Continuous Deductive Verification with Frama-C

Frama-C & SPARK Day 2019, Paris, France

Denis Efremov, efremov@ispras.ru
Jens Gerlach, jens.gerlach@fokus.fraunhofer.de
Continuous Verification
What problems we are trying to solve?

• Formal verification of a project (e.g., ACSL-By-Example)
  • Global logic definitions (lemmas, common predicates, ...)
  • Changes in a toolchain

• Formal verification of a continuously developed project
  • Developers != Verifiers
  • Can’t be verified once and for all
  • Verified code sometimes differs from the original one
  • Need to maintain specifications to reflect code changes
Continuous Verification
What could we do?

• Continuous Integration (CI) + Verification == Continuous Verification (CV)
• Automation of proofs as much as possible
  • Auto-active verification
  • Special strategies for VCs transformations and solvers runs
  • Contradiction checking
    • Transformation (smoke detector in Why3)
    • //@ assert 0 == 1; //@ check false;
• Frequent replays of proofs
• Tracking of differences between the original and verified code
  • In case verifiers can’t force developers to accept the verified code
Vessedia Project

• IoT Operating System (OS) Contiki
  • More than 1000 commits in 2018 by 43 authors
  • Changed more than a thousand files
  • Added 70 thousand lines of code and deleted approximately 16 thousand

• Formal verification of parts of the Contiki with Frama-C/WP
• Verified parts: AES-CCM modules, lists functions, memory allocation module
• Project: https://www.vessedia.eu/

• Towards Formal Verification of Contiki: Analysis of the AES–CCM* Modules with Frama-C. A. Peyrard, N. Kosmatov, S. Duquennoy, S. Raza
• Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. A. Blanchard, N. Kosmatov, F. Loulergue
• Formal Verification of a Memory Allocation Module of Contiki with Frama-C: a Case Study. F. Mangano, S. Duquennoy, N. Kosmatov
AstraVer Project

• Verification of a closed-source access control system
• Size of code < 10,000 SLOC
• Constant development of code
  • Started around 2014
  • Need to maintain ACSL specifications
  • Rewrote all specifications 3x times by now

• Deductive Verification of Unmodified Linux Kernel Library Functions. Efremov D., Mandrykin M., Khoroshilov A.
Our General Approach

• Store specifications next to the code
  • Developers could benefit from specifications
  • Store verification results of a previous run for Frama-C/WP
  • Store verification sessions for Frama-C/AstraVer(Jessie)

• For every modified function (or for all verified functions)
  1. Extricate it from the sources with all dependencies and specifications
  2. Patch the extracted code to obtain the version ready for verification
  3. Replay the verification
     • Compare results with existing sessions or previous results
Step 1. Extricate. Motivation (1)

Size of code

• Unsupported features of the toolset:
  • Blocks parsing: int128, asm goto, __builtin*, zero-size arrays, ...
• Source code size:
  • Module size: < 10 KSLOC
  • Headers from the kernel: + 400 KSLOC (less than 100 KSLOC is relevant)
  • It takes ~20 minutes for the tools to start and generate proof obligations
• Different functions can use different settings for the verification, e.g. –wp-model ‘Typed+Cast’ instead of the default model
Step 1. Extricate. Motivation (2)
Size of a verification task

• Other functions may force the verification tools to include additional theories to verification tasks
  • A single bitwise operation from other function may lead to the inclusion of bitwise definitions to verification tasks
• “Unrelated” global definitions also extend verification tasks
• Sometimes it is possible to fully prove functions one by one, but it is hard to achieve the same for them together
Step 1. Extricate. Implementation

SELinux Callgraph

Extricate

sel_netport_sid_slow function

GitHub: https://github.com/evdenis/spec-utils
Step 1. Extricate. The example

```c
struct S1 { int a; int b; }
struct S2 { struct S1 *s; ... }
int func1(int a, int b) {
    ...
}
int func2(struct S1 *s) {
    func1(s->a, s->b);
}
int func3(struct S2 *s) {
    func1(...);
    func2(...);
}
```

Extricate func2
Step 2. Patch. Motivation (1)

• Not a mandatory step
• Verified Code != Original Code & Verifiers != Developers
  • verification toolset is not able to handle a code pattern
  • verification toolset does not support some verification features for now
  • verification driven refactoring
  • ...

• Need to track the differences between a verified version and the original one
• Temporary step before either developers will accept the changes or verification toolchain will be improved

• A set of patches allows one to precisely track the issues and keep the same sources for the development and the verification
  • Don’t need to resolve merge conflicts with specifications (prevents automation) or backport the patches
Step 2. Patch. Motivation (2)

Developers <-> Verification Engineers

C'est la vie:

- Developers: design, write, test, release code
- Specifiers: do formal methods stuff
- Specifications are separate from code
- Nothing

Dreams for the future:

The idea of the slide was borrowed from David R. Cok presentation
Step 2. Patch. Implementation

• ACSL specifications from a verified version
  • Committed to the repository to the mainline development branch
  • Without modifications of the code

• In case the verified version of code differs
  • The modifications are local enough
  • Stable enough against development updates
Step 2. Patch. The example (1)

```c
static void set_key(...) {
<...
-
        memcpy(round_keys[0],key,AES_128_KEY_LENGTH);
+ for(i = 0; i < AES_128_KEY_LENGTH; i++) {
+    round_keys[0][i] = key[i];
+ }
...
}
```

- The set_key function from Contiki-NG os/lib/ccm-star.c
- Verified version differs from original one by “inlining” the memcpy function
- Frama-C fails to reason about non-modified version
- Developers will not accept this change
Step 2. Patch. The example (2)

```
expression E;

- E << 2
+ E * 4

@set_key@

- AES_128.set_key
+ set_key
```

- Simple patch for replacing bitwise shift
- Not easy to convince the developers to get rid of it
  - They tend to think this code looks smarter when they use it
  - Makes Frama-C/WP cry
- Function pointer
- Doesn’t supported by Frama-C for now
- Can be replaced by the direct call
Step 2. Patch. The example (3)

- void * list_tail(list_t list)
+ struct list * list_tail(list_t list)
{
+   int n;
    ...
- for(l = *list; l->next != NULL; l = l->next);
+ for(l = *list; l->next != NULL; l = l->next)
{
    //@ assert valid(l);
+   //@ assert 0 <= n < \length(to_ll(*list, NULL))-1;
+   ++n;
+ }
+ }
    ...
}

• The list_tail function from Contiki-NG os/lib/list.c
• Replace “void *” with a concrete type
• Introduce additional local variable “n”
• Add body for the “for” loop
• Ghost expression for a loop body is not currently supported by Frama-C
Step 2. Patch. The example (4). Fail

The Original Code

```c
void list_remove(list_t list, void *item) {
    struct list *l, *r;
    if(*list == NULL) { return; }
    r = NULL;
    for(l = *list; l != NULL; l = l->next) {
        if(l == item) {
            if(r == NULL) {
                *list = l->next;
            }
            else {
                r->next = l->next;
            }
            l->next = NULL;
            return;
        }
        r = l;
    }
}
```

The Verified Code

```c
void list_remove(list_t list, struct list *item) {
    if(*list == NULL) { return; }
    if(*list == item) {
        *list = (*list)->next;
        return;
    }
    struct list *l = *list;
    int n = 0;
    while(l->next != item && l->next != NULL){
        l = l->next;
        ++n;
    }
    if(l->next == item){
        l->next = l->next->next;
    } else {
```

```c
    
```
Step 3. Replay. Implementation

• Frama-C/WP doesn’t support sessions for now
  • One needs to store the results of a previous run

• Check for results downgrade
  • Could be due to a code change by developers
  • Could be due to a global logical definitions change
  • Could be due to a verification toolchain update
  • Could be due to a server heavy load with other tasks (flickering)

• Frama-C/Jessie/Why3 replay
• Frama-C/WP run
Results

• Contiki-NG - https://github.com/evdenis/Contiki-NG
  • Extrication + Semantic patches, 50 functions
  • Replay based on a previous run

• AstraVer
  • Extrication
  • Tens of thousands verification conditions, replay takes about 6-7 hours
  • Replay based on sessions and why3 strategies

• ACSL-By-Example - https://github.com/fraunhoferfokus/acsl-by-example
  • Replay based on a previous run

• VerKer - https://github.com/evdenis/verker
  • Replay based on sessions
Questions?
How do we manage specifications

• We store specifications next to the code
  • Separate header files for axiomatizations (e.g., predicates, lemmas, logic functions)
  • Contracts for functions in headers files
  • Assertions and invariants in a body of a function
  • Approximately 2.6 lines of specification for a single line of code

• We believe that a developer could benefit from specifications
  • Even write a simple precondition
  • At least he can update a code without touching specifications