Applying GPT-4 to SAW Formal Verification

WEDNESDAY, AUGUST 9, 2023
ARTIFICIAL INTELLIGENCE (HTTPS://GALOIS.COM/BLOG/CATEGORY/ARTIFICIAL-INTELLIGENCE/), MACHINE LEARNING (HTTPS://GALOIS.COM/BLOG/CATEGORY/MACHINE-LEARNING/), SAW (HTTPS://GALOIS.COM/BLOG/CATEGORY/TOOLS/SAW/)

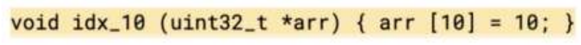
Adam Karvonen (https://galois.com/blog/author/adam-karvonen/)

There has been a lot of chatter recently about large language models, including GPT-4 and LLaMa. At Galois, we have been experimenting with GPT-4, the most capable available large language model. One group of intriguing results that I am excited to present is in the creation of SAWScript memory safety proofs. Using a very simple approach, I was able to use GPT-4 to generate a complete and correct memory safety proof for the program salsa20 (https://github.com/GaloisInc/saw-script/blob/master/exercises/memory-safety/salsa20/salsa20.c). With additional work, this method could be extended to produce proofs for more complex programs.

A Brief Overview of SAW Memory Safety Proofs

SAW is a software verification tool that has been used in many government and industry assurance projects. For example, it was recently used to verify AWS LibCrypto, a cryptographic library used in production by Amazon Web Services. SAW is a powerful tool, but applying this power requires the programmer to write a series of *proof scripts*. I wanted to see whether GPT-4 could automate this task.

As an example, consider the following function:



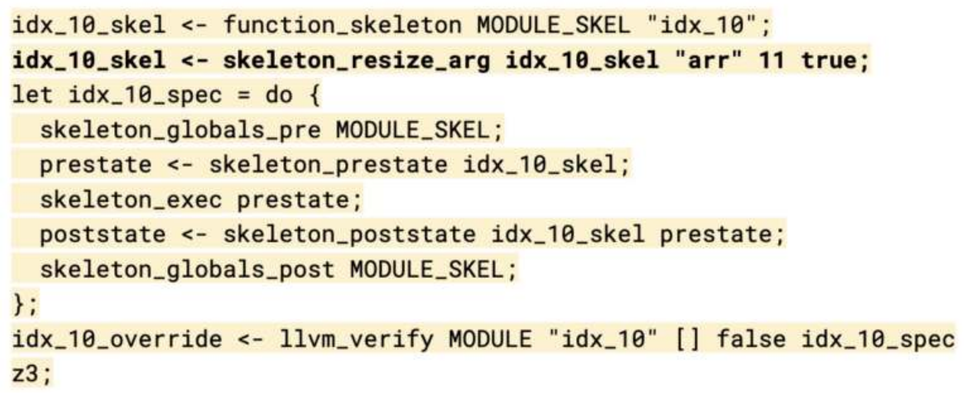
```
void idx_10 (uint32_t *arr) { arr [10] = 10; }
```

(https://galois.com/wp-content/uploads/2023/08/Screenshot-2023-08-04-at-9.36.07-AM.png)

Galois's Software Analysis Workbench (SAW) can explore all possible inputs to this function using an SMT solver and search for any possible memory errors, such as out of bounds access or integer overflow. However, many programs and functions (including this one) make certain assumptions about their inputs, such as the size of an array. To solve this problem, SAW can add preconditions specifying an assumption (such as the size of an array), and SAW will explore all possible inputs allowed by the precondition.

With no preconditions, SAW will immediately realize that in our example function, an array with less than 11 elements will generate an out of bounds access error in the function idx_10. So, we must add a precondition to assume that the array has 11 elements. SAW will then explore all possible inputs with this assumption and find no memory errors. It is also possible using a different bit of syntax to specify the values or range of values of variables as well.

The following SAWScript with a bolded precondition encodes the properties we need. With this, SAW will verify the memory safety of the function.



```
idx_10_skel <- function_skeleton MODULE_SKEL "idx_10";
idx_10_skel <- skeleton_resize_arg idx_10_skel "arr" 11 true;
let idx_10_spec = do {
  skeleton_globals_pre MODULE_SKEL;
  prestate <- skeleton_prestate idx_10_skel;
  skeleton_exec prestate;
  poststate <- skeleton_poststate idx_10_skel prestate;
  skeleton_globals_post MODULE_SKEL;
};
idx_10_override <- llvm_verify MODULE "idx_10" [] false idx_10_spec
z3;
```

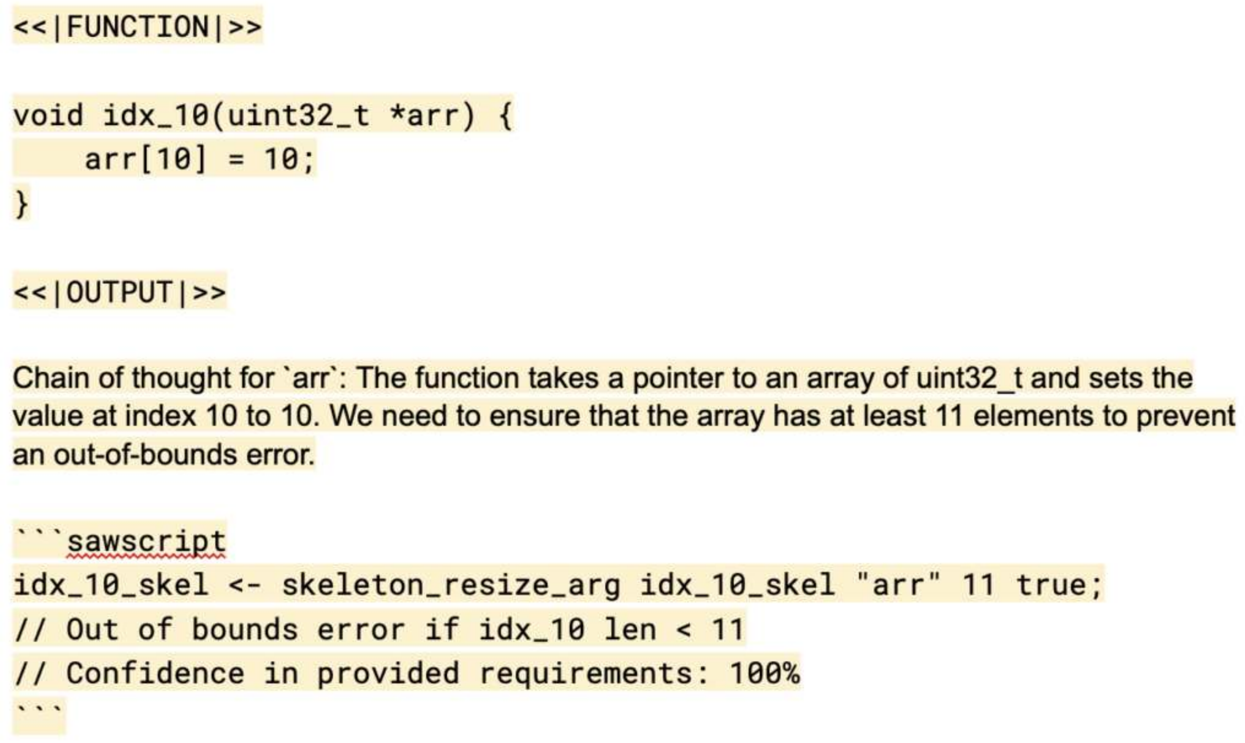
(https://galois.com/wp-content/uploads/2023/08/Screenshot-2023-08-04-at-9.36.43-AM.png)

This SAWScript uses *skeleton specifications*, an experimental SAW feature, which provides a simpler interface to memory preconditions. For this experiment, I wanted a very simple interface to memory preconditions. This is because GPT-4 is not very familiar with SAWScript. There is much less SAWScript on the internet than more common languages like C, Python, or Haskell, and we must prompt GPT-4 on how to use SAW.

For the limited case of memory safety using skeleton specifications, the input to SAW is fairly simple. We need preconditions to set the range of values for variables, the sizes of arrays, whether an array's size is fixed or arbitrary, and if the contents of a pointer must be initialized when passed to the function.

Initially, I tried to write a list of SAWScript rules and instructions for GPT-4, but this didn't work well in practice. It is difficult to write a list of rules that accurately captures intent with no ambiguity. I found that it was much easier to use an approach called "few-shot prompting" and create examples of functions and their associated outputs.

Here is an example prompt I created to teach GPT-4 the correct response for the above function idx_10:



```
<<|FUNCTION|>>

void idx_10(uint32_t *arr) {
    arr[10] = 10;
}

<<|OUTPUT|>>
```

Chain of thought for `arr`: The function takes a pointer to an array of uint32_t and sets the value at index 10 to 10. We need to ensure that the array has at least 11 elements to prevent an out-of-bounds error.

~~~
```sawscript
idx_10_skel <- skeleton_resize_arg idx_10_skel "arr" 11 true;
// Out of bounds error if idx_10 len < 11
// Confidence in provided requirements: 100%
```
~~~

(https://galois.com/wp-content/uploads/2023/08/Screenshot-2023-08-04-at-12.37.14-PM.png)

In this case, GPT-4 only has to output the precondition. I created several of these toy functions, demonstrating to GPT-4 how to use SAWScript in various scenarios: arrays with a fixed size, arrays with an arbitrary size, and variables with required values. In the GPT-4 prompt we used in our tests, there was an introduction to the task followed by five separate examples. After these examples, I would then append a function from the target example (for example, the algorithm salsa20).

A Python script orchestrated all GPT-4 calls and used a regex to find each precondition and insert it into the appropriate spot in the SAWScript program. For certain preconditions that GPT-4 returned, the script would send another prompt to refine the precondition. For example, if the precondition contained `skeleton_resize_arg`, a prompt would be sent to determine if the argument must be initialized when passed to the function. Finally, if SAW did return an error, the python script would prompt GPT-4 with the original prompt, GPT-4's response, and the error message. Surprisingly, just returning a SAW error message was enough for GPT-4 to fix some failing SAW proofs. This was surprising to me because of how rare SAW error messages are on the Internet and thus in GPT-4's training dataset.

Results

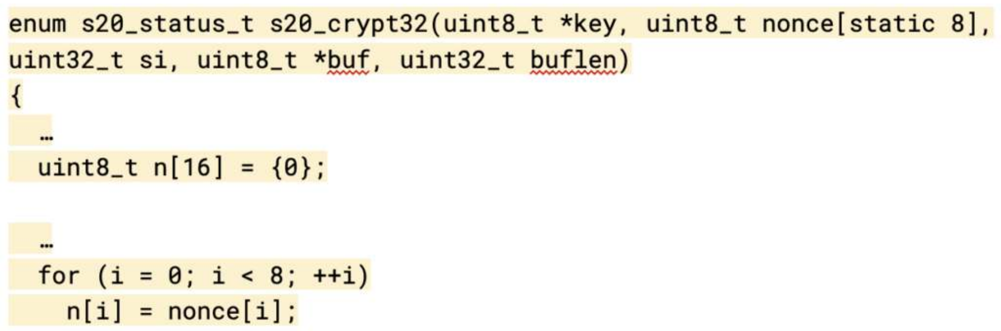
The problem of identifying memory skeletons is not inherently very difficult, but it requires some domain knowledge and is a common sticking point for novice SAW users. In addition, the specific memory skeleton specification I used is an experimental feature of SAW that is not commonly used and I found no examples of this particular skeleton usage on Google, which suggests that it is also rare or not present in GPT-4's training dataset. For this reason, I consider this a moderate challenge for GPT-4.

However, even my simple approach was enough to generate a memory safety proof for salsa20, which contains about 150 lines of code. I consider this a success: with a very small level of effort I was able to automate a proof task that would otherwise require manual human work.

One interesting aspect of this approach is how general the solution is. The prompt is only to create preconditions to prevent memory errors, and GPT-4 is able to use its extensive C knowledge to correctly create preconditions for a variety of variables being manipulated by bitwise operators and pointer operations, even though these bitwise operators and pointer operations are not mentioned in the prompt input. In addition, GPT-4 is able to correctly handle more subjective tasks, such as determining if the size of an array should be fixed or arbitrary.

A concrete excerpt of a function from salsa20 and GPT-4's output:

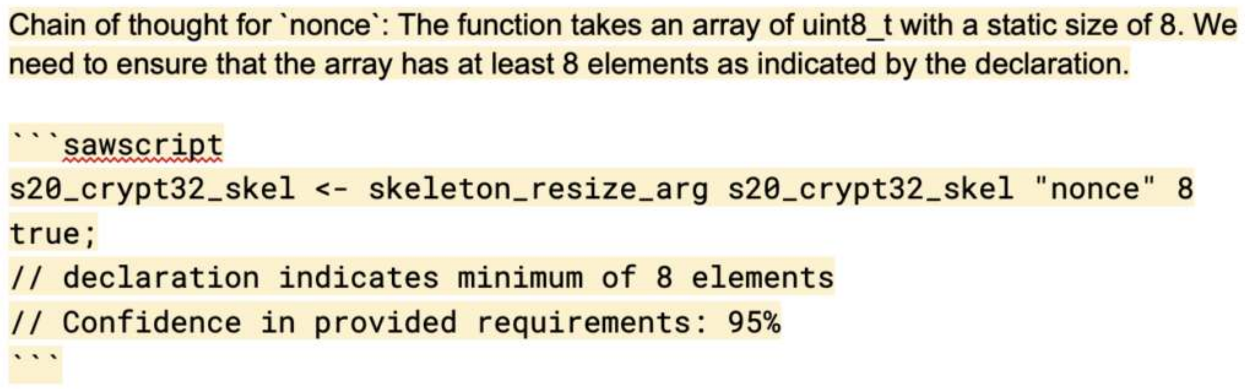
Salsa20 function excerpt:



```
enum s20_status_t s20_crypt32(uint8_t *key, uint8_t nonce[static 8],
uint32_t si, uint8_t *buf, uint32_t buflen)
{
  …
  uint8_t n[16] = {0};

  …
  for (i = 0; i < 8; ++i)
    n[i] = nonce[i];
```

(https://galois.com/wp-content/uploads/2023/08/Screenshot-2023-08-04-at-9.39.27-AM.png)
GPT-4 Output:



Chain of thought for `nonce`: The function takes an array of uint8_t with a static size of 8. We need to ensure that the array has at least 8 elements as indicated by the declaration.

~~~
```sawscript
s20_crypt32_skel <- skeleton_resize_arg s20_crypt32_skel "nonce" 8
true;
// declaration indicates minimum of 8 elements
// Confidence in provided requirements: 95%
```
~~~

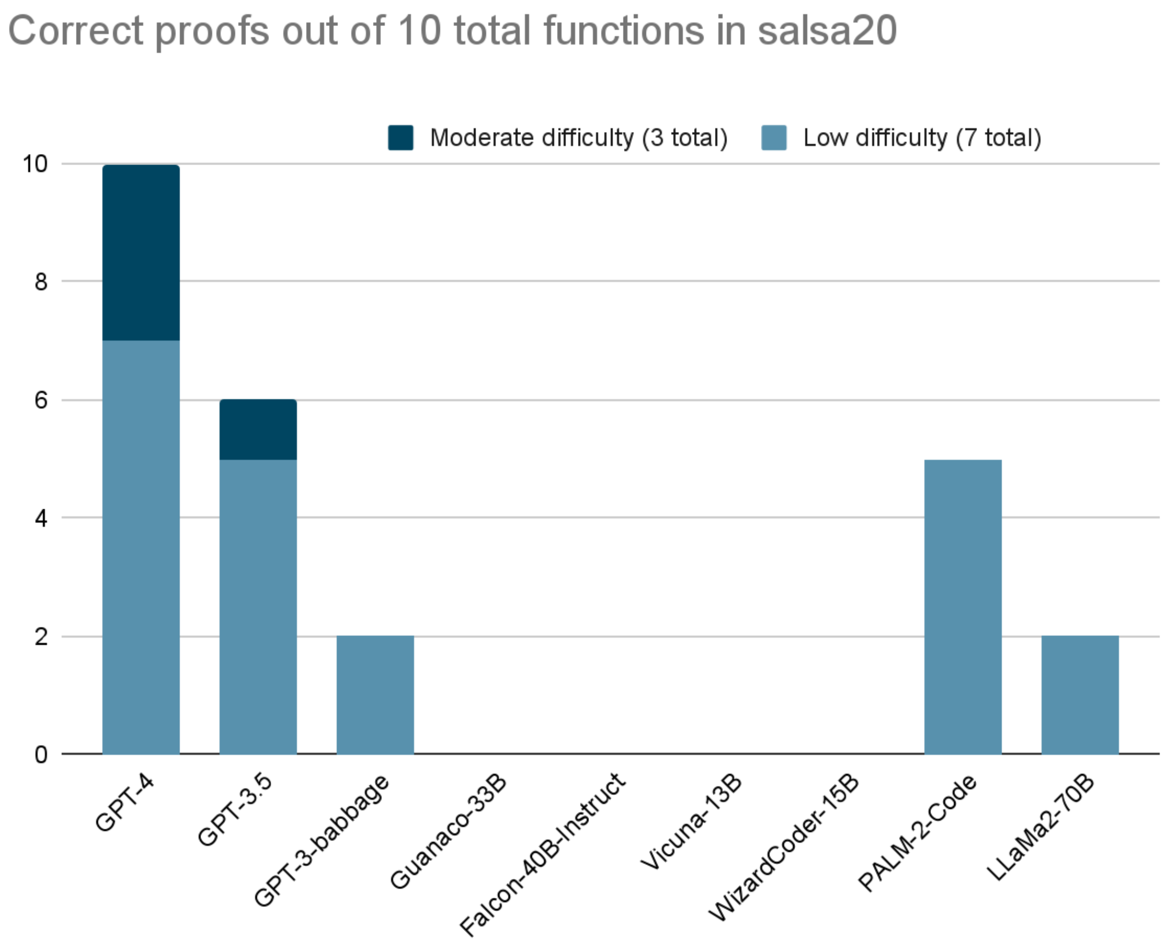
(https://galois.com/wp-content/uploads/2023/08/Screenshot-2023-08-04-at-12.37.46-PM.png)
The full GPT-4 transcript produced correct memory safety proofs for all arguments to the `s20_crypt32` function, and may be viewed here (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/crypt32_transcript.txt).

This is a cool little demo. However, real codebases are much more complex, and this approach would need further development to be used on more complex files. For example, we could help GPT-4 by annotating the code with non-ML tools, and further break the task down into simpler subtasks. A significant benefit of approaches like ours is that GPT-4 doesn't have to get everything 100% correct. As long as some of its outputs are helpful to a human developer, then it can still be useful.

Takeaways

One of the major takeaways is that prototyping with off the shelf large language models is very easy. It took less than 20 hours to go from the initial idea to a functioning prototype, and most of these 20 hours were spent plumbing a Python script into Docker, SAW, a C parsing library, and the OpenAI API. Experimenting with prompts requires much less effort and enables faster iteration than many traditional approaches. However, we are also investigating fine tuning models on custom datasets, as it may enable use cases out of the reach of prompt engineering.

A second insight from this project is when exploring a certain task, first test the most capable model available. If I had used any model other than GPT-4, I would have concluded that large language models were not suited to this task. If this approach was going to work at all, the model should be able to easily write a proof for a simple program like salsa20. To illustrate this, I tested a variety of models on this task, including Google's PALM-2-Code and the five leading open source models on the Hugging Face leaderboard. Only GPT-4 was able to correctly prove all functions memory safe. As I had initially created my prompt for GPT-4, I spent some time experimenting with various prompts with every model to ensure that the problem was with the models and not the prompt.



I have found through my experimentation that prompts for large language models are very important. When you are evaluating a large language model, what you are actually evaluating is the combination of the model and the prompt. It is very easy to be misled into thinking that a model can't do something by using a poor quality prompt. When testing the capability of large language models for a particular task, it is a good practice to spend several hours experimenting with different prompts using the most capable available model.

Interestingly, more capable models like GPT-4 seem less sensitive to the specific language of a prompt, and more sensitive to the content of that prompt. With earlier, less capable models like GPT-3 there was a sense that prompting is very mysterious, and you can change one word in a prompt and it suddenly works for no apparent reason. This does not seem to be the case for more capable models like GPT-4. Instead, a good prompt relies on writing complete and clear specifications and breaking complex tasks down into simpler subtasks where possible (the same might be said for working on complex tasks with humans).

Conclusion

It appears possible to use capable large language models on tasks not present in their dataset purely through prompt engineering, without any finetuning of the underlying model. Prompt engineering takes very little effort and time compared to traditional approaches that rely on collecting large amounts of high quality data. There is friction added when instructing the model how to use a language or tool in the prompt, and this approach would be much more challenging on an unfamiliar task that contains a lot of nuance and complexity. Fortunately, the task of writing SAWScript memory safety proofs using skeleton specifications primarily requires C knowledge which is present in GPT-4's training dataset.

There are many unanswered questions on how to best leverage potentially unreliable large language model outputs. Current large language models should not be blindly trusted in high stakes applications, but there are many potential applications with lower stakes where it may be possible to use them productively. Alternatively, some higher stakes applications might find use for LLMs when LLM outputs can be robustly verified as either correct or incorrect. Formal methods present one possible avenue, where it is possible to use an oracle such as SAW to evaluate model outputs. We are currently exploring and searching for more of these potential applications.

Appendix

For more specific details on our process, please see our prompt input (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/input.txt), array initialization prompt input (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/array_init_input.txt), and complete SAW program output (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/salsa20_compositional_proof.saw) examples on GitHub.

PREVIOUS (HTTPS://GALOIS.COM/BLOG/2023/08/SPINOUT-STORIES-MUSEDEV/)

NEXT

421 SW 6th Avenue, Suite 300
Portland, Oregon 97204
(https://www.google.com/maps/place/Galois,+Inc./@45.520811,-122.678081,17z/data=!4m6!1m3!3m2!1s0x54950a04159ece0f:0x36857895c75e27d7!2sGalois,+Inc.!3m1!1s0x54950a041

Arlington, VA

901 N Stuart Street, Suite 501
Arlington, Virginia 22203 (https://goo.gl/maps/pxFK95q48t32)

Minneapolis, MN

111 Third Avenue South, Suite 350
Minneapolis, MN 55401 (https://maps.app.goo.gl/csQrxYhmMPHDXLZJA)

Dayton, OH

444 E 2nd Street
Dayton, Ohio 45402 (https://goo.gl/maps/cRzPpKEF6eD2)

T 503.626.6616 (tel:15036266616)
F 503.350.0833

contact@galois.com (mailto:contact@galois.com)