MCTrace User Manual (Version 1.0)

Galois (pate-darpa-amp@galois.com)

July 28, 2023

DTrace is a popular tool supporting the definition of small snippets of code which are then inserted into an application at specified locations.[1] For example, these code snippets might be inserted directly before and after every call made to a given function, for example a syscall such as write(), in order to track the time elapsed each time the function is called. Such probes are defined in the DTrace scripting language and inserted into the source, which is then compiled.

The MCTrace tool is intended to provide the same functionality as DTrace, without requiring access to application source code, or to the DTrace run-time. For MCTrace, the probes defined as DTrace scripts are compiled into the appropriate machine language for the target architecture, and then inserted into the binary. Note that the binary must be modified both to include the code for the probes themselves, as well as inserting the calls to those probes at specified locations in the original binary code.

Contents

[1] `https://en.wikipedia.org/wiki/DTrace`

1 Introduction

This is the User Manual for MCTrace, updated to correspond with the software release as of the end of AMP Phase 2 (7/28/23). This manual can be found within the repository snapshot comprising the Phase 2 code delivery, in the directory `docs/usermanual`.

In addition to building a Docker image suitable for running MCTrace in the development environment provided by the software archive, README.md includes instructions for building a standalone Docker image, suitable for use by a potential user who is not interested in development or in the source code provided in the archive. Our intention is to provide this standalone Docker image on request, rather than expecting prospective users to build it themselves. Use of the standalone Docker image is discussed in Section 4.

2 MCTrace

This section describes the MCTrace tool, including its execution environment and requirements. The information in this section can also be found in the repository at the top level, in the file `MCTRACE.md`.

For instructions on how to build MCTrace, see Section 3.

2.1 Introduction to MCTrace

The MCTrace tool enables users to insert instrumentation into binaries in order to collect fine-grained tracing information. MCTrace functions similarly to DTrace but does not require any operating support (or even an operating system). The input format of MCTrace is a subset of the DTrace probe script language. Prior knowledge of DTrace concepts and terminology is assumed in this document.

2.2 Concept of Operations

After identifying a program to be instrumented (e.g., for debugging purposes or to collect telemetry), the steps to use MCTrace are as follows:

1. Write a DTrace script that collects the desired data.
2. Run the `mctrace` tool in `instrument` mode to insert probes into the binary as directed by the script.
3. Run the instrumented binary.
4. Collect telemetry data emitted by the instrumented binary and use `mctrace` and associated scripts to decode it.

The rest of this section provides more details on the steps above.

2.3 How MCTrace Works

Using MCTrace requires:

- a PowerPC or x86_64 ELF binary to instrument,
- an object file implementing the MCTrace Platform API (see below), and
- a DTrace probe script containing the probes that will be used to modify the provided ELF binary.

MCTrace works by producing a modified version of its input binary that calls Dtrace probes at the points described in the DTrace probe script. MCTrace compiles

DTrace probe scripts into native code using a compiler backend (e.g., LLLVM). It then uses binary rewriting to insert the generated probes into the binary. Through static analysis, it identifies program locations corresponding to DTrace probe target sites; at each target site, it inserts calls to the compiled probes.

Some supported DTrace language features require access to platform-specific functionality such as memory allocation. Since the DTrace code will run within the context of the modified binary rather than an operating system kernel, MCTrace requires some additional code to provide access to such platform-specific features. The MCTrace tool provides the input program with access to this platform-specific functionality by way of an object file of compiled code called the Platform Implementation. This object code that implements the required functions must conform to a set of C function prototypes called the Platform API. A complete implementation of the Platform API must provide implementations of all of the functions the in header file `mctrace/tests/library/include/platform_api.h` provided in the MCTrace GitHub repository. Once compiled, the platform API implementation must be provided to the `mctrace` as the `--library` argument when invoking the `mctrace` tool.

For demonstration purposes, simple platform API implementations for PowerPC and x86_64 Linux user-space are available in the repository in `mctrace/tests/library/` and in the standalone Docker release image in `examples/library/` in both source and object code forms.

The following example demonstrates how an x86_64 binary `foo` would be instrumented and how its produced telemetry would be obtained. In this example, the user wants to count the number of times that the program performs a file read by measuring the number of calls to the `read` system call. The probe script to accomplish this is as follows:

```
int read\_calls;

::read:entry {
  read\_calls = read\_calls + 1;
  send(0);
}
```

This probe increments the `read_calls` DTrace variable just prior to each call to the `read` system call. Futhermore, the complete set of DTrace global variables (containing only `read_calls` in this case) is then transmitted as telemetry according to the platform implementation of `send()`.

Once the probes are written, `mctrace` would be invoked as follows.

- The `foo` binary and the `probes.d` probe script are provided to `mctrace` as inputs.
- The `--output` option tells `mctrace` where the instrumented binary should be written.
- The `--var-mapping` option tells `mctrace` where to record metadata that allows it to later decode telemetry data.
- The `--library` option tells `mctrace` where to find the platform implementation object code.
- The `--var-mapping` option tells `mctrace` where to write its metadata describing the encoding of the telemetry data stream.

```
$ mctrace instrument --binary=foo \
                --output=foo.instrumented \
```

```
--var-mapping=foo.mapping.json \
--library=mctrace/tests/library/platform\_impl.o \
--script=probes.d
```

The resulting binary can then be run:

```
$ ./foo.instrumented 2>telemetry.bin
```

Once the instrumented binary has finished running, the file `telemetry.bin` will contain the binary telemetry data corresponding to the value of `read_calls` after each invocation of the probe. The telemetry data must then be decoded:

```
$ extractor.py foo.mapping.json --columns < telemetry.bin
```

The `extractor.py` script uses the telemetry mapping file to decode the written telemetry data.

NOTE: In this example, the instrumented binary was run with a `stderr` redirect. This is because the x86_64 platform implementation for `send()` used in this example writes the DTrace global variable data to `stderr`. In practice, `send()` would trigger some other platform-specific mode of data transmission such as sending packets on a network connection, writing to a local bus, etc.

For full details of the MCtrace tool's command line usage, run `mctrace instrument --help`.

2.4 Supported DTrace Language Features

This information is also available in the repository, in `DTRACE.md`.

MCTrace uses the DTrace language as the means for expressing how it should modify its input binary. While MCTrace does not implement all of the DTrace language, the following core DTrace language features are supported:

- Probe name pattern-matching. Supported metacharacters are '*', '¿, '[...]', and '
- Global variables
- Constants with suffixes 'u', 'U', 'l', 'L', 'ul', 'UL', 'll', 'LL', 'ull', and 'ULL'
- Data types:
  - 'int'
  - 'char'
  - 'short'
  - 'long'
  - 'long long'
- Integer arithmetic operators '+', '-', and '*'
- Builtins:
  - 'int arg0'
  - 'timestamp'
  - 'long ucaller'

Example DTrace probe scripts demonstrating these features can be found in `mctrace/tests/eval/`.

In addition to the core DTrace language features listed above, MCTrace also supports some additional DTrace constructs specific to MCTrace:

- A `send` action of the form: `send(<numeric channel ID>)`. Invoking `send` will result in an invocation of the `platform_send` function in the Platform API with that channel ID and the global data store.

- A `copyint32(<address>)` subroutine that returns the 32-bit value from the specified location.

2.5 Current Limitations of MCTrace

As of the Phase 2 release (7/28/23), MCTrace has the following limitations:

- MCTrace supports only statically-linked input binaries.
- In the current implementation, `arg0` always returns a 32-bit value (on both PowerPC and x86_64 platforms). Similarly, `copyint32` takes a 32-bit address as its input on both platforms (and returns a 32-bit value). As a result, these work best on the PowerPC 32-bit platform since the argument and return value width match the architecture.
- Platform API implementations are subject to the following restrictions:
- Functions in the Platform API implementation must be self-contained and cannot call other functions even in the same object file.
- Functions in the Platform API (as opposed to DTrace scripts) cannot make use of global variables.

3 Building MCTrace

The information in this section can also be found in the repository file `README.md`.

MCTrace can be built for one of two purposes: either for local development in a Haskell build environment, or for release as a Docker image. Instructions for each method are detailed below.

3.1 Development Build Instructions

The development environment setup and build processes are automated. The build process requires Ubuntu 20.04. To perform a one-time setup of the development environment including the installation of LLVM, cross compilers, and other required tools, run the development setup script:

```
./dev_setup.sh
```

Once the development environment is set up and the required tools are installed, MCTrace can be built with the build script:

```
./build.sh
```

After the build has completed, various cross compilers and other tools can be added to the `PATH` environment variable for easier access with:

```
. env.sh
```

To build the example test programs for x86_64 and instrument them using various testing probes, run:

```
make -C mctrace/tests/full
```

To do the same for PowerPC, run:

```
make -C mctrace/tests/full ARCH=PPC
```

MCTRACE can then be run manually:

```
cabal run mctrace <args>
```

3.2 Release Build Instructions

To build the stand-alone release Docker image, execute the following from the root of the repository:

```
cd release
./build.sh
```

This will build two docker images:

- A self-contained image that contains MCTrace, its dependencies, associated tools, and examples. For information on using this image, see Section 4.
- A minimal image containing just MCTrace and its dependencies. A helper script, `release/mctrace` has been provided to run the command in a container. Note that paths passed to this script should be relative to the root of the repository and paths outside of the repository will not accessible.

4 User Release

This section describes how to run the standalone Docker image for MCTrace described in Section 1. Much of this information is also available in the repository as `release/README.md`, or in the standalone Docker image itself, as `README.md`, at the top level.

To load and run the image, run the following commands from a directory containing the image, using the filename `mctrace.tar.gz`

```
docker image load -i mctrace.tar.gz
docker run -it -w /mctrace-test mctrace
```

This will drop you into a bash shell within the Docker container in the directory `mctrace-test` where you can use `mctrace` to instrument binaries. All relative paths mentioned in this section are relative to `mctrace-test`.

4.1 Docker Image Contents

The docker image contains PowerPC and x86_64 test programs and example probes that can be used to exercise MCTrace. Important folders are as follows:

- `examples/eval` contains a collection of probes primarily derived from those provided to us by WebSensing. The probes have been modified slightly after discussions with WebSensing to fit the currently supported DTrace syntax in MCTrace.
- `examples/full` contains source code and binaries for bundled test programs.
- `examples/binaries` contains binaries from a statically compiled version of GNU coreutils for use with `mctrace`.

4.2 Using MCTrace in this demonstration

The `mctrace` tool is in the shell `PATH` when running within Docker, so no special steps are needed to be able to invoke the executable. For instructions on how to run the `mctrace` tool with the appropriate command-line arguments, see `MCTRACE.md` included in the Docker image.

As an example, the `read-write-syscall-PPC.4.inst` binary in this distribution is the instrumented version of the PowerPC binary `read-write-syscall-PPC` and was instrumented with the following `mctrace` command:

```
mctrace instrument --binary=/mctrace-test/examples/full/read-write-syscall-PPC \
   --output=/mctrace-test/examples/full/read-write-syscall-PPC.4.inst \
   --library=/mctrace-test/examples/library/PPC/platform_impl.o \
   --var-mapping=/mctrace-test/examples/full/read-write-syscall-PPC.4.json \
   --script=/mctrace-test/examples/eval/write-timing-probe.d
```

- The `--binary` and the `--script` options tell mctrace to instrument the specified binary with the given probe script.
- The `--output` option specifies the name for the instrumented binary.
- The `--library` option specifies the path to the Platform API implementation.
- The `--var-mapping` option tells `mctrace` where to record metadata that allows it to later interpret the collected telemetry.

The above command instruments the binary with probes that triggers at the start and end of the `write` function and computes timing information for the call. Note that the instrumentation command produces a significant amount of DEBUG logs, that can be ignored at the moment.

When probes call the DTrace `send` action, the current test implementation of `send` pushes the set of telemetry variables, in a compact binary format, to the standard error. A script `extractor.py` has been included with the image to help interpret this data.

To invoke the instrumented binary and use the `extractor.py` script to decode any emitted telemetry:

```
/mctrace-test/examples/full/read-write-syscall-PPC.4.inst 2>&1
>/dev/null | \
  extractor.py /mctrace-test/examples/full/read-write-syscall-PPC.4.json \
  --extract --big-endian
```

This produces output similar to the following:

```
{"write_count":1,"write_elapsed":162240,"write_ts":1681222607714740774}
{"write_count":2,"write_elapsed":1740,"write_ts":1681222607714800756}
{"write_count":3,"write_elapsed":1309,"write_ts":1681222607714803400}
{"write_count":4,"write_elapsed":1344,"write_ts":1681222607714805742}
{"write_count":5,"write_elapsed":1228,"write_ts":1681222607714807992}
{"write_count":6,"write_elapsed":1222,"write_ts":1681222607714810214}
{"write_count":7,"write_elapsed":1257,"write_ts":1681222607714812555}
```

Note that `2>&1 >/dev/null` has the effect of piping the standard error to the next command while suppressing the standard output of the command. We do this because the provided platform API implementations writes `send()` data to `stderr` and we need that data to be piped to the extractor script.

When extracting telemetry data from instrumented PowerPC binaries, the flag `--big-endian` must be passed to the extractor script as in the command above. The flag should be elided when working with x86_64 binaries.

The `extractor.py` script offers a few other conveniences when extracting data from instrumented programs; for example it can produce columnar outputs and filter columns. See `extractor.py --help` for details on these options. See

See the file README.md for a list of other binaries for PowerPC and x86_64 included in the Docker image as well some example probes that can be used to instrument each binary. Note that many other combinations of example programs and probes can work together; the full list of combinations can be found in `examples/full/Makefile`.

5 Acknowledgement

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract Number N66001-20-C-4027. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA & NIWC Pacific.