Filtering equivalent changes from dependency updates with CBMC

Jonas Mårtensson

Blekinge Institute of Technology
& Tutus Data AB

October 25, 2022

Background

- OSS dependencies have become ubiquitous
- Regressions are a core concern for dependency updates [1]
- Countermeasures:
  1. Regression testing
  2. Change impact analysis (CIA)

- Refining impact assessments
  - Equivalence analysis
  - C Bounded Model Checker (CBMC) [2]

Purpose

”To investigate the relevance of CBMC based equivalence analysis in relation to CIA for dependency updates”

Motivation:

- Extensive use of C internally at Tutus
- Research gap

Related work

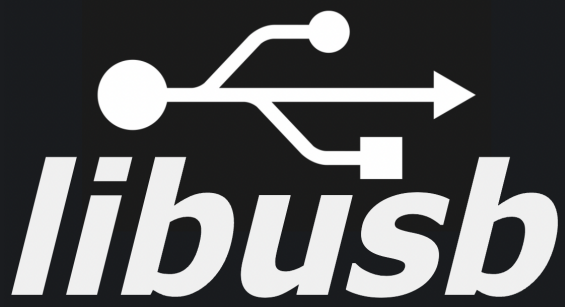
- *Regression verification: proving the equivalence of similar programs (RVT)* [3]
- *Proving Functional Equivalence of Two AES Implementations Using Bounded Model Checking* [4]

- *Can we trust tests to automate dependency updates? A case study of Java Projects* [5]

Contribution

- Equivalent Update Filter (EUF)
  - CIA tool for dependency updates
  - Can auto-generate verification programs

- Example use case:



libusb_release_interface() harness

```c
#include "libusb/libusbi.h"
#include "config.h"
#include "libusb/libusb.h"
#include "libusb/os/events_posix.h"
#include "libusb/os/threads_posix.h"
#include "libusb/version.h"
#include "libusb/version_nano.h"

struct libusb_device_handle nondet_libusb_device_handle();
int nondet_int();

int libusb_release_interface_old_b026324c6904b2a(struct libusb_device_handle* dev_handle, int interface_number);
int libusb_release_interface(struct libusb_device_handle* dev_handle, int interface_number);

void euf_main() {
  struct libusb_device_handle* dev_handle;
  *dev_handle = nondet_libusb_device_handle();
  int interface_number = nondet_int();

  int ret_old = libusb_release_interface_old_b026324c6904b2a(dev_handle, interface_number);
  int ret = libusb_release_interface(dev_handle, interface_number);

  __CPROVER_assert(ret_old == ret, "Equivalent output");

}
```

libusb_release_interface(): 26b16 → 15bd8

```
int API_EXPORTED libusb_release_interface(libusb_device_handle *dev_handle,
  int interface_number)
{
  int r;

  usbi_dbg("interface %d", interface_number);
  if (interface_number >= USB_MAXINTERFACES)
    return LIBUSB_ERROR_INVALID_PARAM;

  usbi_mutex_lock(&dev_handle->lock);
  if (!(dev_handle->claimed_interfaces & (1U << interface_number))) {
    r = LIBUSB_ERROR_NOT_FOUND;
    goto out;
  }

  r = usbi_backend.release_interface(dev_handle, interface_number);
  if (r == 0)
    dev_handle->claimed_interfaces &= ~(1U << interface_number);

out:
  usbi_mutex_unlock(&dev_handle->lock);
  return r;
}
```

```
int API_EXPORTED libusb_release_interface(libusb_device_handle *dev_handle,
  int interface_number)
{
  int r;

  usbi_dbg("interface %d", interface_number);
  if (interface_number < 0 || interface_number >= USB_MAXINTERFACES)
    return LIBUSB_ERROR_INVALID_PARAM;

  usbi_mutex_lock(&dev_handle->lock);
  if (!(dev_handle->claimed_interfaces & (1U << interface_number))) {
    r = LIBUSB_ERROR_NOT_FOUND;
    goto out;
  }

  r = usbi_backend.release_interface(dev_handle, interface_number);
  if (r == 0)
    dev_handle->claimed_interfaces &= ~(1U << interface_number);

out:
  usbi_mutex_unlock(&dev_handle->lock);
  return r;
}
```

libusb_release_interface(): b0fd4 → 6cae9

```
int API_EXPORTED libusb_release_interface(libusb_device_handle *dev_handle,
  int interface_number)
{
  int r;

  usbi_dbg("interface %d", interface_number);
  if (interface_number < 0 || interface_number >= USB_MAXINTERFACES)
    return LIBUSB_ERROR_INVALID_PARAM;

  usbi_mutex_lock(&dev_handle->lock);
  if (!(dev_handle->claimed_interfaces & (1U << interface_number))) {
    r = LIBUSB_ERROR_NOT_FOUND;
    goto out;
  }

  r = usbi_backend.release_interface(dev_handle, (uint8_t)interface_number);
  if (r == 0)
    dev_handle->claimed_interfaces &= ~(1U << interface_number);

out:
  usbi_mutex_unlock(&dev_handle->lock);
  return r;
}
```

```
int API_EXPORTED libusb_release_interface(libusb_device_handle *dev_handle,
  int interface_number)
{
  int r;

  usbi_dbg(HANDLE_CTX(dev_handle), "interface %d", interface_number);
  if (interface_number < 0 || interface_number >= USB_MAXINTERFACES)
    return LIBUSB_ERROR_INVALID_PARAM;

  usbi_mutex_lock(&dev_handle->lock);
  if (!(dev_handle->claimed_interfaces & (1U << interface_number))) {
    r = LIBUSB_ERROR_NOT_FOUND;
    goto out;
  }

  r = usbi_backend.release_interface(dev_handle, (uint8_t)interface_number);
  if (r == 0)
    dev_handle->claimed_interfaces &= ~(1U << interface_number);

out:
  usbi_mutex_unlock(&dev_handle->lock);
  return r;
}
```

Research questions

1. RQ1: How effective is CBMC based equivalence analysis at reducing impact sets?
2. RQ2: To what extent are auto-generated verification programs useful?
3. RQ3: Correctness of classifications?

Method

- Experiment with 360 random updates to assess (RQ1) and (RQ2)
  - jq $\rightarrow$ libonig
  - jabberd $\rightarrow$ libexpat
  - airspy $\rightarrow$ libusb
- Manual inspection of functions with a “multi” result to assess (RQ3)

RQ1. Reduction capabilities

- Maximum observed reductions:
  - Change set: 35 % (libusb & libonig)
  - Impact set: 17 % (airspy)

	Change set Δ	Impact set Δ
libonig	0.05 ±0.06	0.0 ±0.0
libexpat	0.01 ±0.04	0.0 ±0.0
libusb	0.07 ±0.09	0.02 ±0.05
Σ	0.04 ±0.07	0.01 ±0.03

RQ2. Usability of harnesses

- “Usable” harnesses were required to pass:
  1. 12 preconditions
  2. Identity verification

	Valid preconditions	Passed identity
libonig	172/272 (0.63)	33/272 (0.12)
libexpat	23/53 (0.43)	2/53 (0.04)
libusb	83/147 (0.56)	23/147 (0.16)
Σ	0.54 $\pm$0.1	0.11 $\pm$0.06

RQ3. Correctness analysis

Library	Function	Non-equivalent	Equivalent
libonig	renumber_node_backref	TP	TN
libonig	onig_get_content_of_callout_args	FP	TN
libonig	onig_get_content_end_of_callout_args	FP	TN
libonig	onig_get_start_of_callout_args	FP	TN
libonig	onig_get_current_of_callout_args	FP	TN
libonig	onig_get_regex_of_callout_args	FP	TN
libonig	subexp_recursive_check_trav	TP	TN
libusb	parse_endpoint	~	TN

- Main finding: struct changes can cause FPs

Closing remarks

1. Non-negligible reductions, 1 % impact sets, 4 % change sets
2. Limited usability of harnesses, 11 % on average
3. Accuracy of classifications: 67 %

- Follow up studies:
  - Methods for limiting CBMC’s execution time
  - Expanded correctness analysis

References I

[1] Raula Gaikovina Kula et al. “Do developers update their library dependencies?” In: *Empirical Software Engineering* 23.1 (2018), pp. 384–417.

[2] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. ISBN: 3-540-21299-X.

[3] Benny Godlin and Ofer Strichman. “Regression verification: proving the equivalence of similar programs”. In: *Software Testing, Verification and Reliability* 23.3 (2013), pp. 241–258.

[4] Hendrik Post and Carsten Sinz. “Proving Functional Equivalence of Two AES Implementations Using Bounded Model Checking”. In: Apr. 2009, pp. 31–40. DOI: 10.1109/ICST.2009.39.

[5] Joseph Hejderup and Georgios Gousios. “Can we trust tests to automate dependency updates? A case study of Java Projects”. In: *Journal of Systems and Software* 183 (2022), p. 111097. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.111097. URL: https://www.sciencedirect.com/science/article/pii/S0164121221001941.