Fluxica Peregrine Reference Manual

Fluxica Computer Engineering
WWW: `www.fluxica.com`
Email: `info@fluxica.com`

This is the Reference Manual of the (embedded) software testing tool Fluxica Peregrine. The Manual relates to the following product versions:

- Fluxica Peregrine ***Demo***
- Fluxica Peregrine ***Fly***
- Fluxica Peregrine ***Soar***
- Fluxica Peregrine ***Stook***

Software Testing Without Bias
Fluxica Computer Engineering

Copyright © 2020-2023 Fluxica Computer Engineering. All rights reserved.

Contents

Disclaimer

The information contained in this document is for informational purposes only and subject to change without notice. Although every precaution has been taken in the preparation of this document, it may contain inaccuracies, omissions and errors. Fluxica Computer Engineering (Fluxica for short) is under no obligation to update or otherwise correct this information. Fluxica makes no representations or warranties with respect to the accuracy or completeness of the contents of this document. Fluxica assumes no liability of any kind, directly or indirectly, in relation to the operation or use of the software and hardware products described in this document. No merchantability or fitness for particular purposes is assumed or implied for these products. No license, directly or indirectly, to any intellectual property rights is granted by this document.

1 System Requirements

This Reference Manual contains the information needed to install and use Fluxica Peregrine. Fluxica Peregrine has the following system requirements:

Hardware Architecture x86-64, AMD64

Operating System Linux-64, Debian (Ubuntu) or Red Hat (Fedora) based

Software C/C++ compiler (`gcc` or `g++`) or Java compiler (JDK with `javac` and `java`)

This manual relates to both C/C++ and Java unless otherwise stated. If a section relates specifically to C/C++ or Java, then this is indicated by the section header (“C/C++ Only” or “Java Only”, respectively).

2 Fluxica Peregrine Testing

2.1 What?

Fluxica Peregrine is a software development tool for **automated in vivo software testing** without bias. In vivo software testing is testing (embedded) computer source code at the execution level, it is the software equivalent of in-circuit hardware testing.

2.2 Why?

The results of the test are used to evaluate software system quality. The evaluation is based on **functionality** (the test reveals unexpected or undesirable behavior) and on **statistics** (the test confirms that the probability is p for unexpected behavior not to occur). The goal of the test is twofold:

1. Increase the fault tolerance of the system.
2. Reduce the time and labor needed for testing the system.

2.3 How?

The source code to be tested can be any code that can be repeatedly executed, from a single statement to an entire program. The developer first adds Peregrine code delimiters around the code segment to be tested and then assigns selected variables in the segment with a matching Peregrine Test Probe. The program including the code delimiters and Peregrine Test Probes is then compiled to an executable file as usual. The Peregrine program will perform and control the test.

After selecting the executable file within the Peregrine program, the file is executed and normal program flow is followed until the code segment is reached. The code segment is then **automatically tested** by the Peregrine program by injecting uniformly distributed pseudo-random (variate) test probes into the segment. These probes are generated on-the-fly at high speed by the Peregrine program. The test continues until one of the following events occurs:

1. A failure is detected.
2. A timeout is triggered.
3. A specified number of test iterations is reached.

In the first two cases, the test is terminated by the Peregrine program. In the latter case, the program under test returns to normal program flow. In all cases, a probe log is archived on-the-fly to provide the test results for post-test analysis.

3 Working Principle

3.1 Smart Fuzz Testing

Fluxica Peregrine is at its heart a smart fuzz tester. Smart fuzz testing is a method to discover unexpected program behavior by injecting random data into the software program under test while matching the shape of the expected input. The smarter a fuzz tester is, the greater the code coverage of the test is.

3.2 Generation Based Testing

A smart fuzz tester is a generation based tester. The input structure of the program under test is analyzed for its basic data type(s) and effective value range(s). Random data is then generated to match type(s) and range(s) and injected into the program to detect potential unexpected program behavior.

3.3 Quantifying Test Uncertainty

The outcome of the test can be used to quantify test uncertainty, which is the probability for unexpected behavior not to occur. Any software program written in C, C++ or Java can be tested with the Peregrine software testing tool.

4 Security

4.1 No Network Connection

Fluxica Peregrine is a stand-alone application program, it runs locally and all functionality is built into the program. The program does not include any network connection functionality, it can not connect over any computer network and thus can not ‘call home’ over the internet.

4.2 Test Results Storage

No information is stored other than the test results contained in the probe log files located in the `$HOME/peregrine/logs/` directory when running the Peregrine program with the GUI. The probe log temporarily residing in memory is shown by clicking the Show Probe Log button. A probe log is stored as a file by clicking the Save Probe Log button and clicking the Remove Probe Logs will permanently remove all probe log files from the `$HOME/peregrine/logs/` directory. Probe logs are plain text files. The file `$HOME/peregrine/.control/.peregrine_log_id.txt` archives the number of probe logs stored so far.

When running the Peregrine program without the GUI, the output of a test is printed to `stdout` and shown in the shell (Linux command line interface). Running the Peregrine program with or without the GUI, the test results are not used by the program.

4.3 License Key Storage

The license key located at `$HOME/peregrine/.control/.peregrine_license_key` includes the type of license and the company name of the company that purchased the key in an encrypted format.

5 Download and Installation

The Peregrine program can be installed manually or by Bash command line script. The latter is discussed in Section 5.3. If the script is used, Section 5.1 and Section 5.2 can be skipped.

5.1 Manual Installation

1. Go to the website `www.fluxica.com`.
2. Go to the **Peregrine** section.
3. Fluxica Peregrine is available in three production versions ***Fly***, ***Soar*** and ***Stook***.

   Note: The ***Demo*** version is available free of charge and able to demonstrate the functionality of the commercial versions, but it runs with restrictions. See also Table 1.

   Each production version has its own `tar.gz` archive file that is available for download after payment.
4. A customer specific license key `.peregrine_license_key` is automatically prepared and included in the archive file before download. The license key is a hidden, plain text file located in the `$HOME/peregrine/.control/` directory after installation. The `.control` directory is a hidden directory.
5. Create the directory `$HOME/peregrine/` and download the selected archive file:
6. Extract the archive file with:

   ```
   $ tar -xvf peregrine_<version>.tar.gz -C $HOME/peregrine/
   ```

   First moving the archive file to the `$HOME/peregrine/` directory and then extracting it with `tar -xvf peregrine_<version>.tar.gz` is also possible.
7. In `$HOME/peregrine/`, assign read, write and execute permission to the file owner of the peregrine program and its GUI program with:

   ```
   $ chmod 700 peregrine
   $ chmod 700 peregrine_gui
   ```
8. The Peregrine program is now installed and ready to be used. For ease of use, it is advised to create the symbolic links discussed in Section 5.2.

	Demo	***Fly***	***Soar***	***Stook***
$N_{T_{max}}$	$1*10^6$	$10*10^6$	$1*10^9$	$1*10^{12}$
$N_{P_{max}}$	1	8	16	32
True random seeding	yes	yes	yes	yes

Table 1: $N_{T_{max}}$ refers to the maximum number of possible test iterations and $N_{P_{max}}$ to the maximum number of test probes that can be logged at any given moment.

5.2 Symbolic Linking

Symbolic links are used to make the Peregrine executable files, header files and library files system wide available.[1] Symbolic links can be defined manually or by Bash command line script. The latter is discussed in Section 5.3.

5.2.1 Executable Files

By creating a symbolic link
`ln -s $HOME/peregrine/peregrine /usr/bin/peregrine` the binary executable file `peregrine` can be executed from anywhere within the associated user account. The same can be done for `peregrine_gui`.[2]

5.2.2 Header File (C/C++ Only)

By creating a symbolic link
`ln -s $HOME/peregrine/c_cpp/peregrine_in_vivo.h /usr/include/peregrine_in_vivo.h`
the header file `peregrine_in_vivo.h` can be included in a C or C++ program using ‘angle brackets’ (i.e., `#include <peregrine_in_vivo.h>`).

5.2.3 Library File (C/C++ Only)

By creating a symbolic link
`ln -s $HOME/peregrine/c_cpp/libperegrine.a /usr/lib/libperegrine.a` the static library file `libperegrine.a` can be linked without having to specify in which directory it resides (i.e., without having to use the `gcc -L` option flag).

5.2.4 Header File (Java Only)

By creating a symbolic link
`ln -s $HOME/peregrine/peregrinejava/PeregrineInVivo.h /usr/include/PeregrineInVivo.h`
the header file `PeregrineInVivo.h` can be included in a Java program using ‘angle brackets’ (i.e., `#include <PeregrineInVivo.h>`).

[1]Rather than creating symbolic links, the files can also be directly copied or moved to the respective standard directories.

[2]Depending on your Linux configuration, it may be necessary to update the `PATH` variable by adding `/usr/bin` to the `/etc/environment` file and then run the `export PATH` command.

5.2.5 Script File (Java Only)

By creating a symbolic link
`ln -s $HOME/peregrine/peregrinejava/cppjava.sh /usr/bin/cppjava` the Bash command line script file `cppjava.sh` can be executed from anywhere within the associated user account.

5.3 Scripted Installation

A Bash command line script `peregrine_auto.sh` that automates the manual installation procedure of Section 5.1 and Section 5.2 is included in the download:

1. Download the `peregrine_<version>.tar.gz archive` file and extract the script file in any directory:

   ```
   $ tar -xf peregrine_<version>.tar.gz ./peregrine_auto.sh
   ```

   If you are using a GUI desktop environment, the script file can also be extracted by double-clicking the archive file, selecting the `support` directory and then drag-dropping the `peregrine_auto.sh` script file to the directory where the archive file is located.

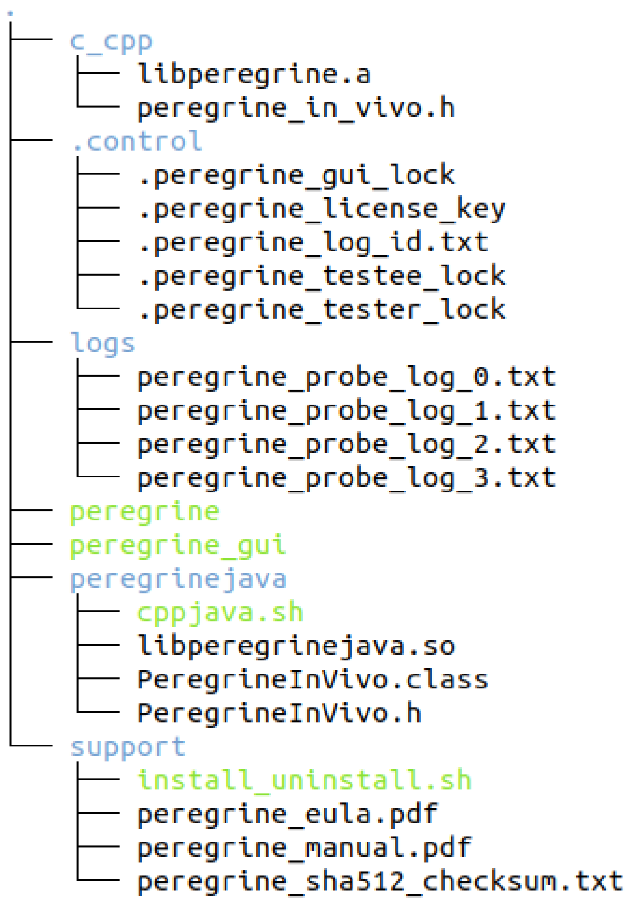
2. To install or uninstall the Peregrine archive file, run the script and follow the instructions:

   ```
   $ ./peregrine_auto.sh
   ```

The symbolic links for the Peregrine executable files, header files and library files are automatically created (install option) and removed (uninstall option) by the script file.

6 File Structure

The Peregrine file structure after installing the archive file is shown in Figure 1.



```
.
├── c_cpp
│   ├── libperegrine.a
│   └── peregrine_in_vivo.h
├── .control
│   ├── .peregrine_gui_lock
│   ├── .peregrine_license_key
│   ├── .peregrine_log_id.txt
│   ├── .peregrine_testee_lock
│   └── .peregrine_tester_lock
├── logs
│   ├── peregrine_probe_log_0.txt
│   ├── peregrine_probe_log_1.txt
│   ├── peregrine_probe_log_2.txt
│   └── peregrine_probe_log_3.txt
├── peregrine
├── peregrine_gui
├── peregrinejava
│   ├── cppjava.sh
│   ├── libperegrinejava.so
│   ├── PeregrineInVivo.class
│   └── PeregrineInVivo.h
└── support
    ├── install_uninstall.sh
    ├── peregrine_eula.pdf
    ├── peregrine_manual.pdf
    └── peregrine_sha512_checksum.txt
```

Figure 1: Peregrine file structure.

Linux uses a file system where everything is a file and if it is not a file, it is a process. A directory is in this context a file listing other files. The Peregrine file structure is summarized below:

- `peregrine`: command line interface (CLI) Peregrine program. The Peregrine program implements testing functionality and runs in parallel with the program under test.
- `peregrine_gui`: graphical user interface (GUI) Peregrine program. The GUI is a decoupled interface to `peregrine`, it does not implement testing functionality and runs in parallel with the program under test and the Peregrine program.

- `c_cpp` directory: contains the interface and library files for C/C++.
  - `libperegrine.a` file: static library file handling interprocess communication between the Peregrine program and the program under test. The lbrary does not implement testing functionality.
  - `peregrine_in_vivo.h`: header file providing a macro interface for accessing Peregrine library functionality.
- `.control` directory: contains the hidden run-time control files required to run the Peregrine program. The files are hidden to emphasize not to change, move or remove them.
  - `.peregrine_license_key`: text file storing a customer specific license key required for using the Peregrine program in a production environment.
  - `.peregrine_log_id.txt`: text file identifying the last probe log stored in the `logs` directory.
  - `.peregrine_tester_lock, .peregrine_gui_lock and .peregrine_testee_lock`: lock files required for synchronizing the parallel execution of the Peregrine program, the GUI and the program under test.
- `logs` directory: location where the probe log files are stored.
  - `peregrine_probe_log_<n>.txt`: text file containing the n-th probe log stored by the user through the GUI, where n is a file counter.
- `peregrinejava` directory: contains the interface and library files for Java.
  - `cppjava.sh`: text file defining the Bash command line script for preprocessing and compiling Java source files.
  - `libperegrinejava.so`: shared library file implementing the same functionality as `libperegrine.a` with Java as target language rather than C/C++.
  - `PeregrineInVivo.class`: class file accessing Peregrine library functionality through the Java Native Interface.
  - `PeregrineInVivo.h`: header file providing a macro interface for accessing `PeregrineInVivo.class`.
- `support` directory: contains support files that are not required to run the Peregrine program.
  - `install_uninstall.sh`: text file defining the Bash command line script for installing and uninstalling the archive file.
  - `peregrine_eula.pdf`: End User License Agreement in Portable Document Format.
  - `peregrine_manual.pdf`: Fluxica Peregrine Reference Manual in Portable Document Format.
  - `peregrine_sha512_checksum.txt`: text file containing the SHA512 checksum for each of the files included in the archive file.

6.1 Dynamic File Creation and Deletion

The `logs` directory, the `.peregrine_log_id.txt` file and the `peregrine_probe_log_<n>.txt` files initially do not exist, they are created by the Peregrine program when needed.

The hidden lock files `.peregrine_tester_lock`, `.peregrine_gui_lock` and `.peregrine_testee_lock` are temporarily stored in the `.control` directory. They are created on program initiation and removed on program termination.

6.2 Manual Lock File Release

In case the program flow of Tester or Testee is terminated prematurely (e.g., with `Ctrl+C`), the Peregrine related interprocess communication (IPC) resources, which includes the lock files, are not released. This will prevent another test to run until these resources are released. If such a situation occurs, the IPC resources can be released by invoking `peregrine` which will then exit with an error message or by invoking `peregrine_gui` immediately followed by closing its window. In both cases, the IPC resources are released.

The lock files can also be manually released by moving to the `.control` directory and removing the respective lock files one by one or all at once:

```
$ cd $HOME/peregrine/.control/
```

followed by

```
$ rm .*lock*
```

7 Test Setup (C/C++ Only)

The following code snippet is used to illustrate how to set up a Peregrine test:

```
int32_t x;

x = input_value();
process_value(x);
```

Setting up the test involves the following four steps:

1. **Include the header file** `peregrine_in_vivo.h`:

   ```
   #include <peregrine_in_vivo.h>

   /* ... */

   int32_t x;

   x = input_value();
   process_value(x);
   ```

2. **Identify the code segment** by adding the two code delimiters `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP`[3] around the segment. The code segment can be anything from a single statement to an entire program:

   ```
   #include <peregrine_in_vivo.h>

   /* ... */

   int32_t x;

   PEREGRINE_RED_CLIP
   x = input_value();
   process_value(x);
   PEREGRINE_BLACK_CLIP
   ```

[3]The delimiters are named after the red and black probe clips in-circuit hardware testing.

3. **Select the test injection point(s)** in the marked out code segment. A potential injection point is any point where variables are assigned or passed. Peregrine Test Probes such as `INT32` (note the capitals) are used to inject a high-speed stream of uniformly distributed pseudo-random (variate) values into the code segment. Variable `x` is selected as an injection point:

```
#include <peregrine_in_vivo.h>

/* ... */

int32_t x;

PEREGRINE_RED_CLIP
x = INT32; /* input_value(); */
process_value(x);
PEREGRINE_BLACK_CLIP
```

4. **Compile and link the program** as usual to create an executable file, but link additionally against the `libperegrine.a` library:

```
$ gcc <your_program_name>.c -lperegrine -o <executable_name>
```

The Peregrine test setup is now prepared and the program is ready for testing. Remove the following Peregrine specific test code artifacts after testing:

1. Header file inclusion `peregrine_in_vivo.h`.
2. Code delimiters `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP`.
3. Peregrine Test Probes such as `INT32`.

7.1 C Versus C++

Other than having to replace `gcc` with `g++` in the last step, the test setup is identical for C and C++. Unlike C, C++ supports function overloading, which is a feature of object oriented programming. C++ uses name mangling to facilitate the overloading feature. Name mangling may result in undefined references when C files are linked to a C++ program. The Peregrine header file uses the preprocessor to detect whether the C or C++ compiler is used and if needed prevents undefined references due to name mangling from occurring.

8 Test Setup (Java Only)

The Peregrine test interface for Java is available as a Java package. As per Java requirement, packages have to be available in the project directory structure. After moving to the project directory, the required directory and its contents can be copied to the current location with:

```
$ cp -r $HOME/peregrine/peregrinejava/ .
```

The Peregrine related code is added to the Java source code using C-styled preprocessor directives. The `PeregrinejavaInVivo.h` header file located in the `peregrinejava` directory acts as an intermediate between the Java source file and the preprocessor. After preprocessing, the Java source file is compiled as usual by the Java compiler.

The following code snippet is used to illustrate how to set up a Peregrine test:

```
/* ... */

public class Testee {
    public static void main(String[] args) {

        int x;

        x = SomeClass.inputValue();
        OtherClass.processValue(x);
    }
}
```

Setting up the test involves the following four steps:

1. **Include the header file** `peregrineInVivo.h`:

```
#include <PeregrineInVivo.h>

/* ... */

public class Testee {
    public static void main(String[] args) {

        long x;

        x = SomeClass.inputValue();
        OtherClass.processValue(x);
    }
}
```

2. **Identify the code segment** by adding the two code delimiters `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP`[4] around the segment. The code segment can be anything from a single statement to an entire program:

```
#include <PeregrineInVivo.h>

/* ... */

public class Testee {
    public static void main(String[] args) {

        long x;

        PEREGRINE_RED_CLIP
        x = SomeClass.inputValue();
        OtherClass.processValue(x);
        PEREGRINE_BLACK_CLIP
    }
}
```

3. **Select the test injection point(s)** in the marked out code segment. A potential injection point is any point where variables are assigned or passed. Peregrine Test Probes such as `LONG` (note the capitals) are used to inject a high-speed stream of uniformly distributed pseudo-random (variate) values into the code segment. Variable `x` is selected as an injection point:

```
#include <PeregrineInVivo.h>

/* ... */

public class Testee {
    public static void main(String[] args) {

        long x;

        PEREGRINE_RED_CLIP
        x = LONG; /* SomeClass.inputValue(); */
        OtherClass.processValue(x);
        PEREGRINE_BLACK_CLIP
    }
}
```

[4]The delimiters are named after the red and black probe clips in-circuit hardware testing.

4. **Preprocess and compile the program** to create the Java class file that can be executed by the Java virtual machine:

   ```
   $ cppjava Testee.java
   ```

   The `JAVAC_CMD` variable in the Bash command line script `cppjava.sh` can be adapted in a text editor to include your usual compiler arguments. The script file is located in the `$HOME/peregrine/peregrinejava` directory.

The Peregrine test setup is now prepared and the program is ready for testing. The location of the `peregrinejava` directory has to be specified when Testee is executed:

```
$ java -Djava.library.path=./peregrinejava Testee
```

Remove the following Peregrine specific test code artifacts after testing:

1. Header file inclusion `PeregrineInVivo.h`.
2. Code delimiters `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP`.
3. Peregrine Test Probes such as `LONG`.

9 Peregrine Test Probes

The Peregrine program controls the test and is referred to as Peregrine **Tester**. The program under test, which has the Peregrine code added to it, is referred to as **Testee**. Communication between Tester and Testee takes place through a shared memory area. Through this memory area, Tester provides test probes that are injected into Testee and continuously monitors the response of Testee. These test probes are generated in real-time by Tester and delivered to Testee as a high-speed stream of independent values of a specific variable type. The speed of this stream is dynamically adjusted to the processing speed of Testee. The test probes are in every practical sense without any bias, most notably without human bias.

A **Peregrine Test Probe** (PTP) is a specific type of variable with a beforehand unknown value that will change to another unknown value after it has been read and before it can be read again. The next value does not depend on one or more previous values. As explained in Section 12, PTP values follow a pseudo-random uniform distribution and are variate values.

9.1 In Vivo Software Testing

PTPs are applied in between the Peregrine code delimiters `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP` where they are assigned to a variable or passed as an argument to a function. Assigning or passing a PTP provides Tester with a direct access **in vivo injection point** into Testee. In vivo software testing is testing "within the living" computer program and monitoring how the program responds to stimuli. Tester provides the stimuli, high-speed streams of variate test probes, and monitors the response of Testee on the stimuli to take test control decisions.

9.2 PTP Nomenclature

The name of a PTP indicates its type and bit length. An `INT16` indicates a 16-bit signed integer, whereas `UINT8` indicates an 8-bit unsigned integer. PTPs with a bit length of less than eight bits such as `UINT2` and `BOOL1` are represented in memory by one byte as the smallest addressable unit of data in C/C++ and Java is a single byte. An overview of the PTPs is shown in Table 2.

9.3 PTP Programming Operators

Any point in between `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP` where variables are assigned or passed can be substituted by a PTP. PTPs can be used in conjunction with the following programming operators: arithmetic, comparison, logical and bitwise. PTPs can not be assigned a value in Testee, PTP values are assigned by Tester outside Testee. Assigning a value to a PTP through the assignment (=), increment (++) or decrement (−−) operator is thus not possible.

PTP Type	Byte count	Bit length	Interval
INT64	8	64	$-2^{63} \ldots 2^{63}-1$
INT32	4	32	$-2^{31} \ldots 2^{31}-1$
INT16	2	16	$-2^{15} \ldots 2^{15}-1$
INT8	1	8	$-2^{7} \ldots 2^{7}-1$
INT4	1	4	$-2^{3} \ldots 2^{3}-1$
INT2	1	2	$-2^{1} \ldots 2^{1}-1$
CHAR8	1	8	$-2^{7} \ldots 2^{7}-1$

PTP Type	Byte count	Bit length	Interval
UINT64	8	64	$0 \ldots 2^{64}-1$
UINT32	4	32	$0 \ldots 2^{32}-1$
UINT16	2	16	$0 \ldots 2^{16}-1$
UINT8	1	8	$0 \ldots 2^{8}-1$
UINT4	1	4	$0 \ldots 2^{4}-1$
UINT2	1	2	$0 \ldots 2^{2}-1$
UINT1	1	1	$0 \ldots 2^{1}-1$
UCHAR8	1	8	$0 \ldots 2^{8}-1$

PTP Type	Byte count	Bit length	Interval
BOOL1	1	1	$0 \ldots 2^{1}-1$

PTP Type	Byte count	Bit count	Range
FLOAT32	4	32	$\approx 1.2 * 10^{-38} \ldots \approx 3.4 * 10^{38}$
DOUBLE64	8	64	$\approx 2.2 * 10^{-308} \ldots \approx 1.8 * 10^{308}$

Table 2: Peregrine Test Probes.

9.4 PTP Combinations

The bit length B_P of a PTP may be shorter, equal or longer than the bit length B_I of the injection point it operates on:

- **Shorter**: the PTP values are uniformly distributed over the B_P least significant bits of the injection point.
- **Equal**: the PTP values are uniformly distributed over all B_I bits of the injection point.
- **Longer**: the PTP values are truncated to their B_I least significant bits and uniformly distributed over all B_I bits of the injection point.

From a performance point of view, applying PTPs with a bit length longer than that of the injection point is not optimal.

Applying PTPs with a bit length equal to that of the injection point is typically useful for black box testing (e.g., the boundary values of the injection point are unknown).

Combining PTPs together with constants and variables where the result has a bit length shorter than that of the injection point allows for defining tests that operate in a specific sub-interval of the injection point.

Examples of combined PTP intervals are:

- An integer in the interval $[-136\ \ldots\ 134]$ is assigned to i:
  `int16_t i = INT8 + INT4;`
- An integer in the interval $[2 * (-2^8)\ \ldots\ 2 * (2^8 - 1)]$ is assigned to i:
  `int16_t i = INT8 + INT8;`
- An ASCII character in the interval $[0\ \ldots\ 127]$ is assigned to c:
  `char c = CHAR8; if (c < 0) c = abs(c + 1);`
- A boolean with a $1/2$ probability for $true$ and otherwise $false$ is assigned to b:
  `bool b = BOOL1;`
- A boolean with a $1/256$ probability for $true$ and otherwise $false$ is assigned to b:
  `bool b = (UINT8==42) ?  true :  false;`
- A float in the interval $[-0.128\ \ldots 0.127]$ is assigned to f:
  `float f = (float)INT8 * 0.001;`
- a double in the interval $[0\ \ldots 1000]$ is assigned to d:
  `double d=fabs(DOUBLE64); while (d>1000) d/=10;`

Examples of combined PTP intervals around a constant or variable are:

- An integer in the interval $[40\ \ldots\ 43]$ is assigned to i:
  `uint_8 i = 42 + INT2;`
- An integer in the interval $[j - 128\ \ldots\ j + 127]$ is assigned to i:
  `int16_t i = j + INT8;`
- An integer in the interval $[0, v]$ is assigned to i:
  `uint16_t i = v * UINT1;`
- An integer in the interval $[-128, 127]$ or $[872, 1127]$ is assigned to i:
  `int16_t i = (BOOL1) ?  INT8 :  1000+INT8;`
- An ASCII character in the interval $[a\ \ldots\ z]$ is assigned to c:
  `char c = 97 + UINT4 + UINT2 + UINT2 + UINT2 + UINT1;`
- An ASCII character in the interval $[A\ \ldots\ Z]$ is assigned to c:
  `char c; do {c=UCHAR8;} while(c<65 || c>90);`
- A double in the interval $[0,\ [const\ \ldots\ const + 0.065535]]$ is assigned to d:
  `double d = (const + (double)UINT16 * 0.000001) * UINT1;`

9.5 Peregrine Probe Types (Java Only)

The PTPs shown in Table 2 are available for both C/C++ and Java. For convenience, the PTPs shown in Table 3 are additionally available for Java. The Java specific PTPs fit the data type naming convention common for Java. Other than their name, the Java PTPs are identical to the C/C++ PTPs referred to as their mirror type in the table.

PTP Type (Java only)	Byte count	Bit length	Interval	Mirror Type
`LONG`	8	64	$-2^{63} \dots 2^{63}-1$	`INT64`
`INT`	4	32	$-2^{31} \dots 2^{31}-1$	`INT64`
`SHORT`	2	16	$-2^{15} \dots 2^{15}-1$	`INT16`
`BYTE`	1	8	$-2^{7} \dots 2^{7}-1$	`INT8`
`CHAR`	1	8	$0 \dots 2^{16}-1$	`UINT16`

PTP Type (Java only)	Byte count	Bit length	Interval	Mirror Type
`BOOLEAN`	1	1	$0 \dots 2^{1}-1$	`BOOL1`

PTP Type (Java only)	Byte count	Bit count	Interval	Mirror Type
`FLOAT`	4	32	$1.2 * 10^{-38} \dots 3.4 * 10^{38}$	`FLOAT32`
`DOUBLE`	8	64	$2.2 * 10^{-308} \dots 1.8 * 10^{308}$	`DOUBLE64`

Table 3: Additional Peregrine Test Probes for Java.

10 Test Execution

The code delimiters `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP` mark out the code segment in Testee that will be probed by Tester. Tester and Testee each run as independent processes. If both Tester and Testee are running, Tester will wait and listen for a PTP request call from Testee. When this call is initiated for the first time by `PEREGRINE_RED_CLIP`, the program flow of Testee is taken over by Tester. Tester then repeatedly executes the code segment and injects a stream of PTPs into Testee. The test continues until one of the following events occurs:

1. A failure is detected.
2. A timeout is triggered.
3. A specified number of test iterations is reached.

The Peregrine Tester program is a headless command line software program, which is particularly convenient for setting up automated batch tests. A desktop GUI is also available for running the Peregrine Tester program. The GUI runs as an independent process and forks two copies of itself that are overlayed with the Tester and Testee program, respectively. Running a test thus consists of at two parallel processes (headless Tester and Testee) or three parallel processes (GUI next to Tester and Testee).

10.1 Headless Testing

The CLI (terminal) command to run Peregrine Tester without a GUI is:

```
$ peregrine <ITERATIONS> <TIMEOUT>
```

The first argument `ITERATIONS` sets the maximum number of test iterations. A single iteration executes the code segment marked out by `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP` one time. This argument is expressed as an integer in the interval $[1 \ldots N_{Tmax}]$, where N_{Tmax} is the maximum number of iterations for a given Peregrine product version as shown in Table 1. If a test reaches the maximum number of test iterations, Testee continuous by executing the code after `PEREGRINE_BLACK_CLIP`.

The second argument `TIMEOUT` sets the maximum time Tester will wait for Testee to complete a single test iteration. The unit of this argument is seconds and it is expressed as an integer in the interval $[1\ \ldots\ 90]$. On timeout, Tester terminates Testee automatically and finalizes the test.

The CLI command either takes no arguments or (i.e., exclusive or) both arguments. Without arguments, the default values for `ITERATIONS` (N_{Tmax}) and `TIMEOUT` (two seconds) are used.

10.1.1 Testee Execution

When Tester is executed by CLI, it does not initiate the execution of Testee. Testee can be invoked in any manner that is convenient (usually in an other terminal). If Testee is running but Tester is not, the code segment between `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP` is passed over by Testee. Tester has to be started before Testee and once Tester is running, it will wait and listen for Testee to reach `PEREGRINE_RED_CLIP` and take control of the test.

10.1.2 PTP Logging

At the end of each test, the last N_P PTPs that have been injected into Testee are outputted to the CLI as a probe log. This output can be redirected to a probe log text file as follows:

```
$ peregrine > peregrine_probe_log.txt
```

The maximum number of PTPs N_{Pmax} that can be logged depends on the product version as shown in Table 1. The number of logged PTPs N_P is less than N_{Pmax} if the test is concluded before probing Testee N_{Pmax} times.

If a PTP (or a combination of PTPs) uncovered faulty or unexpected program behavior, the probe log can be used to analyze the source code of Testee.

During testing, a progress bar is outputted to the CLI indicating the number of test iterations that has been performed as a percentage of the maximum number of possible test iterations. The progress bar and command related user feedback are not redirected to the probe log text file with the command given above, only the probe log itself is redirected.

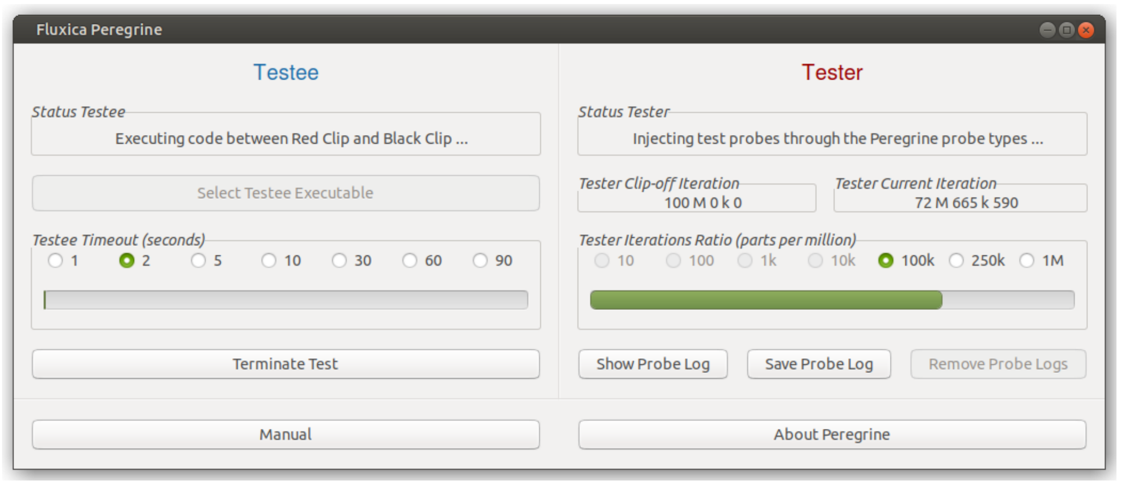
10.2 Desktop GUI Testing

Next to double clicking the desktop icon named `peregrine_gui`, Peregrine Tester can be run with the following CLI command:

```
$ peregrine_gui
```

Figure 2 shows the main window of the GUI. The left side of the GUI interfaces with Testee, whereas the right side interfaces with Tester. Button specific functionality depends on the test state and is controlled by the GUI by activating or de-activating buttons.

The GUI provides two types of actions, test actions (process related) and log actions (archive related). Tester and Testee each have their own progress bar to visualize the progress of the entire test and of a single test iteration, respectively.



Status Testee: Executing code between Red Clip and Black Clip ...

Status Tester: Injecting test probes through the Peregrine probe types ...

Tester Clip-off Iteration: 100 M 0 k 0

Tester Current Iteration: 72 M 665 k 590

Testee Timeout (seconds): 1 [✓ 2] 5 10 30 60 90

Tester Iterations Ratio (parts per million): 10 100 1k 10k [✓ 100k] 250k 1M

Figure 2: Peregrine main window.

10.2.1 Test Actions

The test process can be controlled through the GUI in terms of starting a test, test duration and test termination. The test process related actions are:

- Select Testee Executable
- Testee Timeout
- Tester Iterations Ratio
- Terminate Test

The **Select Testee Executable** button selects the file containing the Testee executable. A Testee executable is either a C/C++ executable or a Java class file. After selecting the executable, Tester automatically starts the test by executing Testee. In case of a Java class file, Tester will invoke the Java virtual machine and run the class file automatically.

The **Testee Timeout** radio button group sets the maximum time Tester will wait for Testee to complete a single test iteration. The timeout can be changed on-the-fly while executing a test. Tester will automatically terminate Testee if the waiting time reaches the selected timeout value. The unit of the timeout setting is seconds in the interval `[1,2,5,10,30,60,90]`.

The **Tester Iterations Ratio** radio button group sets the number of test iterations N_T to perform, it determines how many times the code segment marked out by `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP` is executed. The number of iterations can be changed on-the-fly while executing a test. The buttons in the interval

`[10,100,1k,10k,100k,250k,1M]` represent the number of iterations as a parts per million of the maximum number of iterations N_{Tmax}. Table 1 shows N_{Tmax} for each Peregrine product version.

The number of test iterations to perform is represented in the *Tester Clipp-off Iteration* box, whereas the *Tester Current Iteration* box represents the ongoing test iteration. For convenience, iteration numbers are represented in powers of 10 with metric prefixes, e.g., `987654321987` becomes `987 T 654 M 321 k 987`.

The **Terminate Test** button terminates the test and Testee immediately. Terminating the test means that Testee code that follows `PEREGRINE_BLACK_CLIP` is not executed.

10.2.2 Log Actions

The last N_P PTPs that have been injected into Testee can be saved to a probe log file. The maximum number of PTPs N_{Pmax} that can be logged at any given moment depends on the Peregrine product version as shown in Table 1. The number of logged PTPs N_P is less than N_{Pmax} if the test is finalized before probing Testee N_{Pmax} times.

If a PTP (or a combination of PTPs) uncovered faulty or unexpected program behavior, the probe log file can be used to analyze the source code of Testee. The log archive related actions are:

- Show Probe Log
- Save Probe Log
- Remove Probe Logs

The **Show Probe Log** button displays the last N_P PTPs that have been injected into Testee together with test specifics such as test state and test duration.

The **Save Probe Log** button saves the PTPs listed by Show Probe Log in a text file in the `$HOME/peregrine/logs/` directory with the following naming convention `peregrine_probe_log_<n>.txt` where `n` is a file counter. The file counter is initially set to 0 and automatically increases allowing to save multiple files after each other.

The **Remove Probe Logs** button removes all the probe log files that have been saved previously and resets the file counter to 0.

10.2.3 Progress Bars

To get a quick indication on the progress of a test, the GUI provides two progress bars:

- Test Iterations
- Testee Timeout

The **Test Iterations** progress bar shows the number of test iterations that have been performed in ratio to the number of test iterations N_T to be performed. Changing N_T during the test will automatically adjust the progress bar in ratio. This progress bar depends on the settings provided by the Tester Iterations Ratio radio button group discussed in Section 10.2.1.

The **Testee Timeout** progress bar shows how long Tester has been waiting for Testee in ratio to the timeout setting. The progress bar is reset to 0 for each test iteration, meaning that if a test iteration takes little time (e.g., a few microseconds) the progress bar stays visually on 0 percent or just above 0 percent.

11 Test Evaluation

The results of a Peregrine test can be classified as follows:

1. **Pass**: The PTPs generated expected program behavior.
2. **Fail**: The PTPs generated unexpected program behavior.
3. **Timeout**: The time frame for Testee to respond has passed.

Expected program behavior includes all first order and all second order functionality. First order functionality has been the reason why the program is developed. Second order functionality has not been the reason why the program is developed, but is required to make first order functionality possible. An example of second order functionality is an error handler.

Unexpected program behavior is all program behavior that is not expected (e.g., undesirable, faulty, defective or vulnerable behavior), it is not part of any kind of functionality the program is supposed to have and is classified as a Fail.

A **timeout** is in the first place a lack of response. Tester does not receive a response from Testee in the specified time frame (i.e., Testee has not completed its test iteration in time). A timeout is then triggered and Tester terminates the test and Testee. If the time frame of a timeout is beyond the point of expected program behavior, the timeout can be classified as a Fail.

11.1 Stringent Test Evaluation

Specifying a time frame for the timeout that is considered unexpected program behavior makes a more rigorous or stringent test evaluation possible. A test result is then classified as a Fail unless it is a Pass:

1. **Pass**: The PTPs generated expected program behavior.
2. **Fail**: The PTPs did not generate program behavior classifying as a Pass.

Formally, if and only if it is a Pass then it is not a Fail: $Pass \iff \overline{Fail}$. A stringent evaluation has two main advantages. The classification is always conclusive, a timeout and also unforeseen and misunderstood output behavior classify as a Fail, and is easy to understand and implement.

11.1.1 Faulty Expected Program Behavior

In certain development settings it may occur that faults in the code base are known but not fixed and the resulting faulty program behavior is then explained (away) as expected program behavior. Evaluating a system for faulty expected program behavior should be avoided as it goes against the idea that a system is evaluated for the behavior it should have rather for the behavior it has.

11.2 Test Oracle

A test oracle determines whether a test passes or fails. It takes the input-output values of the system under test and compares this with the input-output values that the oracle determines the system should have. In case of a Pass or Fail, the oracle performs one of the following actions:

1. **Pass**: Do nothing.
2. **Fail**: Broadcast to Tester it is not a Pass by raising `PEREGRINE_ORACLE`.

The Fail related action will cause Tester to immediately end the test and report the probe log. The oracle should not terminate Testee, as test control and thus program control is with Tester.

Defining the oracle before starting to program and iteratively testing each step in the programming process is good engineering practice, it prevents and detects faults right from the beginning. The longer a fault is not detected, the more consequences it will have as the program grows in size and the more effort it will require to correct it.

The following code snippet is prepared for Peregrine testing with an in-program test oracle:

```
uint16_t val1, val2, process_result;

val1 = set_value(val1);
val2 = set_value(val2);
process_result = process_values(val1, val2);
```

The test oracle confirms, by not raising `PEREGRINE_ORACLE`, that the result of the function `process_values` falls within a specified range:

```
#include <peregrine_in_vivo.h>

/* ... */

uint16_t val1, val2, process_result;

PEREGRINE_RED_CLIP

/* val1 = set_value(val1); */
/* val2 = set_value(val2); */
process_result = process_values(UINT16, UINT16);

if (process_result>=0 && process_result<=255) { /* Do nothing. */ }
else PEREGRINE_ORACLE; /* Raise a Fail and immediately terminate the test. */

PEREGRINE_BLACK_CLIP
```

An alternative test oracle can implement a mapping matrix `oracle_matrix` that takes the input values to `process_values` and maps them to the value that `process_result` should have. The oracle compares the input-output combination of the function `process_values` with the input-output combination the function has been designed for:

```
#include <peregrine_in_vivo.h>

/* ... */

uint16_t val1, val2, process_result;

PEREGRINE_RED_CLIP

val1 = UINT16; /* set_value(val1); */
val2 = UINT16; /* set_value(val2); */
process_result = process_values(val1, val2);

if (oracle_matrix(val1,val2)!=process_result) PEREGRINE_ORACLE;

PEREGRINE_BLACK_CLIP
```

As with the previous example, the oracle uses a stringent evaluation by comparing against expected program behavior. The first oracle confirms if a function result is within the expected operating range, whereas the second oracle confirms if a function result matches the expected input-output combination in a one-to-one fashion. The first oracle can not detect unexpected program behavior within the boundaries of expected program behavior, the second oracle can.

This is a limited version of the manual. The full version of the manual is available in the commercial download packages Peregrine Fly, Soar and Stook.

13 Test Performance

If the time required by the Testee process would be reduced to an absolute minimum then the maximum performance of Tester for a given hardware platform can be determined. The performance of Tester has been measured on an x86-64 platform with an Intel E31230 CPU clocked at 3.2 GHz and with 4GB of internal memory running Ubuntu 20.04.5 LTS with Linux kernel version 5.15.0-67-generic. The source code has been compiled with gcc version 9.4.0. The program for measuring the performance for generating `UINT8` PTPs is:

```
#include <peregrine_in_vivo.h>

void main() {
    uint8_t x; /* Run for each PTP type. */

    PEREGRINE_RED_CLIP
    x=UINT8;
    PEREGRINE_BLACK_CLIP
}
```

The time required to generate $1 * 10^9$ PTPs is measured for all PTP types. The maximum performance of Tester depends on the byte count of the applied PTP type. PTPs with a bit length of less than eight bits such as `UINT2` and `BOOL1` are represented in memory by one byte as the smallest addressable unit of data in C/C++ and Java is a single byte. The performance results are shown in Table 5.

PTP Type	#Bytes	#Bits	Time (secs)	Probes/secs	Bits/secs
`INT64`, `UINT64`, `DOUBLE64`	8	64	120	$8.3 * 10^6$	$533.3 * 10^6$
`INT32`, `UINT32`, `FLOAT32`	4	32	99	$10.1 * 10^6$	$323.2 * 10^6$
`INT16`, `UINT16`	2	16	91	$11.0 * 10^6$	$175.8 * 10^6$
`INT8`, `UINT8`, `CHAR8`, `UCHAR8`	1	8	84	$11.9 * 10^6$	$95.2 * 10^6$
`INT4`, `UINT4`	1	4	84	$11.9 * 10^6$	$47.6 * 10^6$
`INT2`, `UINT2`	1	2	84	$11.9 * 10^6$	$23.8 * 10^6$
`UINT1`, `BOOL1`	1	1	84	$11.9 * 10^6$	$11.9 * 10^6$

Table 5: Maximum Tester performance for generating $1 * 10^9$ PTPs of each type for a given hardware platform.

13.1 Time and Space Complexity

The performance of Tester is constant, both time and space complexity are of order $O(1)$. The overall performance of a test is determined by the time and space complexity of the code segment marked out in Testee by `PEREGRINE_RED_CLIP` and `PEREGRINE_BLACK_CLIP`.

14 Example Applications

The concept behind the examples presented in this section apply to both C/C++ and Java. The test setup and the PTP related programming constructs in each of the examples apply in a one-to-one fashion to both C/C++ and Java. An example entirely written in Java is presented in Section 14.6.

14.1 Quick Code-Test-Evaluation Cycles

The rule of thumb presented in Section 12.1.2 is used to demonstrate quick code-test evaluate cycles. The source code of the `is_prime` example is shown below:

```
int prime(uint16_t num, uint16_t i) {
    if (i==1) return 1;
    else {
        if (num % i==0) return 0;
        else return prime(num, i-1);
    }
}

bool is_prime(uint16_t num) {
    if (prime(num, num/2)==1) return true;
    else return false;
}
```

The test driver code `prime.c` prepares for testing with the Peregrine program:

```
#include <peregrine_in_vivo.h>

/* ... */

void main() {
    PEREGRINE_RED_CLIP

    is_prime(UINT16);

    PEREGRINE_BLACK_CLIP
}
```

Program peccability p is set to maximally 2^{-B}, where B is at least the bit length of the PTP. In this case, probability p relates to all possible inputs because the PTP covers the entire bit length of the `is_prime` function parameter. The program is compiled with:

```
$ gcc prime.c -lperegrine -o prime
```

The Peregrine program is run with GUI and after selecting the `prime` executable testing automatically starts. The test triggers a timeout after 2150 iterations. The last PTP injected reads `(UINT16, 0)` in the probe log. The test is run again and after 20340 test iterations a timeout is triggered and the last PTP injected reads `(UINT16, 0)`. Another run triggers the timout at 34118 iterations and the last PTP injected reads `(UINT16, 1)`. This last PTP value is especially odd, as it is a prime number. The test consistently triggers a timeout for the input values 0 and 1.

Analyzing the source code reveals two problems. The modulo operator will cause the program to exit abruptly if its right argument is 0. If `is_prime` is called with 0 or 1 as a function argument, `prime` is called with 0 for its second argument. The divide operator truncates all fractional results towards zero (i.e, `num/2` gives 0 if `num` equals 1).

To account for 0 and 1 as input values, the following two lines are added to `is_prime` immediately at the beginning of its body:

```
if (num==0) return false;
else if (num==1) return true;
```

Compiling and running the test again does not trigger a timeout. The test is terminated by clicking the Terminate Tester button after more than $1 * 10^6$ iterations. The probe log is shown below, the probe values have been omitted:

```
- - - - - - - - - Test Results - - - - - - - - -

Test status: Terminated on request by Engineer

Iterations performed: 1 M 18 k 439
Iterations set to: 10 M 0 k 0
Iterations maximum: 1 G 0 M 0 k 0

Total number of PTP bits injected: 16 M 295 k 8
Combined PTP bit length B per iteration: 16

For the PTP injection points:
    If program peccability p is maximally 1/(2^16)
        then end test reliability t is: → 100 %

    p is the probability for a PTP to uncover unexpected program behavior.
    t is the probability the test is conclusive.

PTPs logged: Last 16
Timeout Testee (seconds): 2
True random seeding: Enabled

Time stamp start test: Thu Mar 16 13:16:26 2023
Time stamp end test: Thu Mar 16 13:17:18 2023
Time elapsed (seconds): 41.822050
```

The test confirms whether `is_prime` returns normally, it does not confirm whether its return values are correct. How conclusive the test is can be calculated using the formula presented in Section 12.1. Test reliability t is:

$$\begin{aligned} t &= 1-(1-p)^{N_T} \\ t &= 1-(1-\frac{1}{2^{16}})^{1*10^6} \\ t &\rightarrow 1 \end{aligned}$$

The test is near 100 percent conclusive that `is_prime` returns normally from any function call in any order with an argument in the interval $[0, 1, 2, ..., 2^{16} - 1]$. As shown in Section 12.2, $302 * 10^3$ test iterations would have been enough for a 99 percent conclusive test. Such a test would take approximately 13 seconds to execute.

14.2 Stress Test

A server side program called `license_server` generates license keys for software products that are ordered online. The program has three parameters:

- Version: the version ordered (four possible versions in the interval $[1, 2, 3, 4]$).
- Nodes: the number of copies ordered (maximally 16 copies in the interval $[1, 2, 3, ..., 16]$).
- Name: the name of the customer (maximally 32 characters in the interval $[4, 5, 6, ..., 32]$).

A stress test is used to determine how many keys can be generated on a given hardware platform per second. The test is defined as a C program around a system call to `license_server`. The parameters of the call receive PTPs as their arguments:

```
#include <peregrine_in_vivo.h>

/* ... */

int main() {

    //    Determining maximum nr. of system calls to license_sever per second.
    //    The hardware platform is a given and the input per call is changing.

    char version[2],nodes[3],name[40],call[100],c;
    int name_len=0;

    PEREGRINE_RED_CLIP

    //    Generate random version number ranging from 1 through 4.
    //    Generate random nodes number ranging from 1 through 16.

    sprintf(version,"%d",UINT2+1);        /* [1,2,3,4] */
    sprintf(nodes,"%d",UINT4+1);         /* [1,2,3,...,16] */

    //    Generate random name of minimally 4 and maximally 32 characters.

    do { name_len=2+UINT4+UINT4;         /* [4,5,6,...,32] */
    } while(name_len<4);

    for(int i=0;i<name_len;i++) {
        do { c=UCHAR8;
        } while(c<97 || c>122);           /* a,b,c,...,z */
        name[i]=c;
    } name[name_len]='\0';

    //    Prepare system call string.
    //    Execute license server with an OS level system call.
    sprintf(call, "%s %s %s %s", "./license_server",version,nodes,name);
    system(call);

    PEREGRINE_BLACK_CLIP

    return 0;
}
```

14.2.1 Test Execution

The program source code above is called `stress_test.c` and compiled as a stand-alone test program:

```
$ gcc stress_test.c -lperegrine -o stress_license_server
```

In one terminal, the Peregrine program (Tester) is started ($1 * 10^6$ iterations and a timeout of 2 seconds). The test results are redirected to a text file:

```
$ peregrine 1000000 2 > stress_log.txt
```

In an other terminal, the stress test setup (Testee) is started. Run time messages from `license_server` are redirected to a text file:

```
$ ./stress_license_server > license_server_output.txt
```

The probe log is available in the `stress_log.txt` file after completing the test, the probe values have been omitted:

```
- - - - - - - - - Test Results - - - - - - - - -

Test status: Finished without raising Fail

Iterations performed: 1 M 0 k 0
Iterations set to: 1 M 0 k 0
Iterations maximum: 1 G 0 M 0 k 0

Total number of PTP bits injected: 1 G 366 M 337 k 872
Combined PTP bit length B per iteration: 1367

Program peccability and test reliability: Not calculated (B > 40)

PTPs logged: Last 16
Timeout Testee (seconds): 2
True random seeding: Enabled

Time stamp start test: Thu Mar 16 17:09:21 2023
Time stamp end test: Thu Mar 16 17:28:23 2023
Time elapsed (seconds): 1142.725532
```

Given the hardware platform, the maximum number of `license_server` calls that can be handled is approximately $1 * 10^6/1143 \approx 875$ per second.

14.2.2 Null Test

A null test reveals how much execution time is consumed by just `license_server`, thus without the time needed for running the test setup itself, the operating system and similar overhead. The test is identical to the stress test in Section 14.2 with the only difference being that the system call to `license_server` is commented out in the second test. The output of the second test is compared to the first test:

```
- - - - - - - - - - Test Results - - - - - - - - - -

Test status: Finished without raising Fail

Iterations performed: 1 M 0 k 0
Iterations set to: 1 M 0 k 0
Iterations maximum: 1 G 0 M 0 k 0

Total number of PTP bits injected: 1 G 366 M 795 k 928
Combined PTP bit length B per iteration: 1367

Program peccability and test reliability: Not calculated (B > 40)

PTPs logged: Last 16
Timeout Testee (seconds): 2
True random seeding: Enabled

Time stamp start test: Thu Mar 16 17:36:44 2023
Time stamp end test: Thu Mar 16 17:36:58 2023
Time elapsed (seconds): 14.142616
```

The test execution time is 14.1 seconds, amounting for the total test overhead execution time required to execute `license_server` $1 * 10^6$ times. The null test shows that preparing the PTPs for a test iteration takes approximately 1.2 percent of the total test execution time, meaning that the stress test performance is primarily determined by the characteristics of `license_server` and not by the test infrastructure.

14.2.3 Error Injection

Artificial faults can be introduced in the argument values of `license_server` to confirm whether faults are properly caught and handled. The source code of `stress_test.c` is adapted to generate input errors that occur approximately once in 2^{16} test iterations: <See next page.>

```
#include <peregrine_in_vivo.h>

/* ... */

int main() {
    char version[2],nodes[3],name[40],call[100],c;
    int version_err=0, nodes_err=0, name_err=0, name_len=0;

    PEREGRINE_RED_CLIP

    //    Generate random version number ranging from 1 through 4.
    //    Generate random nodes number ranging from 1 through 16.

    sprintf(version,"%d",UINT2+1);        /* [1,2,3,4] */
    sprintf(nodes,"%d",UINT4+1);        /* [1,2,3,...,16] */

    //    Generate random name of minimally 4 and maximally 32 characters.

    do { name_len=2+UINT4+UINT4;        /* [4,5,6,...,32] */
    } while(name_len<4);

    for(int i=0;i<name_len;i++) {
        do { c=UCHAR8;
        } while(c<97 || c>122);            /* a,b,c,...,z */
        name[i]=c;
    } name[name_len]='\0';

    //    Generate error approx. 1 in 2^16 iterations for version.

    if (UINT16==65535) {
        if (BOOL1) sprintf(version,"%d",0);
        else sprintf(version,"%d",5);
        version_err++;
    }

    //    Generate error approx. 1 in 2^16 iterations for nodes.

    if (UINT16==65535) {
        if (BOOL1) sprintf(nodes,"%d",0);
        else sprintf(nodes,"%d",17);
        nodes_err++;
    }
    //    Generate error approx. 1 in 2^16 iterations for name.

    if (UINT16==65535) {
        for(int i=0;i<40;i++) name[i]='X';
        if (BOOL1) name[3]='\0';
        else name[33]='\0';
        name_err++;
     }

    //    Prepare system call string.
    //    Execute license server with an OS level system call.

    sprintf(call, "%s %s %s %s", "./license_server",version,nodes,name);
    system(call);

    PEREGRINE_BLACK_CLIP

    printf("\n\n#version errors: %d, #nodes errors: %d, #name_errors: %d\n",
            version_err, nodes_err, name_err);

    return 0;
}
```

The number and type of errors caught and handled by `license_server` should match the final values for `version_err`, `nodes_err` and `name_err`. The number of errors caught could be more than indicated by the error counters due to naturally occurring errors, but not less. The test is run with the same settings as before and requires 886 seconds to conclude:<See next page.>

```
- - - - - - - - - Test Results - - - - - - - - -

Test status: Finished without raising Fail

Iterations performed: 1 M 0 k 0
Iterations set to: 1 M 0 k 0
Iterations maximum: 1 G 0 M 0 k 0

Total number of PTP bits injected: 1 G 413 M 874 k 866
Combined PTP bit length B per iteration: 1414

Program peccability and test reliability: Not calculated (B > 40)

PTPs logged: Last 16
Timeout Testee (seconds): 2
True random seeding: Enabled

Time stamp start test: Thu Mar 16 18:05:30 2023
Time stamp end test: Thu Mar 16 18:25:23 2023
Time elapsed (seconds): 1192.936482
```

The final value of the error counters is outputted in the Testee terminal:

```
Errors generated: #version errors: 16, #nodes errors: 18, #name_errors: 18
```

The terminal output of `license_server` is available in `license_server_output.txt`. The implementation of `license_server` contains one or more faults if the number of errors caught and handled by `license_server` for a specific input parameter (e.g., `version`) is less than its associated error counter (e.g., `version_err`). If the number of errors caught and handled is equal to the error counter, the implementation is probably as expected. If the number is greater, the implementation probably contains one or more faults.

14.3 Black Box Test

A thermostat is software controlled and designed for a field (user) setting. The software targets a microcontroller and has the following function declaration at its core:

```
void run_thermostat(uint8_t, uint8_t, int8_t);
```

The three input parameters have a combined bit length of maximally 24 bits. The result of `run_thermostat` is written to two 8-bit integers of unsigned type, providing a combined bit length of maximally 16 output bits. Nothing else is known or assumed about the internal workings of this function, it is a black box test.

14.3.1 Oracle

An oracle monitors the input-output combination of this function and compares it to the required input-output combination. The comparison is based on a mapping matrix where each input variable is divided into intervals that map to the required output intervals. If an input-output combination is covered by the matrix, then the oracle returns a Pass (`true`). In all other cases it returns a Fail (`false`) and `PEREGRINE_ORACLE` is raised. The oracle has access to the memory locations where `run_thermostat` stores it results (through `&mem_1` and `&mem_2`, respectively).

14.3.2 Test Setup

A while loop is used to continuously execute `run_thermostat` on the target microcontroller platform:

```
while(1) run_thermostat(get_set_temp(), get_event(), get_ambient_temp());
```

A trial run on the target hardware reveals that it takes approximately 2.5 milliseconds for `run_thermostat` to complete a call. The program source code is prepared for testing:

```
#include <peregrine_in_vivo.h>

uint8_t set_temp, event;
int8_t ambient_temp;

/* while(1) */

PEREGRINE_RED_CLIP

set_temp=UINT8; event=UINT8; ambient_temp=INT8;
run_thermostat(set_temp, event, ambient_temp);
if (!oracle(set_temp, event, ambient_temp, &mem_1, &mem_2)) PEREGRINE_ORACLE;

PEREGRINE_BLACK_CLIP
```

14.3.3 Test Reliability vs. Test Time

Test reliability t should reach at least 99.99 percent. The combined bit length of the input parameters is 24 bits and is used in its entirety as a PTP injection point. Program peccability p is then 1 in 2^{24}, which is the probability for a variate PTP to uncover a fault. The required number of test iterations without the oracle returning a Fail is:

$$\begin{aligned}
N_{min} &= \frac{log(-t+1)}{log(1-p)} \\
N_{min} &= \frac{log(-0.9999+1)}{log(1-\frac{1}{2^{24}})} \\
N_{min} &= 154.5 * 10^6 \approx 155 * 10^6
\end{aligned}$$

Knowing that the microcontroller platform requires approximately 2.5 milliseconds to output a response to its input, performing a test of $155 * 10^6$ iterations will take at least 107 hours or 4.5 days. If a fault is discovered after several days of testing, the test has to be done again. If multiple faults are found this way, the total test time is likely to become impractical and this usually means in a development setting the test will be cut short to meet project deadlines at the expense of not reaching the required test reliability.

14.3.4 Fail Fast: First Simulate Embedded Hardware

A good test fails fast and fails often, it should find faults as quickly as possible. Due to the nature of the development process, embedded hardware usually contains less faults than its associated software. Filtering out faults goes faster when a test iteration takes less time. Trying to trigger software specific faults in a low performance computing environment that embedded hardware often is, is not an efficient test setup. For example, a change in thermostate temperature readout will not change faster than the actual change in ambient temperature.

It is more efficient to identify software related faults on a dedicated computing platform that is focused on delivering CPU cycles. First test the software on a desktop (or server) where the embedded hardware is simulated with a test reliability approaching 100 percent and then test on the target hardware in a real-time setting. This will reduce the total test time and it becomes easier to identify whether a fault originates from the software or the hardware. Additionally, it eliminates the need to create a more complex and thus costly test emulation environment.

If the computing platform described in Section 13 runs the `run_thermostat` function and simulates the hardware responses of the microcontroller platform, $155 * 10^6$ test iterations are reached in less than a minute. The short test time allows for quick code-test-evaluate cycles. Once the test passes, the test is performed on the target hardware.

14.4 Boundary Value Test

A probability analysis application uses Pascal's triangle to determine the coefficients of a binomial of the form $x + y$ when it is raised to a positive integer power n. For $n = 3$, the coefficients in the expansion

$$(x+y)^3 = x^3 + 3x^2y + 3xy^2 + y^3 = \mathbf{1}x^3y^0 + \mathbf{3}x^2y + \mathbf{3}xy^2 + \mathbf{1}x^0y^3$$

are printed in bold. The first five rows of Pascal's Triangle are the following:

$$\begin{array}{lccccccccccc}
n=0: & & & & & & 1 & & & & & \\
n=1: & & & & & 1 & & 1 & & & & \\
n=2: & & & & 1 & & 2 & & 1 & & & \\
n=3: & & & 1 & & 3 & & 3 & & 1 & & \\
n=4: & & 1 & & 4 & & 6 & & 4 & & 1 & \\
n=5: & 1 & & 5 & & 10 & & 10 & & 5 & & 1 \\
 & & & & & & \dots & & & & &
\end{array}$$

The coefficients of the binomial $(x + y)^n$ correspond with the entries of line n in Pascal's triangle. The rows of Pascal's trangle are numbered top to bottom beginning with row $n = 0$. The entries in a row are numbered left to right beginning with $k = 0$. The entry at row 0 is a 1 and each row starts and ends with a 1.

Any other entry of any other row is determined by adding the two numbers above it. For example, the first number 10 in the sixth row is the result of adding numbers 4 and 6 in the fifth row (the second and third number in the row). The second number 10 in the sixth row is the result of adding the numbers 6 and 4 in the fifth row (the third and fourth number in the row).

Pascal's triangle can be used to quickly determine a specific coefficient of a binomial on-the-fly. The function `pascal_triangle_coefficient` is implemented to provide the coefficients for the first 32 rows (row 0 through row 31). This implementation was initially tested by requesting the coefficients for all rows, which returned correct results, but spurious results appeared in its application context. In case of error, the function is supposed to return -1, but it instead returned meaningless values.

14.4.1 Test Setup

Setting up a test starts with identifying the parameter types of the function declaration:

```
int32_t pascal_triangle_coefficient(int8_t n, int8_t k);
```

The bit length of both input parameters is 8 bits, the combined bit length covering all possible input values is 16 bits. Each parameter receives an `INT8` PTP as its argument. The operating interval of `INT8` is $[-128 \dots 127]$, which covers the entire operating interval of the input variables as well as their boundary values. An oracle evaluates the return value of `pascal_triangle_coefficient`:

```
#include <peregrine_in_vivo.h>

/* ... */

PEREGRINE_RED_CLIP

coeff = pascal_triangle_coefficient(INT8, INT8);
if (coeff!=-1 && (coeff<1 || coeff>300540195)) PEREGRINE_ORACLE;

PEREGRINE_BLACK_CLIP
```

14.4.2 Oracle Implementation

The conditional expression

```
if (coeff!=-1 && (coeff<1 || coeff>300540195)) PEREGRINE_ORACLE;
```

is an oracle implementation. It stops the program (Testee) and raises `PEREGRINE_ORACLE` if `pascal_triangle_coefficent` returns a value that is outside the interval $[1 \ldots 300540195]$. The oracle should not terminate the test, as test control and program control of Testee is with Tester.

The smallest value entry in Pascal's triangle is 1 and the highest value in row 31 is 300540195. A returning value outside this interval is certainly not an entry within the first 32 rows. An exception is made for the value -1, as this value indicates that a fault has been anticipated and caught by `pascal_triangle_coefficent`.

14.4.3 Test Result Evaluation

Compiling Testee and running it through the GUI reveals a reproducible error that is detected by the oracle as a Fail. The actual test execution time is negligible, also because the fault is triggered after a couple of thousand iterations. The probe logs[5] of five test runs are summarized in Table 6.

Test Run	#Iterations to Fail	n	k
1	11345	27	28
2	5130	15	16
3	9519	30	31
4	234	26	27
5	7455	22	23

Table 6: Combination of input values triggering an unexpected boundary fault.

[5]Note that the probe log lists the last injected probe as the first one, but C function parameters are evaluated from last to first.

Analyzing the values for n and k that cause the oracle to detect a Fail reveals a boundary value fault, a row can not be longer than its index. The first line of `pascal_triangle_coefficient` handles boundary values and returns and error value in case a parameter is out of bounds:

```
if (n<0||n>31||k<0||k>n+1) return -1;
```

Changing the precondition $k > n + 1$ to $k > n$ should fix this fault. This fault rarely occured because requests are usually made within bounds. The reason why this precondition was misaligned is due to the initial observation in the implementation process that rows have one more entry than its line number. Although this is in itself correct, the index for row entries was wrongly assumed to start at 1 instead of 0. When this was conceptually corrected in the code body, the preconditions were partially overlooked and manual (biased) testing by primarily applying input within the boundary values left this fault undetected.

14.4.4 Test Reliability

After changing the precondition, the test was initiated by terminal command for $500 * 10^6$ iterations. The oracle did not raise a Fail. The probe log, without the probe values, is given below:

```
- - - - - - - - - Test Results - - - - - - - - -

Test status: Finished without raising Fail

Iterations performed: 500 M 0 k 0
Iterations set to: 500 M 0 k 0
Iterations maximum: 1 G 0 M 0 k 0

Total number of PTP bits injected: 8 G 0 M 0 k 0
Combined PTP bit length B per iteration: 16

For the PTP injection points:
    If program peccability p is maximally 1/(2^16)
        then end test reliability t is: → 100 %

    p is the probability for a PTP to uncover unexpected program behavior.
    t is the probability the test is conclusive.

PTPs logged: Last 16
Timeout Testee (seconds): 2
True random seeding: Enabled

Time stamp start test: Thu Mar 16 18:50:54 2023
Time stamp end test: Thu Mar 16 18:52:22 2023
Time elapsed (seconds): 87.824120
```

It took 88 seconds to execute this test. The combined bit length B of the two `INT8` PTPs is 16 and the probability p to uncover a fault with a single PTP equals 2^{-B}. The test has a reliablity t of:

$$\begin{aligned} t &= 1-(1-p)^{N_T} \\ t &= 1-(1-\frac{1}{2^{16}})^{500*10^6} \\ t &\rightarrow 100\% \end{aligned}$$

The probability that `pascal_triangle_coefficient` behaves as expected is approaching 100 percent, which is similar to saying that the probability for unexpected program behavior to occur is approaching 0 percent.

Alternatively, if a test reliability of 99.9 percent would be considered good enough (e.g., for quick code-test-evaluate cycles), the number of minimally required test iterations would be:

$$\begin{aligned} N_{min} &= \frac{log(-t+1)}{log(1-p)} \\ N_{min} &= \frac{log(-0.999+1)}{log(1-\frac{1}{2^{16}})} \\ N_{min} &= 452.7*10^3 \approx 453*10^3 \end{aligned}$$

Executing $453*10^3$ iterations would take $88/(500*10^6/453*10^3) \approx 80$ milliseconds.

14.5 Unit Test

The source code of the `pascal_triangle_coefficient` function in Section 14.4 is presented below:

```
int32_t pascal_triangle_coefficient(int8_t n, int8_t k) {
    if (n<0||n>31||k<0||k>n) return -1;
    int ptc[n];

    if(n==0) ptc[0]=1;
    if(n==1) {
        ptc[0]=1;
        ptc[1]=1;
    }
    else {
        int tmp[n], i, j;
        tmp[0]=1; tmp[1]=1;

        for(i=2;i<=n;i++) {
            for(j=0;j<=i;j++) {
                if(j==0||j==i) ptc[j]=1;
                else ptc[j]=tmp[j]+tmp[j-1];
            }
            for(j=0;j<=i;j++) tmp[j]=ptc[j];
        }
    }

  return ptc[k];
}
```

A unit test is prepared to verify functional and boundary behavior. The results of `pascal_triangle_coefficient` are compared against an alternative implementation with the following function declaration and function definition:

```
int32_t binomial_coefficient(int8_t, int8_t);

int32_t binomial_coefficient(int8_t n, int8_t k) {
    if (n<0||n>31||k<0||k>n) return -1;

    int64_t binco=1;
    if (k>n-k) k=n-k;
    for (int i=0;i<k;i++) { binco*=(n-i); binco/=(i+1); }

    return binco;
}
```

14.5.1 Test Version 1: Alternative Implementation

Although `binomial_coefficient` is a to-the-point implementation, the intermediate results overflow a 32-bit integer requiring the variable `binco` to be declared as a 64-bit integer. This does not meet the requirement to (also) target 32-bit platforms, but it is useful for setting up an oracle to compare results:[6]

```
#include <peregrine_in_vivo.h>

/* ... */

PEREGRINE_RED_CLIP

row=INT8, col=INT8;
coeff=pascal_triangle_coefficient(row, col);
if(coeff!=-1 && (    coeff<1 ||
                    coeff>300540195 ||
                    coeff!=binomial_coefficient(row,col))) PEREGRINE_ORACLE;

PEREGRINE_BLACK_CLIP
```

Next to checking the boundary values, the oracle now also compares the functional results of both implementations with each other. After probing the function with $1 * 10^6$ PTPs ($500 * 10^6$ test iterations), the oracle did not detect a fault.

14.5.2 Test Version 2: PTP Mapping

Meaningful values for `row` and `col` are in the interval $[0 \ldots 31]$, whereas `INT8` operates in the interval $[-128 \ldots 127]$. The test can be more focused by mapping the PTP values that are outside the interval onto this interval:

```
#include <peregrine_in_vivo.h>

/* ... */

PEREGRINE_RED_CLIP

row=INT8; while(row>33) row=row-36; while(row<-2) row=row+36;
col=INT8; while(col>33) col=col-36; while(col<-2) col=col+36;
coeff=pascal_triangle_coefficient(row, col);
if(coeff!=-1 && (    coeff<1 ||
                    coeff>300540195 ||
                    coeff!=binomial_coefficient(row,col))) PEREGRINE_ORACLE;

PEREGRINE_BLACK_CLIP
```

[6] Declaration of variables that are passed to the function follow the function declaration (i.e., `uint8_t row, col; int32_t coeff;`).

The values assigned to `row` and `col` are in the interval $[-2\ldots33]$ to include boundary values as well. The range of the interval $[-2\ldots33]$ is $255/35 = 7.3$ smaller than the range of the original interval. By PTP mapping, the number of relevant probes to test with has increased 7.3 times. After probing the function with $1*10^6$ PTPs ($500*10^6$ test iterations), the oracle did not detect a fault.

14.5.3 Test Version 3: PTP Bit Masking

The number of PTP values that can be generated per second is constant thus adding PTPs to the test setup will increase test processing time. Alternatively, by means of bit masking, multiple PTPs of a shorter bit length can be derived from a PTP of a longer bit length to decrease test processing time:

```
#include <peregrine_in_vivo.h>

/* ... */

PEREGRINE_RED_CLIP

tmp=UINT16;
row=-16 + (0b0000000000111111 & tmp);             /* [-16,-15,-14,...,47] */
col=-16 + ((0b1111110000000000 & tmp)>>10);        /* [-16,-15,-14,...,47] */

coeff=pascal_triangle_coefficient(row, col);
if(coeff!=-1 && (    coeff<1 ||
                    coeff>300540195 ||
                    coeff!=binomial_coefficient(row,col))) PEREGRINE_ORACLE;

PEREGRINE_BLACK_CLIP
```

With a bit mask using the first and last 6 bits of `UINT16`, the intervals assigned to `row` and `col` is for both cases $[-16\ldots47]$. The right shift operator $>>$ moves its left operand value b bits to the right, where b is specified by its right operand. Three bits of the PTP are not used. After probing the function with $1*10^6$ PTPs ($500*10^6$ test iterations), the oracle did not detect a fault.

14.5.4 Test Performance

As shown in Table 7, test performance benefits from not using more PTPs than necessary and optimizing the operations performed on a PTP before injecting them into the target test code. The specifications of the hardware platform that has been used for comparing the three test versions are summarized in Section13. Although the processing time of a test may be longer, a test can be made more relevant if PTP values outside a specified test interval are mapped onto this interval. Table 8 shows the approximate number of test probes that map to the specified test interval for each test version. Among the three test versions, PTP bit masking has the best performance. PTP mapping by using while loops is a relatively expensive operation.

Test Version	1	2	3
Test duration [seconds]	63	388	132

Table 7: Unit test processing time for $500 * 10^6$ iterations.

Test Version	1	2	3
Test interval `row`	$[-128...127]$	$[-2...33]$	$[-16...47]$
Test interval `col`	$[-128...127]$	$[-2...33]$	$[-16...47]$
#probes per interval `row`	$\approx 2.0 * 10^6$	$\approx 13.9 * 10^6$	$\approx 7.8 * 10^6$
#probes per interval `col`	$\approx 2.0 * 10^6$	$\approx 13.9 * 10^6$	$\approx 7.8 * 10^6$
#probes per interval/second `row` and `col`	$\approx 63.5 * 10^3$	$\approx 71.6 * 10^3$	$\approx 118.2 * 10^3$

Table 8: The number of test probes per test interval for $500 * 10^6$ test iterations.

14.5.5 Test Reliability

By filling out the formula $t = 1-(1-p)^{N_T}$, where p is 2^{-B}, B is 16 for the combined bit length of the assigned PTPs and N_T is $500 * 10^6$, test reliability t is shown to approach 100 percent for each test version.

Table 9 provides an indicative list of the number of test iterations required if a test reliability of alternative value would be considered good enough (e.g., for quick code-test-evaluate cycles). As explained in Section 12.1, test reliability t expresses the probability that program peccability is equal or less than p.

Test Version	1	2	3
p	$(256 * 256)^{-1}$	$(36 * 36)^{-1}$	$(64 * 64)^{-1}$
N_T for $t = 0.99999$	$754.5 * 10^3$	$11.8 * 10^3$	$47.2 * 10^3$
N_T for $t = 0.9999$	$603.6 * 10^3$	$9.4 * 10^3$	$37.7 * 10^3$
N_T for $t = 0.999$	$452.7 * 10^3$	$9.0 * 10^3$	$28.3 * 10^3$
N_T for $t = 0.99$	$301.8 * 10^3$	$6.0 * 10^3$	$18.9 * 10^3$
N_T for $t = 0.98$	$256.4 * 10^3$	$5.1 * 10^3$	$16.0 * 10^3$
N_T for $t = 0.95$	$196.3 * 10^3$	$3.9 * 10^3$	$12.2 * 10^3$
N_T for $t = 0.90$	$150.9 * 10^3$	$3.0 * 10^3$	$9.4 * 10^3$

Table 9: Program peccability p and required number of test iterations N_T for a given test reliability t.

14.6 Monkey Test

Monkey testing can be categorized into smart monkey tests or dumb monkey tests. Dumb monkey testing is usually implemented as random, automated unit tests following a black box approach. A system is tested with no prior knowledge of its internal workings. Smart monkey testing is adding some basic prior knowledge of the internals workings of the system under test to dumb monkey testing. A sorting algorithm, QuickSort, is getting first dumb monkey tested and then smart monkey tested.

Quicksort is a divide-and-conquer sorting algorithm. It is among the sorting algorithms one of the most efficient approaches for sorting an array. Quicksort is straightforward in terms of mathematics and has been proven to be correct by formal reasoning. It is thus known that a correct implementation of Quicksort will behave as expected.

Problems with algorithms such as Quicksort typically occur due to imprecise implementation and the assumption it will work as expected due to its track record. Quicksort is often used in mission critical systems and its actual implementation requires rigorous testing before it is fielded. The implementation of QuickSort below will be monkey tested to quantify the probability that it will behave as expected:

```
public class QuickSort {
    static int part(int arr[],int begin,int end) {
        int pivot=arr[end];
        int i=(begin-1);

        for (int j=begin;j<=end-1;j++) {
            if (arr[j]<pivot) {
                i++;
                int tmp=arr[i];
                arr[i]=arr[j];
                arr[j]=tmp;
            }
        }

        int tmp=arr[i+1];
        arr[i+1]=arr[end];
        arr[end]=tmp;
        return (i+1);
    }

    static void quickSort(int arr[],int begin,int end) {
        if (begin<end) {
            int p=part(arr,begin,end);
            quickSort(arr,begin,p-1);
            quickSort(arr,p+1,end);
        }
    }
    public static void main(String[] args) {
        int[] arr=new int[5];
        arr[0]=1656; arr[1]=-9; arr[2]=-8754; arr[3]=768734; arr[4]=65436;

        quickSort(arr,0,arr.length-1);
    }
}
```

The minimum length the array has to be declared with is 5 as the test has to cover the behavior associated with the different type of array elements:

1. Begin element.
2. End element.
3. Connected to the begin element.
4. Connected to the end element.
5. Connected to in between elements on both sides.

14.6.1 Dumb Monkey Test

A test is setup without prior knowledge of the internals of QuickSort. The test assigns an `INT` PTP to each array element. An oracle checks if the output of Quiksort is correct by comparing each element in the array with the next element. If the current value in the array is greater than the next value `PEREGRINE_ORACLE` is raised:

```
#include <PeregrineInVivo.h>

public class QuickSort {

    /* Implementation of quickSort and part. */

    }

    public static void main(String[] args) {
        int[] arr=new int[5];

        PEREGRINE_RED_CLIP

        for(int i=0;i<arr.length;i++) arr[i]=INT;
        quickSort(arr,0,arr.length-1);
        for(int i=0;i<arr.length-1;i++) if (arr[i]>arr[i+1]) PEREGRINE_ORACLE;

        PEREGRINE_BLACK_CLIP
    }
}
```

For Java source code to be tested, the `$HOME/peregrine/peregrinejava/` directory and its contents have to be copied to the Java project directory. The source code is compiled with `cppjava QuickSort.java` and the resulting `QuickSort.class` is used for testing. The test can be executed by terminal command or directly by GUI.[7] See also Sections 8 and 10.

The test was run for $1 * 10^9$ iterations and the oracle did not raise a Fail. The probe values have been omitted from the test output results:

```
- - - - - - - - - Test Results - - - - - - - - -

Test status: Finished without raising Fail

Iterations performed: 1 G 0 M 0 k 0
Iterations set to: 1 G 0 M 0 k 0
Iterations maximum: 1 T 0 M 0 k 0

Total number of PTP bits injected: 160 G 0 M 0 k 0
Combined PTP bit length B per iteration: 160

Program peccability and test reliability: Not calculated (B > 40)

PTPs logged: Last 32
Timeout Testee (seconds): 2
True random seeding: Enabled

Time stamp start test: Fri Mar 17 13:57:37 2023
Time stamp end test: Fri Mar 17 14:08:49 2023
Time elapsed (seconds): 672.193098
```

The test took more than 11 minutes to execute with a test reliability t of:

$$\begin{aligned} t &= 1-(1-p)^{N_T} \\ t &= 1-(1-\frac{1}{2^{160}})^{1*10^9} \\ t &\rightarrow 0\% \end{aligned}$$

where p equals 2^{-B} and B is the combined bit length of the PTPs assigned to the array elements. Even though no failure has been detected, the test results can not be used as a meaningful quarantee that the QuickSort implementation does not include unexpected behavior.

If the number of test iterations would be set to $1 * 10^{12}$, the test would take more than 186 hours and test reliability would still effectively be at 0 percent. Duo to the long bit length B, a dumb monkey test is not meaningful for confirming whether QuickSort contains faulty behavior because it has no statistical relevance.

[7] The path to the `peregrinejava` directory is automatically taken into account by the GUI, but has to be specified if the test is executed by terminal.

14.6.2 Smart Monkey Test

A test is setup with prior knowledge of the internals of QuickSort. An algorithm like QuickSort is defined by the data it operates on, in this case 32 bit signed integers stored in an array. The core of the algorithm is designed around the comparison operator. The result of the comparison $a < b$ is $true$ for $b = a + 1$, $b = a + 2$, $b = a + 3$, etc., and $false$ for $a = b$, $a = b + 1$, $a = b + 2$, etc. The result of the comparison does not change if the difference in value between the operands is more than one. For any 32 bit integer value passed to one operand of the comparison operator there are $2^{32} - 1$ values that are not equal to the value passed to the other operand.

The test can be made statiscally meaningful by testing what is relevant in relation to the comparison operators, by focusing on array values that change the result of the comparison operators. Each array element is assigned a constant together with a short bit length PTP, creating a PTP interval around a constant. The constant is an arbitrary value. An `INT4` PTP generates values in the interval $-2^{-3} \ldots 2^{3} - 1$ that are added to the constant:

```
#include <PeregrineInVivo.h>

public class QuickSort {

    /* Implementation of quickSort and part. */

    }

    static final int CONST=987654;

    public static void main(String[] args) {
        int[] arr=new int[5];

        PEREGRINE_RED_CLIP

        arr[0]=CONST+INT4;
        arr[1]=CONST+INT4;
        arr[2]=CONST+INT4;
        arr[3]=CONST+INT4;
        arr[4]=CONST+INT4;

        quickSort(arr,0,arr.length-1);
        for(int i=0;i<arr.length-1;i++) if (arr[i]>arr[i+1]) PEREGRINE_ORACLE;

        PEREGRINE_BLACK_CLIP
    }
}
```

The combined bit length B of the PTPs is 20 bits, the setup then tests for a program peccability p is of 2^{-20}. The test should reach a test reliability t of 99.99 percent, the number of minimally required test iterations is then:

$$\begin{aligned} N_{min} &= \frac{log(-t+1)}{log(1-p)} \\ N_{min} &= \frac{log(-0.9999+1)}{log(1-\frac{1}{2^{20}})} \\ N_{min} &= 9.66 * 10^6 \approx 10 * 10^6 \end{aligned}$$

After $10 * 10^6$ executing test iterations, the oracle did not raise a Fail:

```
- - - - - - - - - Test Results - - - - - - - - -

Test status: Finished without raising Fail

Iterations performed: 10 M 0 k 0
Iterations set to: 10 M 0 k 0
Iterations maximum: 1 T 0 G 0 M 0 k 0

Total number of PTP bits injected: 200 M 0 k 0
Combined PTP bit length B per iteration: 20

For the PTP injection points:
    If program peccability p is maximally 1/(2^20)
        then end test reliability t is: 99,993 %

    p is the probability for a PTP to uncover unexpected program behavior.
    t is the probability the test is conclusive.

PTPs logged: Last 32
Timeout Testee (seconds): 2
True random seeding: Enabled

Time stamp start test: Fri Mar 17 14:26:02 2023
Time stamp end test: Fri Mar 17 14:26:06 2023
Time elapsed (seconds): 4.646023
```

Let's assume the longest available batch time for scheduling a test is 12 hours in a production environment (from 6.00 PM to 6.00 AM). The current test setup requires 4.6 seconds to execute $10*10^6$ test iterations, 12 hours of testing time would thus allow for approximately $93 * 10^9$ test iterations. For such a test, if the oracle would not raise a Fail, test reliability t would approach 100 percent in an almost absolut manner.[8]

[8]Running a test with $N_T = 25 * 10^6$ is also communicated as approaching 100 percent and probably good enough for most practical purposes.

14.6.3 Key Takeaway

The combined bit length of the PTPs determine the duration of the test. Although it is tempting to quickly assign PTPs to various test input variables, it will also quickly increase the combined bit length of the PTPs. The time needed for testing will exponentially grow every time a PTP is added. Having a basic understanding of the internal workings of a system is key for (1) not selecting too many PTPs, (2) assigning PTPs to injection points that are the crux of the program and (3) map PTP values to an operating interval that covers the values that make an injection pivot its outcome.

FCE Software Testing Without Bias
Fluxica Computer Engineering

Copyright © 2020-2023 Fluxica Computer Engineering. All rights reserved.