Artifact: Gillian, Part II - Real World Verification for JavaScript and C

Welcome to the artifact README for the CAV 2021 paper: "Gillian, Part II: Real-World Verification for JavaScript and C". In this document, we describe how to use this artifact to reproduce the results presented in the paper.

Table of contents

Reproducing the results

To make the task simple, we created a `Makefile` that lets you run everything. In particular, in the root of the `Gillian` folder, running:

- `$ make c` will verify the correctness of all of the functions that were specified in the C aws-encryption-sdk. The three most important ones are: `aws_cryptosdk_enc_ctx_deserialize`, `parse_edk`, and the main deserialisation function, `aws_cryptosdk_hdr_parse`. This verification takes approximately 14 minutes on a machine with an Intel Core i7-4980HQ CPU 2.80 GHz, DDR3 RAM 16GB, and a 256GB solid-state hard-drive, running macOS.
- `$ make js` will verify the correctness of all of the functions that were specified in the JS aws-encryption-sdk. The three most important ones are: `decodeEncryptionContext`, `deserializeEncryptedDataKeys`, and the main deserialisation function, `deserializeMessageHeader`. This verification takes approximately 45 seconds on the same machine

as above.

- `$ make c-proc PROC=function_name` will run the verification of only the C function whose identifier is `function_name`, if it has a specification. For example, `$ make c-proc PROC=parse_edk` verifies the `parse_edk` function.
- `$ make c-lemma LEMMA=lemma_name` will run the proof of the C lemma whose identifier is `lemma_name`, if such a lemma exists.

In addition to this, there are three rules that allow you to reproduce the bugs that we found in the C code:

```
$ make c-byte-cursor-ub
$ make c-string-bug
$ make c-header-bug
```

The first rule will trigger an undefined behaviour (details here); the second an over-allocation (details here); and the third a logical error in the parsing of the header (details here).

Finally, there are two rules for reproducing the bugs found in the JavaScript code:

```
$ make js-proto-bug
$ make js-frozen-bug
```

The first rule will trigger the prototype poisoning bug (details here); and the second the bug in which the encryption context is returned non-frozen (details here). These will be a bit slower to execute as they will be producing the appropriate logs.

Gillian in more detail

What is Gillian telling us?

Gillian execution consists of several phases, and Gillian prints out some basic information to the terminal about each of them.

- First, the code is parsed and compiled. The Gillian instantiation provides information on how to compile the program to GIL, by implementing a signature called `ParserAndCompiler.S`, declared in `GillianCore/parserAndCompiler/ParserAndCompiler.ml`. During this phase, Gillian simply prints `Parsing and Compiling...`.
- Next, Gillian performs some preprocessing on the logic annotations of the analysed program. For example, it will inline definitions of any non-recursive predicate in the specifications pre-conditions and post-conditions, and normalise function pre-conditions. This phase is normally fast, but can get slower when the analysed predicates are complex (which is the case for the predicates defined in the `header.c` file of the AWS SDK C implementation). During this phase, Gillian prints `Preprocessing...`
- Once preprocessing is done, Gillian will start collecting the tests. Each test corresponds to one function specification that is to be verified or to one lemma that is to be proven. This phase is usually very fast During this phase, Gillian will print the following:

```
Obtaining specs to verify...
Obtaining lemmas to verify...
Obtained X symbolic tests in total
```

together with the time spent in these three first phases.

- Next, Gillian starts running the obtained symbolic tests one by one, indicating which one is currently being proven. The execution may branch, and for each branch taken, Gillian will print:
  - `n` when the end of the function has been reached in "normal mode";
  - `e` when the end of the function has a been reached in "error mode". Error mode does not exist in C and is used to model the way functions end when using `throw` in JavaScript.

  After this, Gillian analyses the results obtained for each of the branchses, printing:
  - `s` when the execution of a branch ends normally and the final state is unified successfully against the specified post-condition;
  - `f` when the execution of a branch has failed early or has ended normally but did not unify successfully against the post condition.

  Finally, after all of this has been done, Gillian prints `Failure` or `Success`, the latter corresponding to every branch finishing as expected and unifying against the desired post-condition.

  Note: It is possible that the Gilian execution fails ungracefully when an error occurs. We are working on improving resilience and user feedback.

Can it be made to say more?

Gillian can provide more detail about the execution by writing varying degrees of details in a log file (`file.log`). That behaviour is controlled by the `-l` command line argument, with options being `normal`, `verbose`, and `disabled`. In the provided Makefiles, every command specifies `-l disabled`, and the default behaviour (no flag) is equivalent to `-l verbose`. Debugging a Gillian execution is currently an "art" with a bus factor of 1.75, which requires a lot of practice.

Since producing the log file is expensive in terms of resource and can produce up to several gigabytes of data, if the reader wants to see what kind of information is contained, we advise running single, simple procedures with the `-l normal` flag.

The command line interface

Gillian is the core framework into which we simply plug information about compilation and memory models for target languages. Therefore, all binaries produced by using the Gillian framework have a similar command-line interface. One can see the details about possible arguments by running the following commands from the Gillian root folder:

```
$ esy x gillian-c --help
$ esy x gillian-js --help
```

The reader will find that both executables expose at least four sub-commands: `compile`, `wpst`, `verify` and `act`:

- `compile` allows one to generate the GIL files from a target-language program;
- `wpst` corresponds to whole-program symbolic testing, presented in our previous paper from PLDI'20, and is out of the scope of this paper;
- `verify` corresponds to full verification (including specifications and abstractions) and *is the only command that is in the scope of this CAV paper*;
- `act` corresponds to automatic compositionnal testing, which uses bi-abduction in the style of Infer, and is out of the scope of this paper.

We give more details about the `verify` command, since it is of interest for this artifact. First of all, the reader is invited to run:

```
$ esy x gillian-c verify --help
```

to obtain the list of possible arguments. We discuss the arguments that are used in the Makefiles.

- `-l`, alias for `--logging` has been detailed earlier
- `--lemma` and `--proc` are used to specify which lemmas or procedures to verify. To run several procs, the flags can be used several times. For example:

  ```
  $ esy x gillian-c verify test.c --proc function_a --proc function_b
  ```

- `--no-lemma-proof` allows for admittance of lemmas without proofs, while still proving lemmas that have proofs. In the case of this artifact, we need to apply lemmas that correspond to parts of the code that have been axiomatised (as explained in the caveats section in the paper).
- (For C only) `--fstruct-passing` is a flag that is passed directly to `CompCert` and is detailed in [its manual](). It enables the following feature: "Support functions that take parameters and return results of composite types (struct or union types) by value".

C: Structure, annotations, bugs

Reading Gillian-C annotations: Predicates

Let us dive into how to read the annotations in C. We recommend that you open the code in VSCode, for which we have developed a syntax highlighting extension that makes annotations more readable.

```
$ code . // In the Gillian root folder
```

We will now walk the reader through a few elements written in `Gillian-C/examples/amazon/byte_buf.c`.

First of all, comments starting with `/*@` are parsed by the Gillian-C parser as logic annotations. Such comments are used to define predicates, lemmas and specifications.

In particular, let us give details about the first predicate that is defined, called `valid_aws_byte_cursor_ptr`. It describes an element of type `struct aws_byte_cursor` on which some constraints have been applied. It can be seen as a form of invariant that should be preserved whenever an element of that type is constructed or modified.

The `struct aws_byte_cursor` type is defined in `Gillian-C/examples/amazon/byte_buf.h` as follows:

```
struct aws_byte_cursor {
    size_t len;
    uint8_t *ptr;
};
```

The reader may recognise that such a cursor is an implementation of an array slice: the cursor contains a pointer `ptr` that can be anywhere inside an array, and a length `len`. It should always be possible to read `len` bytes at address `ptr`. The original AWS code for this structure contain annotations for concrete tests that try to enforce such a behaviour. Every function that uses an aws_byte_cursor ends with a post-condition check of the form

```
AWS_POSTCONDITION(aws_byte_buf_is_valid(cursor))
```

where `aws_byte_buf_is_valid` (and what it depends on) is defined as follows:

```
/**
 * The C runtime does not give a way to check these properties,
 * but we can at least check that the pointer is valid. */
#define AWS_MEM_IS_READABLE(base, len) (((len) == 0) || (base))
bool aws_byte_cursor_is_valid(const struct aws_byte_cursor *cursor) {
    return cursor != NULL &&
           ( (cursor->len == 0) ||
             (cursor->len > 0 && cursor->ptr && AWS_MEM_IS_READABLE(cursor->ptr,
cursor->len))
           );
}
```

This predicate checks that the cursor is not NULL, and that it points to a structure that either has length 0 (and we don't care what's in the pointer because nothing will be read ever), or to something that has length greater than 0 and a non-NULL pointer. The comment about limitations in the C runtime is [also extracted from the AWS code itself](#).

Gillian-C can overcome this limitation by providing the necessary expressivity to talk about what is in memory. The predicate is defined as follows:

```
pred nounfold valid_aws_byte_cursor_ptr(+cur, length, buffer: List, alpha) {
  (cur -> struct aws_byte_cursor { long(0); buffer }) *
  (length == 0) * (alpha == nil);

  (cur -> struct aws_byte_cursor { long(length); buffer }) * (0 <# length) *
  ARRAY(buffer, char, length, alpha) * (length == len alpha) *
  (length <=# MAX_IDX_SIZE)
}
```

It contains two definitions separated by a semi-colon, which can be thought of as disjuncts:

The first definition contains three bits of information, joined by the `separating conjunction`

- First, we have `cur -> struct aws_byte_cursor { long(0); buffer })`, which says that `cur` is a valid pointer to a structure of type `struct aws_byte_cursor`, of which the `len` field has value `long(0)` (the type corresponds to the internal type used by CompCert in x64 mode to interpret `size_t`).
- Next, we have that in this case the length is 0 and the represented list of bytes is `nil`, which is an alias for the empty list. Note that `length` corresponds to the mathematical value representing the length.
- Finally, it says nothing about `buffer`, but the `-> struct xxx` annotation is compiled to a predicate in GIL that captures the layout of the structure in memory, and this predicates does contain the information that buffer is either a pointer or `NULL`.

The second definition is similar to the first one, but specifies the non-empty case:

- First, here, `length`, (the mathematical value contained by the C value in `cur->len`) has to be strictly greater than 0, and strictly smaller than the maximum indexable size.
- Next, `buffer` has to be a valid pointer that points to an array of bytes in memory (the type being denoted by `char`), of size `length` and content `alpha`.

The use of the separating conjunction in Gillian-C is particularly interesting: this predicates does not only describe the shape and value of data in memory, it also gives **ownership** of the array slice to the pointer `cur`. Indeed, while in C it would be possible to have two cursors `cur1` and. `cur2` pointing to the same slice, it would *not* be possible to write the follwing assertion:

```
valid_aws_byte_cusor_ptr(cur1, #b, #alpha) * valid_aws_byte_cursor(cur2, #b, #alpha)
```

This idea of ownership is used extensively in modern languages such as Rust.

Finally, note that predicates in Gillian can have additional specifiers placed in their headers:

- `nounfold` means that, although this predicate is not recursive, it should not be unfolded automatically during preprocessing;
- `pure` means that this predicate does not specify spatial resource, and it can therefore be duplicated (i.e. if the predicate `is_NULL(x)` is marked as pure, Gillian will understand without unfolding it that `is_NULL(x) * is_NULL(x)` is a valid assertion).

Reading Gillian-C annotations: Specifications and Lemmas

In Gillian-C, function specifications are written using the following syntax:

```
[axiomatic?] spec [function_name] ([arg1], ..., [argn]) {
   requires: [Pre: assertion]
   ensures: [Post1: assertion]; [Post2: assertion]; ... [Postn: assertion]
}
```

which defines a specification for `function_name` with the given arguments `arg1, .. argn`, which has precondition `Pre` and several possible post-conditions `Post1` to `Postn`, separated by semi-colons. It is also possible to have several specifications for the same fonction, in which case there will be several `requires/ensures` pairs separated by the keyword `OR`. If the `axiomatic` keyword is added before the `spec` keyword, Gillian will not try to prove the spec. Axiomatisation is used for most `array_list` functions and all `hash_table` functions.

Similarly, in Gillian-C, a lemma is simply a specification for which the proof is not a function, but a list of *logic commands* or *tactics*, outlined shortly. The lemma syntax is as follows:

```
lemma [lemma_name] ([arg1], ..., [argn]) {
   hypothesis: [Pre: assertion]
   conclusions: [Post1: assertion]; [Post2: assertion]; ... [Postn: assertion]
   proof: [logic_commands]
}
```

The proof is optional, since we may not be able to write proofs when lemmas are using axiomatic predicates.

Reading Gillian-C annotations: Tactics

Most verification proofs require human help. For example, it is necessary to write loop invariants or to ask Gillian to unfold a specific predicate. These tactics are just logic commands that will be called in the middle of an execution. If the reader is interested, the type definition corresponding to the logic commands in C (later compiled to logic commands in GIL) is in `Gillian-C/lib/cLogic.ml`:

```
type t =
    | If          of CExpr.t * t list * t list (** Conditional execution of logic
command *)
    | Unfold      of {
        pred : string;
        params : CExpr.t list;
        bindings : (string * string) list option;
        recursive : bool;
      } (** Unfolding of a specific predicate *)
    | Unfold_all of string (** Recursively unfold all predicates with the given name
(with a fuel). *)
    | Fold        of string * CExpr.t list (** Fold a predicate *)
    | Apply       of string * CExpr.t list (** Apply a lemma *)
    | Assert      of CAssert.t * string list
        (** Assert for verification, takes an assertion and binders *)
    | Branch      of CFormula.t (** The symbolic engine should branch on the given
formula *)
    | Invariant   of {
        assertion : CAssert.t;
        bindings : string list;
      } (** Loop invariant *)
    | SymbExec (** Ignore the next function specification and symbolically execute
instead *)
```

Understanding the discovered bugs

We found three bugs in the AWS implementation, and we have already explained earlier how to reproduce those bugs. In this section, for each bug, we explain how it can happen and suggest a fix.

A potential undefined behaviour in aws_byte_cursor_advance

As explained earlier, an `aws_byte_cursor` is a kind of abstraction that lets the user consume a buffer while always knowing how much memory is left. The function `aws_byte_cursor_advance` has the following documentation:

```
/**
 * Tests if the given aws_byte_cursor has at least len bytes remaining. If so,
 * *buf is advanced by len bytes (incrementing ->ptr and decrementing ->len),
 * and an aws_byte_cursor referring to the first len bytes of the original *buf
 * is returned. Otherwise, an aws_byte_cursor with ->ptr = NULL, ->len = 0 is
 * returned.
 *
 * Note that if len is above (SIZE_MAX / 2), this function will also treat it as
 * a buffer overflow, and return NULL without changing *buf.
 */
struct aws_byte_cursor aws_byte_cursor_advance(struct aws_byte_cursor *const cursor,
const size_t len);
```

In the case where the operation is estimated to be valid, the following operation is performed:

```
cursor->ptr += len;
```

The issue is that if `ptr` is NULL, then the tests that checks if "the cursor has at least len bytes remaining" still passes, in which case the operation `NULL + 0` is performed. Although most compilers will not complain, this is technically an undefined behaviour according to the C specification.

There are several ways of fixing this issue. We believe that is is not an implementation error but a documentation/specification issue, i.e. the function documentation should specify that the function should never be called with a cursor that contains a `NULL` pointer if the length passed as argument 0. In all the code we analysed, that pre-condtion was respected. Therefore, we decided to add the following formula to the precondition that we wrote:

```
((0 <# #length) || (not (#buffer == NULL)))
```

Link: https://github.com/awslabs/aws-c-common/issues/771

Over-allocation in aws_string

The AWS standard library for C (called `aws-c-common` and is heavily used in the encryption SDK) defines a type `struct aws_string`. This is an abstraction over raw C strings in order to make them safer to manipulate, and is defined in `string.h` as follows:

```
struct aws_string {
    struct aws_allocator *const allocator;
    const size_t len;
    /* give this a storage specifier for C++ purposes. It will likely be larger after
init. */
    const uint8_t bytes[1];
};
```

It contains :

- an allocator, used if one ever needs to use a customised function to delete the string;
- a length, used to check that one is never reading past the size of the string, and to retrieve the length in O(1); and
- an array of bytes that contains the actual string.

Note that this structure is defined using an ambiguous feature of the last element being a flexible array member. In order to allocate such a structure, one needs to know the size of the array and add it to the size of the remaining elements. In the buggy code, this operation is done in some code that is essentially equivalent to this:

```
struct aws_string *str = malloc(sizeof(struct aws_string) + 1 + length);
```

This is indeed what one would expect: size of the structure + length of the string + 1 byte for the null character `\0` terminating the string. However, because the flexible array member is specified using `bytes[1]`, `sizeof` gives it a size of `1 byte`. Because of alignment constraints in C, the size of the structure is given as `24` instead of `16`. Therefore, every allocated aws_string has 8 bytes too many.

This is quite a complex bug to fix because of the inconsistent behaviour of flexible array members between C and C++, as well as between different versions of C. For verification to pass, we decided on a fix that is correct in most C versions (including CompCert C) but is still ambiguous in C++, and removed the size specifier.

We detected the bug because thes structure created by the constructore could not unify against the the predicate we manually wrote to describe the expected layout in memory.

Link: https://github.com/awslabs/aws-c-common/issues/776

Wrong aad_len can be accepted by hdr_parse.

This issue is probably the most worrisome: there is a bug in the way the `aws_cryptosdk_hdr_parse` function parses the encyption context.

It starts by calling `aws_byte_cursor_advance` to read the length of the encryption context (called `aad_len`). However, it does not check the returned value after that call. As said in the description of the first bug, if there isn't enough data to read the amount requested, the `advance` function will return a cursor containing a `NULL` buffer and a length of 0. Therefore, the function `aws_cryptosdk_enc_ctx_deserialize` will be passed a cursor that has `len = 0`, and will do nothing, returning `AWS_OP_SUCCESS`.

This allows one to create headers that are not well-formed, but could still be parsed correctly, either in their entirety or if supplied partially. More details are given in the Github issue.

Link: https://github.com/aws/aws-encryption-sdk-c/issues/695

JS: structure, annotations, bugs

Relevant folder structure

The instantiation of Gillian to JavaScript (Gillian-JS) can be found in the `Gillian-JS` folder. The folders and files related to the verification of the deserialisation module of the JS implementation of the AWS SDK can be found in the `Examples\Amazon` folder inside `Gillian-JS`, and they are:

- the main files:
  - `deserialize_factory.js`: the **main** main file, containing all of the functions of the deserialisation module and their specifications
  - `AmazonLogic.jsil`: language-dependent predicates and lemmas describing the deserialised AWS header
  - `ByteLogic.jsil`: basic conversion from lists of bytes to various numerics
  - `EncryptionHeaderLogic.jsil`: language-independent predicates and lemmas describing the serialised AWS Message Header
  - `ListLogic.jsil`: predicates and lemmas for advanced list management
  - `Utf8Logic.jsil`: axiomatisation of conversion from bytes to UTF-8 strings
- files inside `bugs\pp`: the main files adapted to reproduce the prototype poisoning bug
- files inside `bugs\frozen`: the main files adapted to reproduce the non-frozen encryption context bug

Understanding the discovered bugs

We found two bugs in the AWS implementation, and we have already explained earlier how to reproduce those bugs. In this section, for each bug, we explain how it can happen and provide a fix.

Prototype poisoning in decodeEncryptionContext

The `decodeEncryptionContext` deserialises the encryption context into a key-value map, initially implemented as a standard JavaScript object, encoding keys as property names and values as property values:

```
var encryptionContext = {};
```

However, given the prototype inheritance mechanism of JavaScript, object property lookup may succeed if the property in question exists further along the prototype chain (say, `"hasOwnProperty"` in `Object.prototype`). This, combined with the runtime correctness check:

```
needs(
  encryptionContext[key] === undefined,
  'decodeEncryptionContext: Duplicate encryption context key value.'
)
```

where the `needs` function is defined as follows

```
function needs(condition, errorMessage) {
  if (!condition) {
    throw new Error(errorMessage)
  }
}
```

results in the function throwing an error if the key coincides with a property of `Object.prototype`.

In the created log file, reachable by typing `open file.log`, line 2781259 says:

```
VERIFICATION FAILURE: Spec decodeEncryptionContext 0 terminated with flag error instead
of normal
```

which can be traced back to the above correctness check using the `needs` function throwing an error. The simplest way to fix this is to create the encryption context object with the prototype `null`:

```
var encryptionContext = Object.create(null)
```

Pull request: https://github.com/aws/aws-encryption-sdk-javascript/pull/216

Authenticated encryption context can be edited by third parties

Again in the `decodeEncryptionContext` function, in the case when the encryption context is empty, the returned object is returned non-frozen:

```
var encryptionContext = Object.create(null)

if (!encodedEncryptionContext.byteLength) {
  /* ERROR: OBJECT NOT FROZEN */
  return encryptionContext
}
```

In the created log file, which is this time at the `verbose` level and is reachable by typing `open file.log`, line 6757449 says:

```
VERIFICATION FAILURE: Spec decodeEncryptionContext 0 - post condition not unifiable
```

which can then be traced back to line 6736798, which says:

```
WARNING: Unify Assertion Failed: (<Cell>(_lvar_3039, "@extensible"; false), ) with
subst
```

revealing that the object is not non-extensible as intended, which is then understood to come from the object not being frozen.

This allows third parties to edit the deserialised encryption context, which constitutes a security breach. As we are not security experts, we cannot estimate the severity of this breach. The fix is to freeze the object before the return statement using

```
Object.freeze(encryptionContext);
```

This bug was communicated to the developers personally and we are expecting it to be fixed soon.

BONUS: Improved implementation of the readElements function

In a nutshell, the implementation of the readElements function was not aligned with the underlying data structure, making its specification and verification very difficult. We suggested an appropriate change; more details are available in the pull request below.

Pull request: https://github.com/aws/aws-encryption-sdk-javascript/pull/215