Frama-C/Eva applied to the Chrony source code: a first analysis

André Maroneze, CEA Tech List

March 15th, 2018

Introduction

As part of an effort sponsored by Orolia, researchers from the LIST, CEA Tech laboratory applied the Frama-C code analysis framework on the Chrony source code, in an attempt to verify the absence of run-time errors. This analysis has been done using the Eva plug-in (based on abstract interpretation) and used as entry point the test distributed with Chrony in `test/unit/ntp_sources.c`. This test initializes several components of the code and performs a reasonable amount of computation, so it seemed a good candidate. Other tests, especially those in the `test/unit` directory, can be set up as entry points for similar analyses.

The analysis has not been deployed to the full extent of the possibilities of Frama-C/Eva; it is possible to further refine it for more precise results. The findings so far do not allow any definitive conclusion about the overall absence of run-time errors, however they do provide some results.

Overall impressions

The Chrony source code is overall well-written w.r.t. constraints imposed by the ISO C standard; Frama-C is based on C99, and no specific modifications were required to parse the source code of Chrony which indicates that it contains few or no non-portable constructs. This ensures better portability and future-proofing of the code.

Also, some defensive programming patterns, such as checking the return code of `scanf` are present in the code. They improve its robustness and are also helpful for code analysis tools. While Chrony has not been written specifically for a code analysis tool, patterns that are helpful for analysis are often also helpful for manual reviews: predictable control flow, assertions indicating code invariants and unreachable code, etc.

Overall, only minor issues were detected with a preliminary run of Frama-C/Eva. However, some code patterns prevent the analysis from being thorough enough to ensure the absence of runtime errors in the considered code parts.

A few patches were applied to the code, either to remedy an issue, or to allow the analysis to proceed further. Some improvements to the Frama-C platform were performed during this analysis, most of which are available in Frama-C 17 - Chlorine. The analysis parametrization is made available as part of Frama-C's open source case studies Github repository. It contains a directory `chrony` with the appropriate `GNUmakefile` and accompanying stubs to allow the analysis to be replayed and modified at will, provided Frama-C 17 is installed. The analysis takes a few minutes to complete (after running `make`), and the results can be inspected using Frama-C's graphical interface (`make ntp_core.eva.gui`).

However, it must be noted that some intrinsic limitations will be present due to some code patterns, therefore it is very unlikely that a 100% alarm-free analysis will be possible, at least without some substantial evolutions of the Frama-C platform and the Eva plug-in. It is likely that future releases of Frama-C will improve these results.

Issues found

Most issues found here were identified as "red alarms" by Eva plug-in, which indicates that they are bound to definitely happen during program execution, assuming the initial context setup was appropriate. These issues typically correspond

to undefined behaviors as stated in the C99 standard.

1. (Undefined behavior) C stdlib usage

logging.c:140, in function `LOG_Message`:

The call to `gmtime` should ideally be guarded, in case it fails, to prevent dereferencing a `NULL` pointer.

A suggestion of a patch is presented below:

```
--- a/logging.c
+++ b/logging.c
@@ -137,9 +137,14 @@ void LOG_Message(LOG_Severity severity,
   if (!system_log) {
     /* Don't clutter up syslog with timestamps and internal debugging info */
     time(&t);
-    stm = *gmtime(&t);
-    strftime(buf, sizeof(buf), "%Y-%m-%dT%H:%M:%SZ", &stm);
-    fprintf(file_log, "%s ", buf);
+    struct tm *gt = gmtime(&t);
+    if (gt) {
+      stm = *gt;
+      strftime(buf, sizeof(buf), "%Y-%m-%dT%H:%M:%SZ", &stm);
+      fprintf(file_log, "%s ", buf);
+    } else {
+      fprintf(file_log, "[GMTIME_ERROR]");
+    }
 #if DEBUG > 0
     if (debug_level >= DEBUG_LEVEL_PRINT_FUNCTION)
       fprintf(file_log, "%s:%d:(%s) ", filename, line_number, function_name);
```

reference.c:545, in function `maybe_log_offset`:

```
if (p) {
  if (gethostname(host, sizeof(host)) < 0) {
    strcpy(host, "<UNKNOWN>");
  }
  fprintf(p, "Subject: chronyd reports change to system clock on node [%s]\n", host);
  fputs("\n", p);
  stm = *localtime(&now);
  strftime(buffer, sizeof(buffer), "On %A, %d %B %Y\n  with the system clock reading \
    %H:%M:%S (%Z)", &stm);
```

There are two minor issues here:

- `gethostname`, in POSIX, is not guaranteed to be null-terminated even when no error returns. Note that the GNU libc ensures this is the case, so this is only an issue in terms of portability to other libc implementations;
- `localtime` may return `NULL` in case of error, so just like in the previous patch for `gmtime`, ideally one might want to check its return value before trying to dereference it (in line `stm = *localtime(&now);`).

2. (Unconfirmed) In conf.c, `parse_fallback_drift`, near line 948:

   ```
   sscanf(line, "%d %d", &fb_drift_min, &fb_drift_max);
   ```

   Variables `fb_drift_min` and `fb_drift_max` are declared as (signed) `int`, and scanned as such, but they are later used as index arrays, which seems to indicate they might have been `unsigned`.

   This is marked as *unconfirmed* because the alarms are indicated as *Unknown* in the tool (indicating the possibility of a false alarm); some manual inspection indicates that this seems to be the case, but a more detailed analysis would be necessary to confirm it.

3. In `keys.c`, function `determine_hash_delay`: variable `NTP_Packet pkt` in the stack is passed to (further down) `MD5Update`, where it is (in some callstacks) read without having been initialized. This is not technically undefined behavior if the type used while reading is an `unsigned char` (due to the absence of trap representations), but nevertheless this can introduce some non-determinism during execution/debugging. It also renders the code more fragile: if someone tries to access the memory using some non-char type, it will create some undefined behavior. Initializing `pkt = {0}` would avoid the issue. Unless there is some significant loss in performance related to such initialization, patching is recommended.

4. Occurrences of invalid format specifiers are present in calls to formatted input/output functions, namely `snprintf`, called in `util.c`, function `UTI_IPToString`. These occurrences constitute undefined behavior according to the C standard (recalled by CERT C as rule FIO47-C), even if they are benign in current compilers, which should not miscompile them. For instance, some `unsigned long` variables are printed using `%ld` instead of `%lu`. Later in the same function, several `unsigned char` variables are printed using `%02x` specifiers. Due to default argument promotions, the `unsigned char` values are actually converted to `int`s, thus the `x` modifier (which requires unsigned values) does not apply to them. Instead, it would be better to use the proper length modifiers for `char`s, that is, `%02hhx`. Note that, in GCC 5 (or newer), option `-Wformat-signedness` emits warnings for some of these issues. Other issues are possibly present in logging-related functions, but not all of them were present during the analysis with Frama-C/Eva, so they are not listed here.

Technical limitations

One of the main points that hinders the analysis with Frama-C/Eva is the presence of dynamically allocated memory in several parts of the code. While Eva is able to handle it, dynamic memory generates too much imprecision, leading to many false alarms that would not occur if the memory had been statically allocated. This is a current limitation of the plug-in that makes it suboptimal to analyze Chrony's code.

Another aspect is the intertwining of parsing and code execution, namely the pattern:

```
...
} else if (!strcasecmp(cmd, "require")) {
  src->params.sel_options |= SRC_SELECT_REQUIRE;
} else if (!strcasecmp(cmd, "trust")) {
  src->params.sel_options |= SRC_SELECT_TRUST;
} else if (!strcasecmp(cmd, "key")) {
  if (sscanf(line, "%"SCNu32"%n", &src->params.authkey, &n) != 1 ||
    src->params.authkey == INACTIVE_AUTHKEY)
  return 0;
...
```

Due to the genericity in the initial environment (considering any sequence of characters), this leads to a very imprecise analysis, where all branches are considered as potentially reachable.

Linked lists are a data structure that is not well-suited for Eva, since they combine dynamic memory allocation with data-dependent control flow. For instance, in `sched.c` there is a `TimerQueue` data structure used by functions such as `SCH_AddTimeout` and `SCH_RemoveTimeout`. The usage of these linked lists leads to several alarms.

Frama-C/Eva is currently unable to deal with recursive functions. During parsing of the configuration file, function `parse_include` recursively calls other parsing functions. A stub has been used to replace the calls to this function; this stub should provide an over-approximation of the variables modified by the function, ensuring that the results indicated by the analysis remain sound. This obviously precludes the identification of errors in the `parse_include` function itself (and those called by it). Combining analyses with other plug-ins might provide workarounds for this situation.

Extending the analysis

As previously mentioned, the analysis performed with Frama-C/Eva can be extended to other entry points. Using the available unit tests, it is straightforward to parameterize a different analysis to consider a different part of the code base. Note that the chosen test, `ntp_core.c`, is the largest one and activates most of the code base of Chrony. Other tests may however provide some complementary information and, due to their smaller size, allow for a faster analysis. The provided `GNUmakefile` contains an example on how to add a second test (`regress.c`), with comments on the few lines that need to be added.

Technical notes

During the analysis, a few functions were stubbed/adapted for better analyzability: to improve precision, efficiency, or to work around limitations. Here is a list of some of these changes:

- `test_unit` (in `test/unit/ntp_core.c`): contains a loop where 1000 tests are executed sequentially. Because of the abstractions performed by the analysis with Eva, unless each test depends on the result of the previous one, there is no need to iterate multiple times: one single test performed in an abstract way will provide information about all possible executions. We therefore parametrized the number of iterations via a macro `NB_TESTS`, which can be defined via `-DNB_TESTS=` and set to a low value to avoid unnecessary recomputations by the analysis.
- `Transform`: computation-intensive function that applies several bitwise operators to a small buffer, as part of the hashing done by MD5. The control flow is fixed and the computations are performed over unsigned integers, which results in virtually no possibility of runtime error. For Frama-C/Eva, this function has been replaced with a specification that abstracts its computations, resulting in a much more efficient analysis. Since the actual final values computed by the function are irrelevant (our analysis assumes an abstract input), once we know that there are no runtime errors (since the analysis reported no alarms), we abstract it away. This is a simplified version of the ACSL specification of the function:

  ```
  /*@
    requires \initialized(in+(0..15));
    assigns buf[0..3];
  */
  ```

  This specification (1) requires that each value in `in` be *initialized*, that is, (a) it corresponds to a valid memory location, and (b) its value it not indeterminate (to avoid trap representations). If those properties are verified at the call site, then our specification guarantees that only memory locations `buf[0..3]` are modified by the function. We do not specify *which* concrete values are set, but our analysis does not require them anyway: the computation of *any* MD5 hash, for any input values, must be free of runtime errors.
- `parse_include`: as is often the case in parsing functions, this function is part of a recursive call chain (`CNF_ParseLine -> parse_include -> CNF_ReadFile -> CNF_ParseLine`). Eva cannot currently handle recursive calls, so a workaround must be used. To increase the amount of analyzed code, and to minimize the complexity of the stub/specification to be used, it is often better to stub the lowest function before a recursive call takes place. In this case, `parse_include` is invoked if and only if the configuration file contains such an inclusion, so commenting it out is a quick way to proceed with the analysis. This obviously precludes Eva from identifying issues in the stubbed function, but it is simpler than devising a sound and precise specification for the function, when its side-effects are hard to estimate (as is often the case with recursive functions). Future versions of Eva might be able to avoid this step.
- `LOG_FileWrite`: logging functions are often source of issues, relying on variadic functions, macros, reflection and specialized mechanisms to output information. While they may contain runtime errors and should be analyzed when possible, their impact is often limited. In this case, we disabled it after the first run to minimize the amount of noise while refining the analysis. It can be enabled back, at the price of a dozen extra alarms.
- `strcasecmp`: a temporary specification has been added to this function, pending its inclusion in the Frama-C libc. Specifications in the Frama-C libc are added incrementally, and new case studies often include non-C99 functions

(POSIX, BSD or GNU extensions) for which no specifications have yet been written. Such specifications can then be submitted for Frama-C developers to evaluate their correctness, precision and relevance, and are often included in the following Frama-C release.

- C stubs for `qsort` and `gethostbyname`, in `fc_stubs.h`: these libc functions have no specification in the Frama-C libc, or their specification is too imprecise and/or complex in ACSL; in this case, writing a short C stub is often a better approach, especially due to the possibility of using non-deterministic Frama-C built-ins (e.g. `Frama_C_interval`). For instance, the `qsort` stub does *not* sort its results, but instead shuffles them; this abstracts the fact that no input array should result in a runtime error when sorted. However, if the caller function does rely on the array being sorted to avoid runtime errors, then our stub will lead to false alarms. For `gethostbyname`, the situation is slightly different: the function returns a pointer to a struct that must be dynamically allocated. Eva cannot currently "invent" the variable which corresponds to this struct, so a code stub is necessary for soundness. However, the actual code of the function itself is too complex and requires stubbing/specifying several other auxiliary functions. In this case, the approach consists in returning an under-approximation of the possible results, allowing the analysis to proceed, but lacking in completeness w.r.t. all possible values of the function. Note that the structure returned by `gethostbyname` contains a linked list of strings; producing a correct over-approximation of such list would require an enormous amount of effort, for a limited benefit. If this part of the code were indeed critical, then the best solution would consist in incorporating the entire code of the actual function itself, and analyzing it. This can be done in a later stage of the analysis, after more important issues are resolved.

Final remarks

Overall, we would praise Chrony as having a clean code base, with a code style that is overall amenable to tools based on static and semantic analyses such as Frama-C. The code has not been specifically developed towards such analyses, which could help provide extra guarantees about it, but in its current form it already presents some helpful patterns.

We expect to be able to further proceed with the analysis in the future, either identifying more issues, or providing extra guarantees about its correctness.

Other Frama-C plug-ins that may be useful to pursue further exploration of the code base are WP and E-ACSL. The former requires some effort in terms of annotations such as function specifications and loop invariants; it could be used in the more critical parts of the code, to provide guarantees about some functional properties. It can also be used to complement and/or replace runtime error analyses in some parts of the code. Finally, since Chrony is a software that can run on a standard Linux machine (in other words, it is not some embedded software that requires a specific hardware and/or operating system), the E-ACSL plug-in can be used to perform runtime detection of undefined behaviors, by instrumenting the code and then executing it on the same machine running the Frama-C analysis.