Case Study: AbsInt + BMC

This is a short explanation for the VMCAI paper.

AWS-C 99 Repos

We use several APIs made for AWS SDK.

- AWS C Common: contains methods for several data structures used for other AWS C packages.
- AWS C Cal: Crypto algorithms, such as rsa, sha1.
- AWS C Compression: Huffman encoding/decoding.
- AWS C IO: The AWS SDK provides support for I/O and TLS for application protocols.
- AWS C SDKutils: Some SDK utility functions.

How we write unit proofs (proof harnesses)

We basically follow the unit test made by each repo, and then follow specs mentioned in the header for writing assumptions and assertions. For the paper, we ignore assertions to prove properties but instead focus on proving memory safety checks.

E.g. the API `aws_byte_buf_append` (aws-c-common/source/byte_buf.c) from aws byte buffer.

```
/**
 * Copies from \p from to \p to. If \p to is too small,
 * AWS_ERROR_DEST_COPY_TOO_SMALL will be returned.
 * dest->len will contain the amount of data actually copied to dest.
 *
 * \p from and \p to may be the same buffer, permitting copying a buffer into itself.
 */
AWS_COMMON_API
int aws_byte_buf_append(struct aws_byte_buf *to, const struct aws_byte_cursor *from);
```

The unit proof `aws_byte_buf_append_harness.c` (seahorn/jobs/byte_buf_append/aws_byte_buf_append_harness.c) includes:

```
/* data structure */
struct aws_byte_buf to;
initialize_byte_buf(&to);

struct aws_byte_cursor from;
initialize_byte_cursor(&from);

/* assume preconditions */
assume(aws_byte_buf_is_valid(&to));
assume(aws_byte_cursor_is_valid(&from));

/* perform operation under verification */
if (aws_byte_buf_append(&to, &from) == AWS_OP_SUCCESS) {
    sassert(to.len == to_old.len + from.len);
} else {
    /* if the operation return an error, to must not change */
    assert_bytes_match(to_old.buffer, to.buffer, to.len);
    sassert(to_old.len == to.len);
}

/* assertions */
sassert(aws_byte_buf_is_valid(&to));
...
```

The proof includes: initialization step for necessary memory objects based on required data structure. Make assumption for a valid objects. Perform calls for function under verification, and check necessary assertions as what specification required.

How we adapt proofs for the case study

Havoc APIs

For some APIs in aws c projects, they will invoke low level libraries such as openssl. Verifying this task is out of scope for us, as we focusing on verifying code written from the AWS team. Thus, we havoc those function as no-ops. That is why we provide some patches in `patch/seabmc_<repo_name>.patch`. This have already applied during docker build process through shell script `scripts/pull_aws_repo.sh`.

Additional assumptions

We also modify the original helper function to avoid allocation failure or introduce disjunctive invariants based on program paths:

- Case 1: assume `malloc` not fail.

```
void *sea_malloc(size_t sz) {
  #ifdef __CRAB__
  assume(sz > 0);
  return malloc(sz);
  #endif
  return nd_malloc_is_fail() ? NULL : malloc(sz);
}
```

We assume allocation not failed, as it will introduce disjuctive invariants for pointers. We also assume each allocation call with non zero size.

- Case 2: assume the pointer is not `NULL` when unit proof calls an API.

```
...
#ifdef __CRAB__
bool nondet_parameter_a = true;
bool nondet_parameter_b = true;
#endif
if (aws_array_list_comparator_string(nondet_parameter_a ? &str_a : NULL,
nondet_parameter_b ? &str_b : NULL) == 0) {
    ...
}
```

We assume the input objects for each AWS API are allocated and fully initialized.