions. For example, the user may want to apply a security fix or to avoid introducing, as part of the update, additional known defects that are present in the updated version of the program.

Conclusion: Allowing users to manually produce a customized update description file separates policy from mechanism in the patch generator.

There are plans to enhance the patch generator to produce a template update description file that the user may customize to produce the final file.

5.3.2. Describing Datatype Updates

Datatype updates can affect both global variables and local variables in functions that use the datatype. Datatype updates are automatically produced by the patch generator as described in Section 5.4. To complement incomplete datatype updates a user can manually write datatype transformers in the mappings file. Datatype transformer names must contain the special prefix `HCU_datatype_transformation__` and to be invoked from the special function `HCU_datatype_transformations_function_`.

For example, the `parseconf_uint_array` variable is an array that has had its size extended in the newer version 2.0.5. The values of this array are preserved in the updated `parseconf_uint_array_v205` variable for version 2.0.5, as shown in Figure 5-2.

Figure 5-2. Transforming datatypes for vsFTPD from 2.0.4 to 2.0.5.

```

void HCU_datatype_transformation__struct__parseconf_uint_setting__arraysize17_to_struct__pa
{ long array_counter ;

    {
    array_counter = 0;
    while (1) {
        HCU_datatype_transformation__struct__parseconf_uint_setting__arraysize17_to_struct__pa

        if (array_counter >= 17 - 1) {
            break;
        }
        array_counter ++;
    }

    // Must extend the array
    // NOTE: The following 11 statements are not automatically generated yet
    ((*new)[16]).p_setting_name = malloc(strlen("delay_failed_login") + 1);
    memcpy(&((*new)[16]).p_setting_name, "delay_failed_login" "\0", strlen("delay_failed_login"));
    ((*new)[16]).p_variable = &tunable_delay_failed_login;

    ((*new)[17]).p_setting_name = malloc(strlen("delay_successful_login") + 1);
    memcpy(&((*new)[17]).p_setting_name, "delay_successful_login" "\0", strlen("delay_successful_login"));
    ((*new)[17]).p_variable = &tunable_delay_successful_login;

    ((*new)[18]).p_setting_name = malloc(strlen("max_login_fails") + 1);
    memcpy(&((*new)[18]).p_setting_name, "max_login_fails" "\0", strlen("max_login_fails"));
    ((*new)[18]).p_variable = &tunable_max_login_fails;

    ((*new)[19]).p_setting_name = 0;
    ((*new)[19]).p_variable = 0;

    }
    }

int HCU_datatype_transformations_function() __attribute__((__HCU_ATTRIBUTE_NON_UPDATEABLE))
{
    HCU_datatype_transformation__struct__parseconf_uint_setting__arraysize17_to_struct__pa

    // NOTE: The following 3 statements are not automatically generated yet
    tunable_delay_failed_login = 1;
    tunable_delay_successful_login = 0;
    tunable_max_login_fails = 3;

    printf("HCU_datatype_transformations_function_v205 executed\n");
    return (0);
}

```

5.3.3. Describing Execution Continuation

UpStare is able to update a running algorithm midstream its execution and resume from a different point (not necessarily the beginning) of another algorithm that is behaviorally equivalent: an algorithm that aims to produce the same results while reusing existing progress. This capability requires user assistance. A user needs to define the mapping of continuation points in the original program to continuation points in the new version.

Continuation points are uniquely identified with an integer enumeration starting from 0 for every program function. This enumeration is embedded in a `.cil.c` file which contains the updateable source code. The continuation points of each function are enumerated in the beginning of the function (though the function name now includes the postfix `_vXXX`, where `XXX` is the version number) as `case` statements in a big `switch` statement.

Tip: There are plans to improve the identification (not the selection) of continuation points to use strings rather than numeric ids. This will further minimize the input needed by a user in defining continuation mappings.

For example, for vsFTPD 2.0.4 the update source code is prepared in `vsftpd-2.0.4/vsftpd.cil.c`. To view the continuation points of function `vsf_standalone_main`, a user should look at the big `switch` statement in the beginning of function `vsf_standalone_main_v204`, which is shown in part in Figure 5-3.

Figure 5-3. Viewing the execution continuation points of `vsf_standalone_main` in vsFTPD 2.0.4.

```
struct vsf_client_launch vsf_standalone_main_v204(void)
{
    ...

    switch (__cil_tmp20) {
    case 0:
        goto vsf_standalone_main_entrypoint;
    case 1:
        goto hcu_try_to_update_1_after;
    case 2:
        goto vsf_sysutil_get_ipaddr_size_2_after;
    case 3:
        goto die_3_after;
    case 4:
        goto vsf_sysutil_fork_4_after;
    ...
    }
```

Figure 5-4 shows an example describing execution continuation when updating Bubblesort and continuing execution with Heapsort.

Figure 5-4. Describing execution continuation when updating from Bubblesort to Heapsort.

```
#include "hcu_mappings.h"
#include "hcu_headers.h"

hcu_mapping_update_description_t mapping_updates_v2[] = { { "main", "main" } };
hcu_mapping_function_update_description_t mapping_function_updates_v2[] = {
    { "main", "main", 1 }
};
hcu_mapping_algorithmic_equivalence_t mapping_equivalence_v2_bubblesort[] = {
    { "bubbleSort",
      "heapSort",
      1,
      { { 2, 1 } }
    }
};

struct hcu_stack_local_bubbleSort_v1_s {
    int i ;
    int j ;
    int temp ;
    struct hcu_stack_frame_fields_s hcu_stack_frame_fields ;
};

struct hcu_stack_local_heapSort_v2_s {
    int i ;
    int temp ;
    struct hcu_stack_frame_fields_s hcu_stack_frame_fields ;
};

struct hcu_function_formal_heapSort_v2_s {
    int *numbers ;
    int array_size ;
};

void HCU_stack_transformer__heapSort(void *transform_stack_to ,
                                     void *transform_stack_from ,
                                     void *transform_params_to )
{
    struct hcu_stack_local_heapSort_v2_s *stack_to ;
    struct hcu_stack_local_bubbleSort_v1_s *stack_from ;
    struct hcu_function_formal_heapSort_v2_s *params_to ;

    stack_to = (struct hcu_stack_local_heapSort_v2_s *)transform_stack_to;
    stack_from = (struct hcu_stack_local_bubbleSort_v1_s *)transform_stack_from;
    params_to = (struct hcu_function_formal_heapSort_v2_s *)transform_params_to;

    /* Continue the heapsort algorithm from the current iteration (redo
       the last iteration). Don't restart it from scratch.
    */
}
```

```

    Our bubbleSort implementation processes an array from the
    end. Thus we simply have to shrink the array size to define the
    new bounds of the array for heapSort. */
params_to->array_size = stack_from->i + 1;
hcu_copy_stack_frame_fields(&stack_to->hcu_stack_frame_fields,
                           &stack_from->hcu_stack_frame_fields);
}

```

The description file defines a variable of an important datatype:

- **hcu_mapping_algorithmic_equivalence_t**

This datatype defines an array of behaviorally equivalent functions.

In each array element, the first parameter is the name of the function that will be updated, `bubbleSort`. The second parameter is the name of the behaviorally equivalent function from which execution will continue (`heapSort`) when it is time for the stack of the original function (`bubbleSort`) to be reconstructed. The third parameter reports the number of elements in the fourth parameter, which is an array. Each element of this array lists the update point number of the original function (update point 2 from `bubbleSort`) and a continuation point in the new function (`heapSort`) from which execution should resume.

The description file also defines a stack-state transformer that will allow Heapsort to continue execution from where Bubblesort stopped, reusing the current program state. The transformer name must contain the special prefix `HCU_stack_transformer__` and accepts three special parameters:

- **void *transform_stack_to**

A pointer to the stack of the new function `heapSort`.

- **void *transform_stack_from**

A pointer to the stack of the old function `bubbleSort`.

- **void *transform_params_to**

A pointer to a `struct` variable that groups the formal parameters of the new function `heapSort`.

The `struct` definitions for the stack of the old and new functions, and the formal parameters of the new function also need to be defined. These definitions can also be produced by the patch generator, as discussed in Section 5.3.1. The stack-state transformer invokes a special function that preserves bookkeeping information maintained by the runtime:

- `hcu_copy_stack_frame_fields(from, to)`

Preserves the execution continuation point in the stack of the new function. Executing this function within a stack-state transformer is required.

Tip: So what happens in this example?

The stack of the updated function `heapSort` does not have state preserved from the stack of the old function `bubbleSort` at all. The stack of the old function is only consulted to change the formal parameters of the new function, and the stack of the new function remains uninitialized. Essentially, the new function continues execution by taking as input a smaller array of numbers to be sorted: it continues sorting from where Bubblesort stopped.

5.3.4. Describing Update Constraints

Defining update constraints can help reduce the amount of state that needs to be mapped from the old version of an application to the new version. Depending on the updating model used, defining update constraints can also help enforce runtime safety.

If the updating model requested is to apply updates lazily, it is often necessary to define constraints that enforce type-safety and transaction-safety. The lazy updating model is enabled either using the `HCU_STATIC_REQUEST_UPDATE_LAZY()` call or by invoking the tool `hcuapply` with the command-line parameter `--update-model=lazy`.

Tip: Defining update constraints for type-safety is not necessary if the updating model requested is to apply updates immediately.

However, defining update constraints for transaction-safety can be useful both for applying updates immediately and lazily.

Figure 5-5 shows an example describing update constraints.

Figure 5-5. Describing update constraints.

```
#include "hcu_mappings.h"
#include "hcu_safety_constraints.h"

hcu_safety_constraint_t hcu_safety_constraints_v2[] = {
```

```

{ "main",
  "functionA",
  1,
  { { -1, -1, 0 } }
}
};

```

The description file defines a variable of an important datatype:

- **hcu_safety_constraints_t**

This datatype defines an array of update constraints.

For each array element, the first parameter is the name of thread on which the constraints should be enforced. The second parameter is the name of function for the requested thread on which the constraints should be enforced. The third parameter is the number of elements in the fourth parameter. The fourth parameter is an array of update constraints in disjunctive normal form. For example, the constraint:

$$(a \mid b) \& c$$

should be expressed as:

$$c \& a \mid c \& b$$

This means one should declare an array with two elements. In the first element one should add the two items: $c \ \& \ a$. In the second element one should add the two items: $c \ \& \ b$. For each constraint definition, there are three parameters

- The minimum execution point allowed for a valid update. Execution must have passed this point to match the constraint.

Note that a minimum of -1 is special and means *anywhere* (starting from the beginning of the function).

- The maximum execution point allowed for a valid update. Execution must have not yet encountered this point to match the constraint.

Note that a minimum of -1 is special and means *anywhere* (until the end of the function).

- Whether this constraint is a negation (NOT).

Tip: So what happens in this example?

The constraint prohibits updates anywhere inside the function `functionA` for the `main` thread.

5.4. Running The Patch Generator

The `hcu_build_patch.sh` patch generator is invoked to produce a dynamic software update patch in source code format. The patch generator compares the old and new versions of a program, identifies their differences, and taking into consideration the dynamic update description provided by the user it produces a patch.

Figure 5-6 shows an example of running the patch generator to produce a dynamic software update patch that can update vsFTPD version 2.0.4 to version 2.0.5. The patch generator requires the following parameters:

- The whole-program merged source code of the original version (`vsftpd-2.0.4/vsftpd_comb.c` from Section 5.1).
- The name of the original version (204).
- The whole-program merged source code of the new version (`vsftpd-2.0.5/vsftpd_comb.c` from Section 5.2).
- The name of the new version (205).
- The file describing the update (`2.0.4_to_2.0.5_mappings.c` from Section 5.3.3).
- A `yes/no` flag indicating whether the patch should support dynamic stack tracing.
- A `yes/no` flag indicating whether the patch should support blocking system calls.

Figure 5-6. Preparing a dynamic software update patch for vsFTPD from 2.0.4 to 2.0.5.

```
bash$ hcu_build_patch.sh \
      vsftpd-2.0.4/vsftpd_comb.c 204 \
      vsftpd-2.0.5/vsftpd_comb.c 205 \
      2.0.4_to_2.0.5_mappings.c yes yes
```

The patch generator produces a dynamic software update patch in source code format (`vsftpd_comb.c_v204_to_v205.hcupatch.c`). It also produces a report of differences between the original version 2.0.4 to the new version 2.0.5. Figure 5-7 shows parts of this report for vsFTPD as an example. The report lists the additions and updates of variables and functions.

Figure 5-7. Patch generator report to update vsFTPD from 2.0.4 to 2.0.5.

```
...
variable 's_p_statbuf___6' not found in file 'vsftpd-2.0.4/vsftpd_comb.c.hcudiff.c'. It
```

```

variable 'tunable_delay_successful_login' not found in file 'vsftpd-2.0.4/vsftpd_comb.c.l
variable 'envtz' not found in file 'vsftpd-2.0.4/vsftpd_comb.c.hcudiff.c'. It is an adde
variable 'tunable_max_login_fails' not found in file 'vsftpd-2.0.4/vsftpd_comb.c.hcudiff
variable 'tunable_delay_failed_login' not found in file 'vsftpd-2.0.4/vsftpd_comb.c.hcud
...
variable 'parseconf_uint_array' has HAD its definition updated. It is an updated variabl
...
function 'str_locate_text_reverse' has HAD its definition updated. It is an updated funct
function 'emit_greeting' has HAD its definition updated. It is an updated function.
function 'handle_login' has HAD its definition updated. It is an updated function.
function 'str_locate_chars' has HAD its definition updated. It is an updated function.
function 'vsf_privop_do_login' has HAD its definition updated. It is an updated function
function 'vsf_remove_uwtmp' has HAD its definition updated. It is an updated function.
function 'handle_retr' has HAD its definition updated. It is an updated function.
function 'vsf_sysutil_connect_timeout' has HAD its definition updated. It is an updated f
function 'handle_upload_common' has HAD its definition updated. It is an updated function
function 'handle_user_command' has HAD its definition updated. It is an updated function
function 'main' has HAD its definition updated. It is an updated function.
function 'handle_mdtm' has HAD its definition updated. It is an updated function.
function 'handle_size' has HAD its definition updated. It is an updated function.
function 'vsf_insert_uwtmp' has HAD its definition updated. It is an updated function.
function 'handle_feat' has HAD its definition updated. It is an updated function.
function 'str_locate_text' has HAD its definition updated. It is an updated function.
function 'get_unique_filename' has HAD its definition updated. It is an updated function
function 'vsf_sysutil_chroot' has HAD its definition updated. It is an updated function.
function 'vsf_sysdep_check_auth' has HAD its definition updated. It is an updated functi
function 'vsf_sysutil_tzset' has HAD its definition updated. It is an updated function.
function 'handle_pass_command' has HAD its definition updated. It is an updated function
function 'vsf_ls_populate_dir_list' has HAD its definition updated. It is an updated fun
function 'calc_num_send' has HAD its definition updated. It is an updated function.
function 'handle_stat' has HAD its definition updated. It is an updated function.
function 'vsf_privop_do_file_chown' has HAD its definition updated. It is an updated fun
Diff Report:

```

```

=====

```

Type definitions:

```

-----

```

| | | |
|----------|-----|--------|
| Added: | 0 | 0.00 |
| Deleted: | 0 | 0.00 |
| Updated: | 1 | 0.14 |
| Same: | 693 | 99.86 |
| Total: | 694 | 100.00 |

Variable definitions:

```

-----

```

| | | |
|----------|-----|--------|
| Added: | 5 | 2.14 |
| Deleted: | 0 | 0.00 |
| Updated: | 1 | 0.43 |
| Same: | 228 | 97.44 |
| Total: | 234 | 100.00 |

Function definitions:

```

-----

```

| | | |
|----------|-----|--------|
| Added: | 0 | 0.00 |
| Deleted: | 0 | 0.00 |
| Updated: | 25 | 4.82 |
| Same: | 494 | 95.18 |
| Total: | 519 | 100.00 |

Patch format. The dynamic software update runtime requires that patches contain three special functions. These functions are invoked by the runtime at special points during an update to coordinate its successful application. They are used to apply datatype updates, to apply function updates, and to provide a description of the update. Besides generating these functions, the patch generator also automatically produces datatype and stack-state transformers.

The three special functions expected by the runtime are:

- `int HCU_update_description_function()`

Invoked before the update is attempted to describe to the runtime possible execution continuations and which threads will be updated.

- `int HCU_datatype_transformations_function()`

Invoked after unwinding the old stack and before reconstructing the new stack to transform the datatypes of global variables and to set values for new global variables.

- `int HCU_function_updates_function()`

Invoked after unwinding the old stack and before reconstructing the new stack to update functions to use their new versions.

This capability is used to execute code that will allow updated functions to take control over the program execution of the old version.

These special functions are automatically produced by the patch generator. The functions can be overridden by the user by supplying different definitions in the mappings file (`2.0.4_to_2.0.5_mappings.c`). If alternate definitions are found in the mappings file, then those definitions are used instead of the ones the patch generator would have produced.

There are two reasons a user may want to provide alternate definitions of these functions. First, if the user wants to manually create patches, or produce them with a patch generator the user writes, and generally experiment with the runtime. Second, if it is identified that the patch generator produces incomplete or incorrect definitions.

Tip: An effective practice is to use the patch generator to create a patch, identify which transformers may be incomplete, to complement them, and then to copy them in the mappings file. This new mappings file that is now more complete is then used to run the patch generator again. The goal is to produce a mappings file that fully describes the update, so that generating and applying patches becomes a fully automatic process.

Warning

The patch generator currently does not automatically produce statements that set the values of new variables, such as the values of the variables `tunable_delay_login`, `tunable_delay_successful_login`, and `tunable_max_login_fails` shown in Figure 5-2. There are plans to implement this feature.

Datatype transformers. The patch generator produces transformers that automatically transfer the existing values of old variables to the updated variables. This works well when a `struct` is extended to have an additional field. It also automatically preserves some datatype conversions such as from `int` to `long`. Finally, it automatically preserves arrays that have their size increase by producing a transformer that copies all previous array values to the new array variable.

When the patch generator encounters a datatype transformation it cannot reason about, or a new field which should be initialized by the programmer it reports a warning, as shown in Figure 5-8.

Figure 5-8. Patch generator warnings for vsFTPD from 2.0.4 to 2.0.5.

```
WARNING: HCU_stack_transformer__handle_alarm_timeout():Don't know how to accurately preserve the file
WARNING: HCU_stack_transformer__handle_io():Don't know how to accurately preserve the file
WARNING: HCU_stack_transformer__handle_sigalrm():Don't know how to accurately preserve the file
WARNING: HCU_stack_transformer__handle_sigurg():Don't know how to accurately preserve the file
WARNING: HCU_stack_transformer__handle_upload_common():Don't know how to accurately preserve the file
WARNING: HCU_stack_transformer__handle_upload_common():Unable to match an int definition
WARNING: HCU_stack_transformer__handle_upload_common():Field 'tmp__4' of type 'int ' is a new field
WARNING: HCU_datatype_transformation__struct__vsf_session_v204__to_struct__vsf_session_v204:Field 'tmp__1' of type 'char' is a new field
WARNING: HCU_stack_transformer__vsf_sysdep_check_auth():Field 'retval' of type 'int ' is a new field
WARNING: HCU_stack_transformer__vsf_sysutil_tzset():Field 'tzbuf' of type 'char [sizeof(tzbuf)]' is a new field
WARNING: HCU_stack_transformer__vsf_sysutil_tzset():Field 'tmp' of type 'long ' is a new field
WARNING: HCU_stack_transformer__vsf_sysutil_tzset():Field 'tmp__0' of type 'unsigned int ' is a new field
```

If the user provides custom datatype transformers in the mappings file, the patch generator produces the custom transformers in the patch.

Stack transformers. For functions that are not modified, the dynamic software update runtime automatically preserves their stack with a `memcpy()` call. Thus the patch generator does not produce stack transformers for them. For functions that are not modified but have the datatypes of their local variables updated, the patch generator automatically produces stack transformers that invoke the appropriate datatype transformers.

If the user provides custom stack-state transformers in the mappings file, the patch generator produces the custom transformers in the patch.

5.5. Compiling The Dynamic Software Update Patch

The dynamic software update patch produced by the patch generator, which is in source code format, is compiled using the `hcucc.pl` compiler to produce a dynamic software update patch in binary format, which is a dynamically loadable shared object library. This binary dynamic software update patch is later used to apply a live update.

Figure 5-9 shows an example of preparing a binary dynamic software update patch that will update vsFTPD from version 2.0.4 to version 2.0.5. The compilation is carried out in two steps.

- First, the patch is compiled to also be updateable itself. The `--update-version=205` argument is used to indicate that the patch will update the original version to the new version 2.0.5.

Note that compilation is executed with the `-fPIC` flag which produces position-independent code. Position-independent code is required when building shared object libraries.

The output of this first compilation step is an updateable object file named `vsftpd_comb.c_v204_to_v205.hcupatch.o`.

Tip: To manage the versioning complexity of applying multiple dynamic software updates during the lifetime of an application, compiling the patch to be updateable appends to many datatypes, variables and functions the postfix `_vXXX`, where XXX is the version number. For example, for an update from vsFTPD 2.0.4 to vsFTPD 2.0.5, the compilation appends the postfix `_v205`.

The datatypes, variables and functions that have their names changed are all datatypes, variables and functions that differ between the two versions, all functions defined in the mappings file, and all automatically generated datatype and stack transformers.

- Second, this updatable object file is re-linked to create a dynamically loadable shared object library file. Creating a loadable shared object library file is necessary for the runtime to be able to load the patch into the address space of an already running process using `dlopen()`.

Note that the name of this library is supplied with the `-soname` linker parameter and it is `vsftpd_v205_to_v205.so`. Also note that the linker by default adds the prefix `lib` to all library names passed to it. Hence `lib + vsftpd_v205_to_v205.so = libvsftpd_v205_to_v205.so`.

The output of this re-linking process is the binary dynamic software update patch `libvsftpd_v205_to_v205.so.205`, and this is the file used to apply a live update, as described in Chapter 6.

Figure 5-9. Compiling a dynamic software update patch for vsFTPD from 2.0.4 to 2.0.5.

```
# Compile the source dynamic software update patch to be dynamically updateable
bash$ hcucc.pl -fPIC \
    -c vsftpd_comb.c_v204_to_v205.hcupatch.c \
    -o vsftpd_comb.c_v204_to_v205.hcupatch.o \
    --update-version=205

# Create a binary dynamic software update patch as a dynamically loadable
# shared object library
bash$ hcucc.pl -shared -Wl,-soname=vsftpd_v205_to_v205.so \
    -o libvsftpd_v205_to_v205.so.205 \
    vsftpd_comb.c_v204_to_v205.hcupatch.o
```

Warning

Unlike preparing updateable programs in Section 4.1, no `--base-version` argument is supplied here. The dynamic software update patch is not the original program: it is an update. When the patch is applied the original program is already running.

Chapter 6. Applying Dynamic Software Updates

Applying dynamic software updates can be accomplished by invoking the tool `hcuapply` supplied with a dynamic software update patch, as prepared in Chapter 5.

An example of applying a dynamic software update to vsFTPd from version 2.0.4 to version 2.0.5 is shown in Figure 6-1. The binary dynamic software update patch `libvsftpd_v204_to_205.so`, which was obtained in Section 5.5, is supplied as a parameter to the `hcuapply` tool.

Figure 6-1. Applying a dynamic software update for vsFTPd from 2.0.4 to 2.0.5

```
bash$ hcuapply --file=libvsftpd_v204_to_205.so \  
          --update-version=205
```

By default, the updating model that will be applied during the update is stack reconstruction. A lazy updating model, in which functions are updated on entry, can be requested with the argument `--update-model=lazy`.

An alternative method of applying dynamic software patches is using the API calls `HCU_STATIC_REQUEST_UPDATE_*()`, as described in Section 4.3.

Chapter 7. System Internals

7.1. Function Call Indirection

Function calls are transformed to be executed using the well-known technique of pointer indirection. For each function `f_v1`, a global function pointer variable `f_ptr` is created that originally points to `&f_v1`. Calls to `f_v1` are transformed to calls to `*f_ptr`.

7.2. Thread Entry-Points

If the `main()` function or the start routine passed to a `pthread_create()` attempt to return during reconstruction they will terminate permanently. To allow the update of `main()` or thread entry points, calls to such functions are initiated from a wrapper function.

7.3. Signal Handlers

The address of signal handlers, defined with `sigaction()` and `signal()`, is stored inside the operating system. To avoid resetting signal handlers when they are updated we initiate calls to them from a wrapper function. Additionally, signal handlers interrupt at will the execution of a program at an arbitrary point. They are incompatible with stack reconstruction and we instrument them to raise a flag on entry and reset the flag before exiting. Requests to update are rejected when a program is executing a signal handler. They are immediately satisfied when the program switches again to normal execution mode, and can update the signal handler code at that point.

7.4. Update Points

Update points are automatically inserted in points of execution in the original program that guarantee immediate updates. They are inserted in the beginning of each function (inside the function), and the beginning of each loop (inside the loop). The capability to update, if needed, at each iteration of a long-running loop makes it possible to dynamically update programs from one algorithm to another while taking advantage of the progress of the older algorithm.

Tip: A more aggressive transformation could insert update points in each basic block at the source-code level. However, there would be little benefit in this approach since function calls and loops are encountered sufficiently often to render update points in other basic blocks, like `if-then-else` and `switch` statements, unnecessary.

It is possible to have update points selectively activated or disabled. The application programmer can specify when requesting an update which update points should affect the update (e.g., all except points 250-259 and 262) and this information is stored in the dynamic updating runtime. This empowers the programmer to use the updating mechanism to enforce additional general safety. After an update is applied, all update points are disengaged. The current implementation is restricted to a coarse-grain activation of update points by using a single `must_update` flag, but we plan to support more fine-grain selective activation.

7.5. Exported Local Variables

The `dlopen()` library call successfully loads a dynamic update patch if the patch references only global variables. References to variables that were declared local in the original version (using the `static` keyword) are not accessible after dynamic loading, leading to system exceptions. The UpStare compiler removes the `static` keyword from all local variables and exports them to global.

7.6. Multi-Threaded Updates

The difficulty in updating multi-threaded programs lies in safely coordinating the update timeliness. When datatypes are updated by one thread, one of the remaining threads may attempt to use code that relies on the old representation of the datatype before it encounters an update point. We adapted an algorithm that blocks all threads in heterogeneous checkpointing for multi-threaded applications for dynamic updates. The idea is to force all but one thread to block when the application must update. The one thread that is not blocked will be the coordinator of the update. It polls the status of the remaining threads until it can tell for sure that all threads are blocked, as defined below.

When a thread reaches an update point and the application must update, it raises a flag indicating that it is **willing to cooperate** on the update and then attempts to acquire a **coordination lock**. The first thread to acquire the coordination lock is the **coordinator** of the update. The coordinator can tell that some threads are blocked if their cooperation flags are raised. But this does not cover all threads. Some threads might be blocked waiting on an application lock owned by a thread that is already willing to cooperate and that is blocked on the **coordination lock**. To that end, the system needs to keep track of the blocking status of various threads. Calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()` are replaced with wrapper calls to keep track of the blocking status of threads. When a thread attempts to acquire a lock, it adds the lock to a **WANT** list. When the lock is acquired, the lock is removed from the **WANT** list and placed on a **HAVE** list. When the thread releases the lock, the lock is removed from the **HAVE** list.

The coordinator determines that a thread is **really blocked** if:

- The thread is willing to update;
- The thread is blocked waiting on a lock owned by another thread that is **really blocked**.

The coordinator keeps on checking the status of the other threads until it can determine that all other threads are **really blocked**, at which time the coordinator initiates the actual update: the stack of each

thread is fully unrolled; all datatypes are transformed; the stacks are reconstructed; and, the threads are released to resume executing the updated version.

7.7. Multi-Process Updates

Multi-process applications whose multiple processes communicate with each other, such as processes that use shared memory, signals, or pipes, still need to be updated immediately as a group to guarantee a safe update.

The UpStare compiler automatically transforms calls to `fork()` into calls to the runtime system. The runtime system traces the process hierarchy and when an update must be applied, stack unwinding and reconstruction is coordinated to be atomic among all children, and their threads, of the application. It also wraps calls to `wait()` and `waitpid()` to free memory used for process hierarchy tracing.

Tip: Supporting multi-process updates can be disabled by supplying the `--no-multi-process` argument to the compiler.

7.8. Blocking System Calls

A single threaded or a multi threaded application may have a thread block on a system call, thereby delaying the update. This is particularly problematic for multi-threaded applications since one thread blocking on a system call may indefinitely delay the update of the code of another thread. That's because all threads need to be blocked by our system to update and we cannot tell how long a thread will block on a system call. Examples include waiting to read user input, waiting to receive data from a network socket, or writing to a file on disk. This indefinite blocking possibility exists because the thread waits to acquire a lock, or is put to sleep on a queue inside the operating system kernel. We aim to provide an updating solution that does not rely on the operating system and as such refrain from instrumenting lock acquisition and release inside the kernel.

We automatically transform applications to always issue I/O as a non-blocking operation that allows the runtime system to regain control over execution. Blocking system calls that are handled include `accept()`, `read()`, `recv()` and `select()`. Blocking calls not yet handled include `pselect()`, `recvfrom()`, and `recvmsg()`.

We automatically segment the `sendfile()` operation to smaller chunks to ensure the system call won't block indefinitely. The `write()`, `send()`, and operations are not broken to smaller chunks yet.

Tip: Since there are some blocking system calls that the compiler does not handle yet, the compiler will stop with an error if it encounters such blocking system calls. This behavior can be overridden by supplying the argument `--ignore-unsupported-blocking-system-calls`. Disabling this

safeguard opens the possibility for a program to block indefinitely if an update is requested and the program executes a blocking system call that does not return.

7.9. Dynamic Stack Tracing

Runtime safety checks are enforced by consulting information about the application call sequences (one per thread) and the call site for every call in these sequences. This information is called the context-sensitive call stack information. This information is available at any point during the execution and is maintained using a dynamic stack tracing mechanism. Programs are instrumented to efficiently and dynamically maintain their stack state at a high-level (source-code) and offer this information to the dynamic software updating runtime environment to enforce safety checks before an update is applied. The captured state is architecture (and operating system) independent.

The stack trace dynamically captures the names of functions that are active on the stack. For each function that is active on the stack, the instrumentation also saves the execution point from which the next stack frame was created when the callee function was called. The combination of function names and their execution points provides an accurate context-sensitive call stack trace. The execution points captured are equivalent to the continuation points described in Section 5.3.3. Using this call stack trace, safety checks such as type-safety and transaction-safety can be enforced more accurately. Type-safety can be enforced if type information is precomputed (statically) for every continuation point. Transaction-safety can be enforced if a user forbids updates from being applied inside specific regions of code which are active on the stack.

Warning

The overhead of dynamic stack tracing can be considerable. It is possible to have the overhead be incurred temporarily: from the time an update is requested until the time the update is applied. However, this support has not been implemented yet.