CS525 : Formal Methods For System Verification

Evaluating Performance of LLMs for Verification of Hardware Designs
Design: HMAC using GPT 3.5

Project Report

Group 1: Akshat Mittal, 200101011
Dhananjai, 200101029
Pranjal Baranwal, 200101083
Satvik Tiwari, 200101091

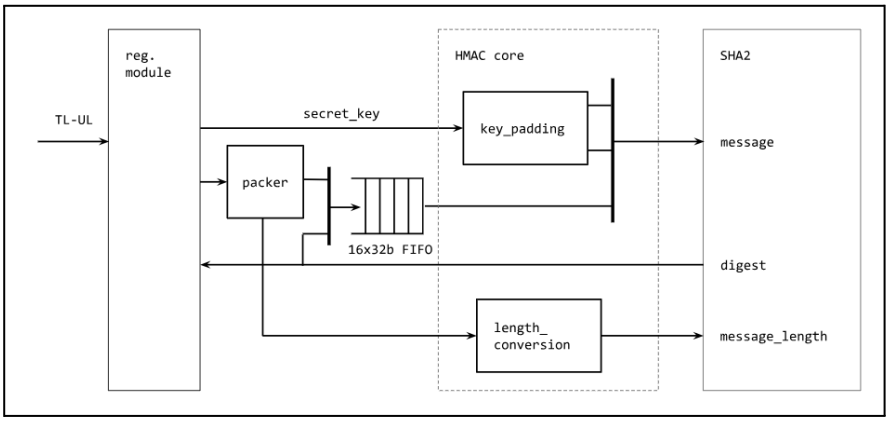
Motivation for the Project

This project aims to utilize the power of Large Language Models (LLMs) to enhance the verification process of hardware designs to reduce the time and resources required for the verification, ultimately accelerating product development and time-to-market while maintaining high quality standards.

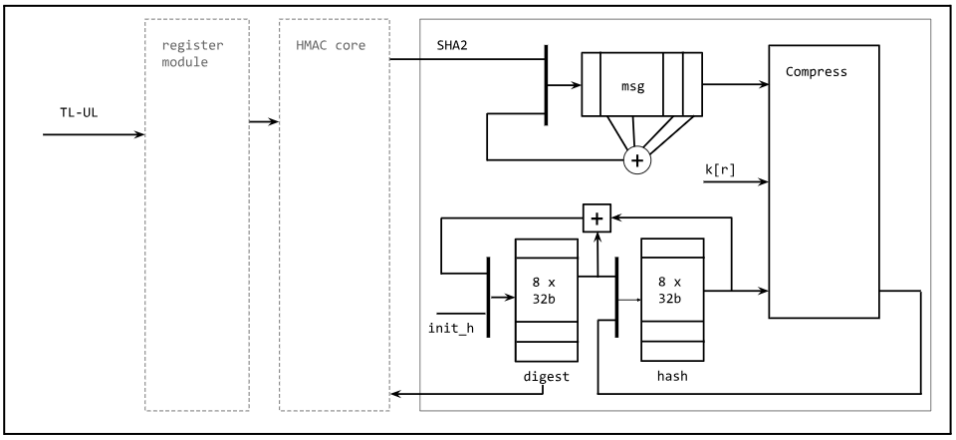
Design Description - HMAC

HMAC is a SHA-256 hash based authentication code generator to check the integrity of an incoming message and a signature signed with the same secret key. It generates a different authentication code with the same message if the secret key is different.

- The 256-bit secret key is written in *KEY_0* to *KEY_7*.
- The message to authenticate is written to *MSG_FIFO*.
- The HMAC generates a 256-bit digest value which can be read from *DIGEST_0* to *DIGEST_7*.
- The *hash_done* interrupt is raised to report to software that the final digest is available.

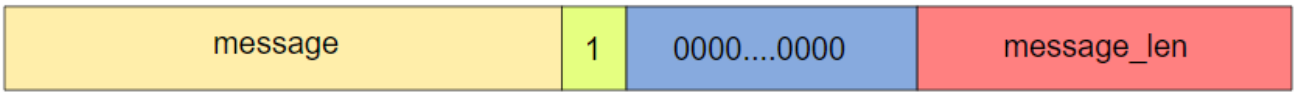


- HMAC core converts the secret key registers into an inner padded key and an outer padded key which are fed to the hash engine.
- It also feeds the result of the first round message (which uses the inner padded key) from the SHA-256 hash engine into the 16x32b FIFO for the second round (which uses the outer padded key).



SHA-256 Message Feed and Pad:

- A message is fed via a memory-mapped message FIFO.
- The logic assumes the input message is little-endian. It converts the byte order of the word right before writing to SHA2 storage as SHA2 treats the incoming message as big-endian.
- The SHA-256 module computes an intermediate hash for every 512-bit block.
- The message must be padded to fill 512-bit blocks. This is done with an initial **1** bit after the message bits with a 64-bit message length at the end and enough **0** bits in the middle to result in a full block.



SHA-256 Computation:

- The SHA-256 engine receives 16 32-bit words from the message FIFO or the HMAC core then begins 64 rounds of the hash computation which is also called compression.
- In each round, the compression function fetches 32 bits from the buffer and computes the internal variables.
- The first 16 rounds are fed by the words from the message FIFO or the HMAC core. Input for later rounds comes from shuffling the given 512-bit block.

HMAC Computation:

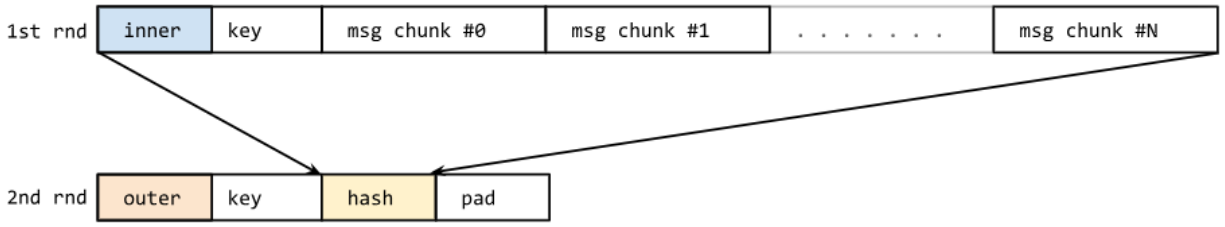
- The first phase of HMAC calculates the SHA-256 hash of the inner secret key concatenated with the actual message to be authenticated. The message length used in the SHA-256 module is calculated by the HMAC core by adding 512 to the original message length.

```
inner_pad_key = {key[255:0], 256'h0} ^ {64{8'h36}} // big-endian
```

- The first round digest is fed into the second round in HMAC. The second round computes the hash of the outer secret key concatenated with the first round digest. In the second round, the message length is a fixed 768 bits.

```
outer_pad_key = {key[255:0], 256'h0} ^ {64{8'h5c}} // big-endian
```

- HMAC assumes the secret key is 256-bit. The onus is on software to shrink the key to 256-bit using a hash function when setting up the HMAC.



Assertions (given in OpenTitan)

The code of the HMAC module having assertions can be found here.
4 assertions were listed in the code, and they are given below:

1. *msg_fifo_req → !in_process*
2. *reg_hash_process → initiated*
3. *!in_process && !initiated → $stable(message_length)*
4. *hmac_en != $past(hmac_en) → !in_process && !initiated*

Assertions generated by LLM from Design Specification

We communicated with GPT-3.5 and details of the prompt can be found here.
The main prompt, excluding the design description that we used is given below:

> Assume you are world's best system verification engineer, given below is a description for a design inside ```. Understand the design thoroughly in and out.
> Generate LTL properties for the design with correct specifications. Here note that you should only use input and output variables in a property. Do not use any intermediate variables.
> For each property that you generate, assign a suitable name to it. Also explain the property and give the equivalent SVA code for it.
> Make sure to consider every aspect of design and only generate relevant ones.
> Generate as many properties as possible.
> Give Name, LTL, Description and SVA code for each.

10 LTL properties that were generated by LLM are as follows:

1. *G((msg_fifo_req && hmac_en) → F(reg_hash_start))*
2. *G((reg_hash_start && reg_hash_done) → F(!reg_hash_process))*
3. *G(fifo_full → !fifo_empty)*
4. *G((hmac_en && msg_fifo_req) → F(message_length' > message_length))*
5. *G((msg_fifo_req && hmac_en) → F(reg_hash_done))*
6. *G(msg_fifo_req → F(hmac_en))*
7. *G(hmac_en → F(msg_fifo_req))*
8. *G(reg_hash_done → F(reg_hash_start))*
9. *G(hmac_en → F(wipe_secret))*
10. *G((wipe_secret && reg_hash_start) → X(secret_key' != secret_key))*

How do we converge on good assertions (Strategy used):

To converge to these properties we only gave LLM the whole description of the design. Along with the description we gave it only the important registers and it was able to generate the results on first try. We saw that it was making more complex and unnecessary assertions if we were asking him to check whether the generated assertions have any inconsistency or not.

Assertions (in point 3) that LLM could not generate: List them

LLM was able to generate 3/4 assertions.

The only assertion it could not generate was the 3rd assertion (of point 3) i.e. *!in_process && !initiated → $stable(message_length)*.

 # 2nd assertion(LLM) was able to cover the 2nd assertion (point 3):

From code we got:

reg_hash_process -> !in_process and *reg_hash_done -> in_process*

reg_hash_start -> !initiated and *reg_hash_process -> initiated* (Assertion 2 Opentitan)

From LLM we are getting :

G((reg_hash_start && reg_hash_done) → F(!reg_hash_process))

reg_hash_start -> !initiated and *reg_hash_done ->in_process* (From code)

!reg_hash_process = !(!in_process & initiated) = in_process or !initiated

So, the 2nd LLM assertion is logically correct.

 # Similarly the 1st assertion(LLM) was able to cover 1st and 4th assertion (point3):

From code we got:

reg_hash_process -> !in_process and *reg_hash_done -> in_process*

reg_hash_start -> !initiated and *reg_hash_process -> initiated* (Assertion 2 Opentitan)

From LLM we are getting :

G((msg_fifo_req && hmac_en) → F(reg_hash_start))

msg_fifo_req -> ! initiated (Assertion 1 Opentitan)

hmac_en -> !initiated & !in_process (Assertion 4 Opentitan)

reg_hash_start -> !initiated (from code)

So, it is covering the 1st and 4th assertion of opentitan.

Does LLM provide new important assertions that was not there in Point 3? List them

Yes, the LLM generated 9th and 10th assertions are important from the point of view of security. The property asks to wipe the secret after the hash process is finished to prevent any data leaks and encryption key misuse. This is a very important step to verify as security key leaks will defy the whole purpose of HMAC encryption algorithm to provide security.

Are the assertions generated by LLM are consistent?

Most of the assertions are consistent but assertion 4 and assertion 5 are inconsistent with the whole design.

 # How assertion 4 is wrong

From code we got:

reg_hash_process -> !in_process and *reg_hash_done -> in_process*

reg_hash_start -> !initiated and *reg_hash_process -> initiated* (Assertion 2 Opentitan)

From LLM we are getting:

G((hmac_en && msg_fifo_req) → F(message_length' > message_length))

We already saw that *(msg_fifo_req && hmac_en) -> !in_process* and *!initiated -> stable message length* (Assertion 3 opentitan). But our LLM is saying unstable length. Hence it is a wrong assertion.

Completeness of the assertions generated by LLM?

The assertions generated by LLM are not complete. Though it tried to cover the 3rd assertion which was about the change in message length but could not cover it correctly. Overall we can't say concretely about the completeness of the assertions.

Verification time by EDA Playground of the LLM generated assertions. Did you find any bug/inconsistency in the design?
Due to errors in import, we could not run the code in EDA Playground. The link for the playground is given here.

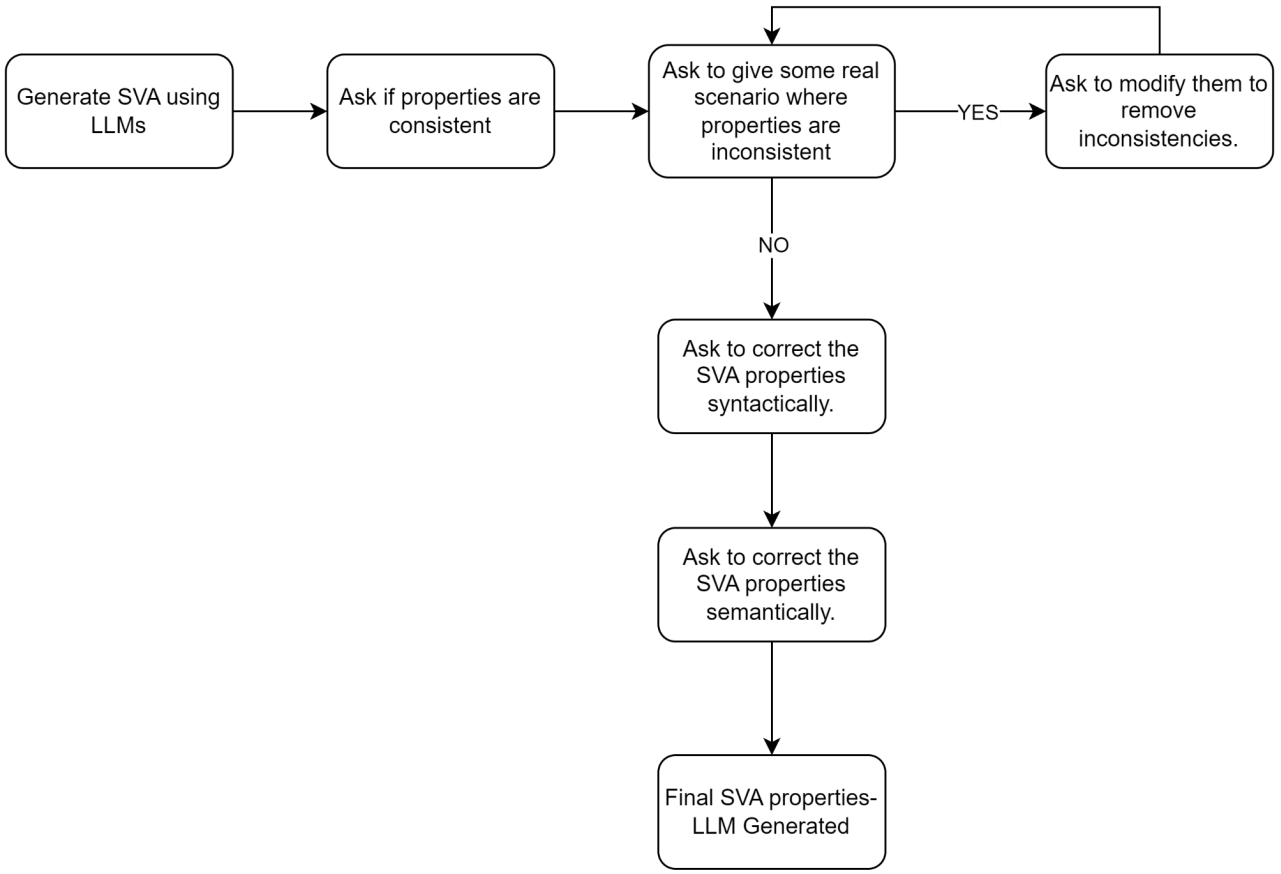
Assertions generated by LLM from Design Implementation in Verilog

We communicated with GPT-3.5 and details of the prompt can be found here.
The model generated 4 SVA assertion properties, given as follows:

1. hash_start → (fall(wipe_secret))
2. $fell(wipe_secret) → $changed(hash_start)
3. reg_hash_process → $changed(message_length)
4. $changed(hmac_en) → !in_process && !initiated

Strategy to generate SVA Assertions using LLM:

Below is the flowchart of the strategy we used to generate assertions and converge to good assertions.



We used the prompt given in research paper to improve the properties syntactically and semantically.

Observations:

One thing we realised is that our LLM generated SVA assertions cover 3 out of 4 actual SVA assertions.

- The last assertion of both our LLM and code is the same.
  LLM Assertion: $changed(hmac_en) |→ !in_process && !initiated
  Code Assertion: hmac_en != $past(hmac_en) |→ !in_process && !initiated
- The 2nd and 3rd assertion of code is covered in the 3rd property generated by the SVA.
  LLM 3rd Assertion: reg_hash_process |→ $changed(message_length)
  Code 2nd Assertion: reg_hash_process |→ initiated
  Code 3rd Assertion: !in_process && !initiated |→ $stable(message_length)
  Using the codes 2nd and 3rd assertion, we can conclude that if reg_hash_process is triggered,

then it will lead to non-stable message length, i.e. changed message length, and this is covered in LLMs 3rd assertion.

- The 1st assertion of code is not covered by LLM.
- The 1st and 2nd assertions generated by LLM are not given in code. These assertions are important for security reasons. The property asks to wipe the secret after the hash process is finished to prevent any data leaks and encryption key misuse.
  LLM 1st Assertion: hash_start |→ fall(wipe_secret)
  LLM 2nd Assertion: $fell(wipe_secret) |→ $changed(hash_start)

Discussions:

1. **How good the LLMs in generating correctness assertions in English language, LTL and SVA from specification?**
   LLMs are good in generating text based responses and correctness properties in plain english language but their ability to generate LTL and SVA from specifications are slightly limited. They do not give highly precise and accurate results in such cases. The design specification may not fully cover the idea of implementation and thus LLM may not be able to understand it clearly and give some random stuff which is not required. In our point of view, in order to generate some good assertions and LTL properties, we need to describe the design thoroughly along with precisely explaining the LLM what we need. Further, we sometimes also need to give the correct way of writing SVA assertions as they are not specifically trained for these things.

2. **How good the LLMs in generating correctness assertions in English language, LTL and SVA from the implementation in Verilog?**
   From the verilog implementation, the required things to generate assertions are more clearly available and LLMs are able to perform better when the verilog implementation is directly given to them with proper prompt. LLMs are able to generate better properties with implementation as compared to design specification with use of proper variables. But the point that they give the SVA fully correct syntactically is still valid.

3. **Do you suggest to use LLM to generate properties in English language/LTL/SVA?**
   Using LLMs, we suggest to generate the verification properties in plain english language as they are better in generating human-like text based responses as compared to more specific LTL/SVA as these require some deep understanding and domain-specific knowledge. LLMs, are excellent at natural language understanding and generation but they do not not possess the expertise to generate highly precise and accurate assertions without specific training.

4. **Your overall recommendation on the use of LLM in design verification?**
   LLMs are powerful things. We surely recommend to use them for design verification purposes. They can easily give a good headstart on a new design anf thus, boost the overall efficiency of a design verification engineer. However, completely relying on them is not a good idea as some of the properties generated is unnecessary and sometimes even inconsistent.