Automated Creation of Source Code Variants of a Cryptographic Hash Function Implementation Using Generative Pre-Trained Transformer Models

Elijah Pelofske*[1], Vincent Urias[1], and Lorie M. Liebrock[2,1]

[1]Sandia National Laboratories
[2]New Mexico Cybersecurity Center of Excellence, New Mexico Tech

Abstract

Generative pre-trained transformers (GPT's) are a type of large language machine learning model that are unusually adept at producing novel, and coherent, natural language. Notably, these technologies have also been extended to computer programming languages with great success. However, GPT model outputs in general are stochastic and not always correct. For programming languages, the exact specification of the computer code, syntactically and algorithmically, is strictly required in order to ensure the security of computing systems and applications. Therefore, using GPT models to generate computer code poses an important security risk – while at the same time allowing for potential innovation in how computer code is generated. In this study the ability of GPT models to generate novel and correct versions, and notably very insecure versions, of implementations of the cryptographic hash function SHA-1 is examined. The GPT models `Llama-2-70b-chat-hf`, `Mistral-7B-Instruct-v0.1`, and `zephyr-7b-alpha` are used. The GPT models are prompted to re-write each function using a modified version of the localGPT framework and langchain to provide word embedding context of the full source code and header files to the model, resulting in over $130,000$ function re-write GPT output text blocks (that are potentially correct source code), approximately $40,000$ of which were able to be parsed as C code and subsequently compiled. The generated code is analyzed for being compilable, correctness of the algorithm, memory leaks, compiler optimization stability, and character distance to the reference implementation. Remarkably, several generated function variants have a high implementation security risk of being correct for some test vectors, but incorrect for other test vectors. Additionally, many function implementations were not correct to the reference algorithm of SHA-1, but produced hashes that have some of the basic characteristics of hash functions. Many of the function re-writes contained serious flaws such as memory leaks, integer overflows, out of bounds accesses, use of uninitialised values, and compiler optimization instability. Compiler optimization settings and SHA-256 hash checksums of the compiled binaries are used to cluster implementations that are equivalent but may not have identical syntax - using this clustering over $100,000$ distinct, novel, and correct versions of the SHA-1 codebase were generated where each component C function of the reference implementation is different from the original code.

1 Introduction

Generative Pre-Trained Transformer (GPT) models are a type of Large Language Model that has shown to be highly capable at a large number of natural language processing tasks, including computer code [1–8].

In this study we explore whether current GPT models can be used to generate correct algorithmic invariant implementations of a cryptographic hash function in C code. Specifically, we examine the task of rewriting an implementation of the cryptographic hash function known as SHA-1 [9]. An interesting byproduct of code-rewriting with the GPT models is that a large number of the implementations are wrong in a variety of surprising ways, including containing software risks. These include compiler optimization instability (meaning that the output changes based on what compiler optimization settings were used), memory leaks, integer overflows, out of bounds writes, and implementations that are correct for some test vectors but not correct for other test vectors. Implementation risks in cryptographic algorithms is a critically important type of bug that exists in cryptography library implementations [10–13]. This study serves to caution that arbitrarily using GPT models for creating, or rewriting, source code can introduce serious flaws. GPT models, with their current capabilities, work well as research tools to study interesting source code variants, but using them for practical code generation poses a software security risk.

[*] E-mail: elijah.pelofske@protonmail.com

Importantly, SHA-1 [9] is considered *broken* since at least one method of generating a SHA-1 hash collision has been demonstrated [14], and is generally considered not secure for validating the integrity of data based on a variety of attacks [15–19]. However, SHA-1 is still widely deployed in information technology systems, and moreover its implementation is relatively short compared to other cryptographic algorithms, making it a good candidate for a proof of concept study on the capabilities of using GPT models for rewriting a cryptographic algorithm implementation.

In the context of malware detection and analysis, polymorphic versions of malware binaries can be very easily constructed [20]. However, changing the underlying syntax while maintaining the same functionality is harder to do, especially in an automated manner. Here, we show that GPT models can be used as tools for creating unique versions of a codebase. This is important because this capability of GPT models can, and likely will, be used by malware developers to create software implementations that have distinct behaviors and syntax, thus making their detection more difficult.

Automated code re-writing, in particular of cryptographic algorithm implementations, is an excellent test case for evaluating GPT models capabilities involving computer code because the implementation must be exactly correct or software implementation risks will be introduced. The validity of GPT produced cryptographic function re-writes is highly testable for correctness and implementation flaws such as being incorrect for some proportion of inputs or memory leaks. A cryptographic algorithm serves as a good reference benchmark for the capabilities of generative machine learning to correctly rewrite computer code because they need to implemented exactly correctly or they do not work – and in particular the code is therefore highly testable for correctness. Although there exist many instances of implementations of cryptographic algorithms in the codebases that very likely were used to train many of the GPT models in existence today, they are not as common (and in as many different varieties) as other foundational functions in computing (e.g., especially for teaching programming, such as sorting algorithms), and moreover their secure and correct implementation is extremely consequential.

Importantly, code re-writing and synthesis tasks are severely constrained by the *token context window size* of the given GPT model. In this case, the cryptographic source code functions and the input prompt do fit within the context window, but depending on the given run the GPT model output may overrun that context window and begin to generate incoherent text. Future evaluations of code re-writing will also be limited by the context window of the GPT models, and therefore GPT models with larger context windows will be needed for synthesizing larger pieces of computer code.

When parsing the generated code, we do not impose any further post-processing beyond attempting to extract the code from assumed markdown-style code formatting. In particular, the extracted strings are directly substituted in the source code for the original function implementation. This means that if the format was correct, we allow the GPT model output to include additional function definitions or even additional standard libraries – the test of whether any of such code modifications succeed is determined by the ensemble of compile attempts and algorithmic correctness tests.

The re-written C code functions are evaluated in a number of ways, most importantly by being compilable by both `gcc` and `clang` [21–23] with a variety of optimization levels, and by algorithmic correctness. Memory leaks, memory allocation flaws, and out of bounds writing are checked using the address sanitizer in clang and gcc [24], as well as Valgrind with memcheck [25–29].

1.1 Brief Literature Overview of GPT Model Code Generation

Ref. [30] used chatGPT to implement cryptographic algorithm source code, however it was in a semi-supervised chat interaction, not in an automated or systematic study.

There exist several previous studies on measuring the capability and accuracy for code generated using GPT models [31–37], however testing for accuracy and completeness can be difficult especially for extremely complicated computer code. Previous studies have also used LLMs for helping with fixing compilation errors [38] and generally as an assistant for writing code [39].

Several studies have investigated the ability of GPT models to repair source code that contains flaws [40–45]. The more general task of using GPT models to generate source code has also been studied in several contexts [33, 36, 46–48]; overall such generated code has the same problem as human developer written code which is properly unit-testing for correctness. The task of automatically producing unit tests for source code and software using GPT models has also been studied [49–53].

2 Methods

2.1 GPT Model Implementation

For this study, langchain [54], and in particular the software codebase localGPT [55], is used with the goal of *anchoring* the GPT output within the reference hash function code – the entirety of the code, not just the single function being re-written within each inference call. This technique is generally known as retrieval augmented generation [56], and the goal is to provide sufficient context via word embeddings of a corpus of text we wish to extract information from such that the GPT models will generate text that is grounded in the content of those documents.

localGPT is used to first create word embeddings, tailored for text generation using the `instructor-xl` model [57], of the entirety of the original source code, which is comprised specifically of the SHA-1 reference implementation and the corresponding header file (shown in Appendix B as Code Listing 9 and Code Listing 10). Note that the original macro and definition comments are left in the header file, meaning that the text will be in the generated word embeddings, so as to provide better context for the functionality of the code. These files are parsed as raw text files.

The localGPT inference calls are performed on a small local cluster on four Nvidia A100 GPU's [58] with 82 Gigabytes of memory, with CUDA Version 12.4, and the GPT models are all obtained from the huggingface GPT repository [59]. The GPT models were trained and run using the Python 3 library PyTorch [60].

The localGPT prompt template used is a combination of *Context* and the user-facing *question*. No chat history was used for the prompting of the re-written code. Regardless of the type of the underlying model, the high-level organization of the prompt is the system prompt [55], followed by the context of the generated word embeddings, followed by the user prompt. The system prompt used in this study, which is from a version of the localGPT codebase [55], is the following:

> You are a helpful assistant, you will use the provided context to answer user questions. Read the given context before answering questions and think step by step. If you can not answer a user question based on the provided context, inform the user. Do not use any other information for answering user. Provide a detailed answer to the question.

A total of three pre-trained language models are used in this study; `Llama-2-70b-chat-hf` [3], `Mistral-7B-Instruct-v0.1` [61], and `zephyr-7b-alpha` [62]. `Llama-2-70b-chat-hf` has a maximum token context window of 4,096, `Mistral-7B-Instruct-v0.1` and `zephyr-7b-alpha` both have a maximum token context window of 32,768. These GPT models are intended to be prompted in a chat-type manner of interaction. These three GPT models were chosen as a representative group for their relatively large context window, and their overall good performance for handling computer code. However, there are a very large number of GPT models in general and there are likely many other models that may perform very well, or even better, than these three.

For each inference call to the GPT model, the source code of the function is appended (along with a newline character) after the prompt text. Each prompt needs to ensure a few things. The first is that the code needs to be enclosed in triple backticks (also known as triple backquotes) so that the code can be automatically parsed from the output. The second is that the code needs to be compatible with the rest of the SHA-1 codebase, including usage of macros, functions from library imports when required, and using consistent function naming schemes so that the algorithm can be automatically tested. In cases where these requirements are not adhered to, the result is either a failure to compile or the compiled binary having a critical error when executed. The prompts are also intended to be code-agnostic; for example the prompts are not requesting a specific algorithm or type of syntax be used. Lastly, the primary intention is for the generated code to be correct, but to have different syntax than the original. A total of 10 prompts are tested which aim to produce generative text output that has these desired properties. The exact text of these 10 prompts are given below:

> **Prompt 1**
>
> Re-write this C code function into an entirely different function that maintains the same functionality as the original code and uses the same function name. Enclose the code in triple backquotes.

Prompt 2

Re-write this C code function into an entirely different function that maintains the same functionality as the original code and uses the same function name. Use different syntax choices when re-writing the source code including but not limited to different control flow, equivalent but different array indexing, different logic operations, different variable types, different algorithm choices, and different variable names. Be creative! Try to obfuscate the intended functionality of the code as much as possible while retaining the same functionality. Enclose the code in triple backquotes.

Prompt 3

Re-write this C code function using different variable names, control flow, and array indexing, so that the functionality of the code is obfuscated, but the functionality is the same as the original code. Use the same function name as the original code and enclose the rewritten code in triple backquotes.

Prompt 4

Obfuscate this C code function by rewriting the syntax and making the code more complicated than it needs to be while performing the same functionality as the original code. Use the same function name as the original code and enclose the rewritten code in triple backquotes.

Prompt 5

Obfuscate this C code. Enclose the code in triple backquotes.

Prompt 6

Re-implement this C function using a different implementation with changed logic and variable names. Use the same function names as the original, and enclose the code in triple backquotes.

Prompt 7

Rewrite this C code using different variable names and different control flow logic, but keep the function name the same. Enclose the code in triple backquotes.

Prompt 8

Re-implement this C function using different logic and variable names. Use the same function names as the original code, and enclose the new code in triple backquotes.

Prompt 9

Act as a professional C code developer. Re-implement this C function using different logic and variable names. Use the same function names as the original code, and enclose the new code in triple backquotes.

Prompt 10

Please alter this C function so that it uses completely different, but still valid, C syntax such that it performs the same computations as the original code. Surround the new C function in triple backticks and use the same function names as the original code. Do not write explanations or justifications in your reply; write only the new C function and nothing else.

For each of the 10 prompts, a total of 100 text-generation calls are performed, and the inference temperature is varied across 11 temperature settings between the maximum of 1 and 0.01;

$\{1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.01\}$. The inference temperature of 1 gives the generative text calls higher stochasticity, meaning that each of the 100 inference calls will very likely output very different source code. The inference temperature of 0.01 gives nearly deterministic code generation. Temperature here refers to the type of distribution that is sampled from whenever each next token is chosen. Here, sampling is always on meaning that the temperature can not be exactly 0, but 0.01 means that the next token that is chosen is almost always the most likely choice (based on the pre-training of the model), and higher temperatures mean that the sampling has some higher likelihood of being a not very likely token. An inference temperature of 100 would give closer to uniform sampling across the tokens when making the choice of what token to produce next. Because of the nature of computer code requiring very exact syntax, we opted for temperature settings starting at 1 and lower so as to (hopefully) generate coherent code. The wide range of inference temperatures gives provides output code that is highly diverse. This procedure is then repeated for each of the 3 GPT models, and is also repeated for the four component functions of the chosen SHA-1 algorithm implementation [63]. Therefore, in total there are $10 \cdot 100 \cdot 11 \cdot 4 \cdot 3 = 132,000$ function re-write attempts that are generated and then parsed. Although, the experiments shown in this study do not fully cover this parameter space, only about 90% of the parameters are covered – however due to the *extra* function re-writes that are parsed, over $132,000$ function re-writes are produced in total. This large number of function re-writes is motivated by the inherent randomness of the GPT model output, and therefore it is important to quantify a large *distribution* of code samples so as to understand the variability in GPT produced code.

2.2 GPT Output Parsing and Code Testing

The SHA-1 implementation re-writing is performed entirely on the four component functions of the SHA-1 C code given by this Github repository [63]. This particular SHA-1 implementation was chosen because it is relatively self contained in terms of required external libraries, and is relatively short. However, the same methodology described in this study could be easily applied (with a reasonable amount of GPU compute time) to other SHA-1 implementations, or any other source code implementation.

With the goal of fully evaluating each of the GPT re-written functions, we execute the following series of parsing attempts and tests for each re-written function, in sequence.

1. The first step is to parse the GPT output to extract the source code. GPT output, at least for the models we tested, can be quite unstructured and not adhere to prompted formats. Here we apply a reasonably black-box approach, in that we do not apply extensive natural language processing to separate out natural language descriptions from source code. The requested format for all of the prompts is to output the re-written C code enclosed in triple backquotes (also known as triple back-ticks). The triple backquotes formatting is chosen because it is quite distinguishable and parseable for the automated execution of the source code. GPT models typically output natural language text in addition to generated code, and therefore we need to use a mechanism to (hopefully) differentiate the source code. It is also markdown formatting syntax, commonly used for code formatting in software development documentation, so it should be a reasonably consistent formatting choice with the source code the GPT models were trained on. Therefore, the following sequence of initial parsing is attempted:
   (a) Apply a regular expression to split the output string into an array at all instances of triple backquotes, and return the string at index 1 (indexing beginning at 0). Note that if the beginning of the string is the first triple backquotes, the string at index 1 is the intended source code function text. If there are not triple backquotes, proceed to step 2. If the output text contained two triple backquote lines enclosing some source code (with anything else before or after that block), then this parsing will succeed correctly, as in the generated code will be extracted. This parsing method specifically extracts the strings in between the delimiter of a pair of triple backticks – this allows the GPT output to have any amount of further text or backticks in the output as this will not be parsed.
   (b) Apply a regular expression to split the output string into an array at all instances of a *single* backquote, and return the string at index 1, as before. If there are not backquotes in the output string, proceed to step 3. In some instances, the GPT model output would enclose the function code in single backquotes instead of the requested triple backquotes. This is not entirely unreasonable since this is also a used in-line markdown formatting, usually for code syntax. Therefore, if step 1 fails we do apply this step in order to maximize the parse-able test cases. This case does not occur very frequently, but it does happen in various GPT model outputs.
   (c) Return the raw source code with no parsing modifications. If we reach this step, most likely the generated text output does include some non C-syntax characters. But, in some cases the GPT output is actually

just re-written C code with no ancillary text, and in those cases this parsing works.

2. If in the previous parsing step, the first or second step applied, then we will apply a further post-processing step here. This step was motivated by some of the GPT model output adhering to markdown formatting in a somewhat strange way, which is by including the computer code language identifier immediately following the markdown triple, or single, back-quotes, such as `` ```Python ``. Therefore, this parsing step is to split remaining text by newlines into an array and check if the first index is the same as any of the standard computer language identifiers. If it is, then we remove that text, and proceed. We also check minor edge cases such as computer language code identifiers followed by a single whitespace, and remove those cases as well. The list of computer code language identifiers that are checked for are given in Appendix C. This parsing step is safe, in the sense that it does not remove any potentially valid C code syntax, since all of the language identifier extensions are not valid C code for the beginning of a function.
3. Since we are considering each function individually, we want to evaluate that function by itself, with no other changes to the source code. Therefore, we directly substitute this re-written function (which, at this stage is simply the post-processed string from the GPT model) for the correct function from the reference implementation, concatenate the other reference implementation C code with the re-written C code, and write that combined string to a new C file. Since the function re-writing does not include the macros, and library imports, those strings are also written to the new C file.
4. The new C file is compiled, along with the reference header file (none of which is ever re-written by the GPT models), along with the reference test C code file (which contains the `main` function as the entrypoint for the program) that will call the functions on test vectors and check if the output is correct. This codebase is compiled using both `clang` and `gcc`. For `gcc`, the optimization levels of 0/1/2/3/fast/s are used (and all default settings otherwise). For `clang`, the optimization levels of 0/1/2/3/fast/s/z are used (and all default settings otherwise). s denotes space optimization compilation, z denotes heavier space reduced optimization. This codebase is compiled on the same computer, and using all identical settings to ensure maximum similarity for the eventual checking of identical generated binaries. The large number of compiler options at this step can produce a variety of outputs and behaviors, including a binary being generated with no errors, a binary being generated but with warnings, a binary not being generated at all, and various compiler optimization instabilities. Importantly, note that the original source code is compiler optimization stable.
5. If the compilation step (this step is tested for each compiler setting) produces a compiled binary, then the binary is executed. This binary is first a check of the correctness of the code, but it also outputs what the computed cryptographic function output is on the input of the test vectors (for example, even if it is incorrect) which can be analyzed at a later point as well. Note that the GPT re-write can introduce ancillary text as well besides simply computing the cryptographic function (which is almost certainly measured as a fail case because the code can not be compiled). At this point of executing the binary, there is no guarantee that it is correct – for example, it can (and we found examples that have) return an error code, it can enter into an apparent infinite loop (which is checked by a timeout), or it can have memory leaks.
6. The last step is is to perform a series of automated memory leak, memory allocation error, and out of bound write checks. These steps are attempted on every function re-write; in the cases where the code does not compile, or the binary throws an error when executed (or times out due to an apparent infinite loop), the test aborts and no memory leak or out of bounds information is obtained. This involves four separate compilation and binary execution steps. All of these four compilation attempts use an optimization level of 0 so as to minimize potential errors, such as false positives, for the automated tools evaluation. The binary output from these tools executing and or compiling the code is recorded, but is not analyzed for correctness or interesting outputs; the output of the tools however is parsed in an automated system to find specific phrases that indicate the lack or presence of certain errors.

   - Compile the code using `gcc` with flags `-g` and `-fno-omit-frame-pointer`, with the full address, undefined, and bounds sanitizer check flags enabled [24]: `-fsanitize=address`, `-fsanitize=bounds`, `-fsanitize=undefined` (and statically linked with `-static-libasan`). Then execute the compiled binary, if it was compiled, and record the output. The automated sanitation checks will print metadata about flaws in the code such as memory leaks or out of bounds accesses.
   - Compile the code using `clang`, and supply the same address, bounds, and undefined sanitize checks [24] as in the above case.
   - Compile the code using `gcc` and all default flags, then if the binary was produced run it through valgrind and record the output (catch exceptions, and set a timeout check usual to handle the binaries with fatal errors or apparent infinite loops). The valgrind flags that are specified are: `--tool=memcheck, -s, --leak-check=full, --leak-check=yes, --show-leak-kinds=all, --error-limit=no`

- Perform the same steps as the previous test, but using `clang`

7. Finally, this last step is not actually parsing the primary function, as described in all of the previous steps. Instead, this step allows us to obtain *extra* function re-writes that were generated by the GPT output, but were not parsed in any of the previous steps. This step was implemented because it was found that in many of the GPT outputs, there were additional markdown formatted fields of code in the output strings. This step splits the output strings based on all occurrences of triple back-ticks, then reading in all resulting split substrings (skipping the first one at index 1, since this was the function parsed in Step 1). Then, for each of these substrings, they are considered potential function re-writes if they contain at least one of each of the following characters; {, }, (curly braces) (,), (parentheses) _ (underscore). These checks are used because any valid C functions needs the curly braces and the parentheses, and then all of the function names used in this codebase contain an underscore. If these checks pass, then the string is parsed and is considered as a potential function rewrite - this means specifically that it then is processed by steps 2-6. The number of extra functions that this step can produce is overall somewhat small (this usually adds about 10 percent additional functions), but some of the GPT outputs can have upwards of 100 function re-writes that get added at this step.

Note that all of these steps (not including the actual GPT inference calls) are intended to be deterministic – these steps are intended to be fully reproducible in order to analyze the GPT source code outputs.

Using the metadata generated from these tests and parsing steps, we compute the following aggregated metrics on the generated source code function variants. Note that these metrics are computed specifically on the individual re-written functions, while leaving all other aspects of the codebase unchanged. The GPT generated code can, and often does, include function names which are not compatible with the assumed function names of the component functions, adds extra functions, among a variety of other interesting outputs. In such cases, no extra parsing is performed – the above tests are executed in a fully automated manner, and the output is then quantified by the following metrics so as to obtain a high-level summary of the types of code produced.

Metric 1. Count of how many of the function variants were able to be compiled, in the sense that a binary was produced from the compilation even if there were warnings, for all compiler settings.
Metric 2. Count of how many of the function variants were *compiler optimization unstable*, meaning that the compilation was successful for at least one, but not all, of the compiler settings.
Metric 3. Count of how many of the function variants were output-verified (e.g., the implementation was correct for the test vectors) for all compiler optimization settings.
Metric 4. Count of how many of the function variants were algorithmically incorrect in some way for all compiler settings. In particular, this means that for all compiler variants output was produced (meaning that the binary did not crash, for example, for any of the compiler settings), but the output was incorrect. The way that the output is incorrect can vary - from being off by a single character, to adding large amounts of output that is ancillary. This count is strictly for the cases where the compiled binaries were able to be produced for all compiler optimization settings.
Metric 5. Count of how many of the function variants, for which binaries could be compiled, were compiler optimization unstable for their algorithmic correctness – meaning that for some compiler optimization settings the code passed all cryptographic algorithm test vectors, but for others it failed. This test allows other optimization settings to cause the binary to not be compiled or executed with a critical or timeout error; the relevant count here measures purely if there were two compiler optimization settings where one resulted in a successful algorithmic test check, and the other resulted in a failed algorithmic test check.
Metric 6. Count of how many of the function variants that were correct for all compiler settings and had a Levenshtein character distance of 0 to the original source code (meaning, that the source code is strictly identical to this function variant), after repeated whitespace was removed and all comments were removed.
Metric 7. Count of how many of the function variants that were correct for compiler settings had a Levenshtein distance greater than 0 with respect to the original source code after repeated whitespace was removed and all comments were removed. This metric shows how many correct function re-writes there were, in the sense that the underlying code was changed in some way, and the code was correct (and not compiler optimization unstable).
Metric 8. Count of how many of function variants produced a compiled binary that crashed due to a timeout error (e.g., a presumed infinite loop) for all compiler settings. The timeout threshold was set at 10 seconds, and for reference, the original implementation completed all tests in less than 1 second of CPU time.
Metric 9. Count of how many of the function variants produced a compiler binary that crashed due to critical error, for all compiler settings.

Metric 10. Count of how many of function variants produced a compiled binary that crashed due to a timeout error (e.g., a presumed infinite loop) for at least one, but not all, compiler setting. Meaning that the occurrence of this error is unstable and dependent on the compiler optimizations.

Metric 11. Count of how many of the function variants produced a compiler binary that crashed due to a critical error (for example, a segmentation fault), for at least one, but not all compiler setting. Meaning that the occurrence of this error is unstable and dependent on the compiler optimizations.

Metric 12. For each compiler setting, a count of how many unique (identical) SHA-256 hash clusters exist for the binaries that were output-verified for all compiler settings and whose source code was distance 1 or greater away from the original (with comments and repeated whitespace removed). One integer is reported for each of the 13 compiler settings (separated by dashes). The hash clusters were computed by taking the hash of each compiled binary; if that hash was the same as other compiled binaries, then they were assigned the same cluster. This hash based clustering abstracted away the problem of similar source code (away from directly computing text distance) to compiled binary similarity. This works especially well thanks to compiler optimization, which can elucidate cases where two pieces of source code are really doing the same computations but just with slightly different syntax (and not entirely different datastructures or control flow). The order of the reported clusters is; gcc level 0, gcc level 1, gcc level 2, gcc level 3, gcc level s, gcc level fast, clang level 0, clang level 1, clang level 2, clang level 3, clang level s, clang level fast, clang level z.

Metric 13. This step aggregates the compiler setting hash clustering into a unified graph (e.g., network) that reveals even more underlying clustering of the source code variants. The graph is defined by each binary being represented by a node, and edges are formed between nodes if the SHA-256 checksum of the two binaries is equal. Next, we check within all of these existing clusters if there exist any with hashes that are equal to the implementations with text distance of 0 to the original source code. This cluster is removed, and not included in the returned counts. In practice, we found there was always exactly 1 such cluster, and it was the largest cluster. The final metric that is reported is an aggregated number of how many disconnected components of the meta-graph exist (this combines binaries that were compiled with the same source code, in addition to the SHA-256 checksum formed graph), which corresponds to showing how many *actually unique* variants of this C code function were generated by the GPT models. Here we also report how many source code versions exist in each of the clusters of the meta-grouping. Note that function variants within each cluster may share identical syntax to each other – this clustering is specifically intended to delineate unique algorithmic invariant implementations of the original code. The integer counts of clustered group sizes are given in an unsorted sequence, separated by dashes.

Metric 14. The number of source code variants that were found to be duplicates of the original source code, but only found via hashing of the compiled binary in Metric 13.

Metric 15. Count of how many of the function variant and compiler setting tuples that did not adhere to the basic format of the algorithm output, and produced some ancillary output strings or non-unicode characters (an example of this could be appending print messages on the internal state of the algorithm). All of these variants are decidedly incorrect, but they are incorrect for potentially additional reasons besides implementing the algorithm incorrectly.

Metric 16. Count of how many of the function variants that under some compiler settings will not compile, whereas for any other compiler setting the code can be compiled and is output-verified. Note that in practice, we never observed an example of this case ever occurring.

Metric 17. Count of how many of the function variants produce binaries that are correct for at least one, but not all, of the test vectors, and this behavior (of some outputs being correct, but others are incorrect) is the same for all compiler optimization settings. In particular, all compiler optimization settings produced a compiled binary, all of the binaries executed without an error, and the compiled binaries failed to pass all of the SHA-1 tests. **These instances are examples of implementation risks, since if these are under-tested they could pass for being correct and then subsequently fail to authenticate the integrity of data.**

Metric 18. Count of how many of the function variants produce binaries that are correct for at least one, but not all, of the test vectors, and this behavior (of some outputs being correct, but others are incorrect) is compiler optimization unstable (meaning that this occurs for at least one, but not all, of the compiler optimization settings). The cases which are not correct could be because that optimization setting caused the binary to not be compiled or to result in an error status. **These instances are examples of implementation risks, since if these are under-tested they could pass for being correct and then subsequently fail to authenticate the integrity of data.**

Metric 19. Count of how many of the function variants produce binaries that are incorrect for all test vectors, but for at least one test vector the output hash is 5 characters or less away from the correct hash, and the output hashes are deterministic regardless of the compiler optimization used. This character distance measure is strictly from the generated hash - if the generated hash contains fewer characters than the correct hash, the missing characters are not counted towards the character distance. In particular, all compiler optimization settings produced a compiled binary, all of the binaries executed without an error, and the compiled binaries failed to pass the SHA-1 tests. The choice of distance of 5 characters is arbitrary - it was selected to identify clear cases where there was very minimal change to the output hash compared to the correct SHA-1 implementation. Notably, instances found by this metric are interesting because they could fail human visual authenticity checks.

Metric 20. Count of how many of the function variants produce binaries that are incorrect for all test vectors, but for at least one test vector the output hash is 5 characters or less away from the correct hash, and the output hashes are inconsistent across different compiler optimization used. The cases which do are not correct could be because that optimization setting caused the binary to not be compiled, or to result in an error status. Like Metric 19, instances found by this metric are notable because they could fail to be found to be incorrect from human visual authenticity checks.

Metric 21. Count of function variants that were incorrect, but the output (e.g., the raw output of the test functions) changed, in at least one way, depending on the compiler optimization level. This count is specifically for the function variants where an executable was able to be compiled and executed without critical or timeout errors for all compiler optimization settings.

Metric 22. Count of how many of the function variants that are incorrect for all test vectors are compiler optimization stable, meaning that the output is the same for all of the tested compiler settings. This count is specifically for the function variants where an executable was able to be compiled and executed without critical or timeout errors for all compiler optimization settings.

Metric 23. Count of how many functions variants were optimization unstable in the sense that for some settings there was a critical error, but for others the resulting binary was correctly output-verified.

Metric 24. Count of how many functions variants were optimization unstable in the sense that for some settings there was a timeout error (likely infinite loop), but for others the resulting binary was output-verified.

Metric 25. Count of function variants that resulted in any detected memory leak using Valgrind (detected using either gcc or clang compiled binaries or both). Note that necessarily these counts are only for the cases where the binaries could be compiled.

Metric 26. Count of the function variants, out of the function variants that were true for Metric 25, that for any compiler optimization level (without the memory checks or memory address sanitizer) was output-verified. Note that in practice, we never observed an example of this case ever occurring.

Metric 27. Count of function variants that had any Valgrind detected `Invalid free() / delete / delete[] / realloc()` error (detected using either gcc or clang compiled binaries or both). Note that necessarily these counts are only for the cases where the binaries could be compiled.

Metric 28. Count of the function variants, out of the function variants that were true for Metric 27, that for any compiler optimization level (without the memory checks or memory address sanitizer utility used in the compilation) was output-verified. Note that in practice, we never observed an example of this case ever occurring.

Metric 29. Count of function variants that had any Valgrind detected `Invalid read` error (detected using either gcc or clang compiled binaries or both). Note that necessarily these counts are only for the cases where the binaries could be compiled.

Metric 30. Count of the function variants, out of the function variants that were true for Metric 29, that for any compiler optimization level (without the memory checks or memory address sanitizer) was output-verified. Note that in practice, we never observed an example of this case ever occurring.

Metric 31. Count of function variants that had any Valgrind detected `Use of uninitialised value` error (detected using either gcc or clang compiled binaries or both). Note that necessarily these counts are only for the cases where the binaries could be compiled.

Metric 32. Count of the function variants, out of the function variants that were true for Metric 31, that for any compiler optimization level (without the memory checks or memory address sanitizer) was output-verified.

Metric 33. Count of function variants that had any Valgrind detected `Conditional jump or move depends on uninitialised value` error (detected using either gcc or clang compiled binaries or both). Note that necessarily these counts are only for the cases where the binaries could be compiled.

Metric 34. Count of the function variants, out of the function variants that were true for Metric 33, that for any compiler optimization level (without the memory checks or memory address sanitizer) was output-verified.

Metric 35. Count of function variants that had any clang or gcc memory sanitizer detected integer overflow error (detected using either gcc or clang compiled binaries or both). Note that necessarily these counts are only for the cases where the binaries could be compiled.

Metric 36. Count of the function variants, out of the function variants that were true for Metric 35, that for any

compiler optimization level (without the memory checks or memory address sanitizer) was output-verified. Note that in practice, we never observed an example of this case ever occurring.
Metric 37. Count of function variants that had any clang or gcc memory sanitizer detected out of bounds error (detected using either gcc or clang compiled binaries or both). Note that necessarily these counts are only for the cases where the binaries could be compiled, and the memory sanitizer check is performed on the binaries compiled using an optimization level of 0.
Metric 38. Count of the function variants, out of the function variants that were true for Metric 37, that for any compiler optimization level (without the memory checks or memory address sanitizer) was output-verified.
Metric 39. Count of how many of the function variants produce binaries that output hashes that are not correct to the SHA-1 implementation (in particular none of the hashes for any of the four test vectors are correct, and all of the hashes have an absolute character distance greater than 5 away from the correct SHA-1 hash), the output does not change depending on the compiler optimization settings that were used (e.g., it is compiler optimization stable), and the output conforms to the basic requirements of a hash function - in this case meaning that there are 40 hexadecimal characters produced for each test vector (which is the same length as the correct SHA-1 hash digests), and the output hashes change (by at least one character) for each of the four test vectors. This case is designed to be disjoint to Metrics 17, 18, 19 and 20, meaning there is no overlap between this Metric and those. These cases are interesting because some of these produce very bad checksums (e.g., clear repeating patterns), but others produce "hashes" that appear to be reasonably high entropy. These test cases are not further analyzed in detail for how secure they are (for example, if there are clear correlations between the input and the output), but are notable because these could in theory be used as (likely bad and non-secure) hash functions – which were produced as a byproduct of the high variability GPT code re-writing output. These counts include only the cases that are not optimization unstable so as to simplify the example test cases.
Metric 40. Count of how many of the function re-writes, for any compiler optimization setting, where there was any hash output that produced a number of characters not equal to 40.

Note that many of the function re-writes may fall into more than one of these categories (Metrics). Also note that the counts of the various function variants that are incorrect can contain duplicate source code, similarly to the correct function rewrites. Duplicates of incorrect function versions is not checked for, but it does occur in at least a few instances.

The use of varying compiler optimization levels is motivated by the following points:

- Higher optimization could uncover source code variants which are quite similar with some minimal changes thus making them not direct copies, but have sufficient similarity that the optimization can produce identical binaries.
- The low optimization level shows a baseline equivalence of the original source code to very minimally modified code (such as minimally changing variable names), thus not producing a meaningful substantive syntax change.
- The fast code optimization option is tested because it can yield even more heavily modified binaries undefined behavior can be revealed
- Interestingly, in some instance the higher optimization levels allow the compiler to generate binaries whereas for the no-optimization level the compiler was not able to produce a binary.

In summary, optimization in the compiler can uncover cases where although the source syntax is different, the underlying logic and algorithmic choices are the same. This allows us to use compiler optimization as a tool to differentiate genuine source code alterations that are meaningful. With the intention of thoroughly checking for variant equivalences that may be difficult to arrive at, both the compiler tools `gcc` and `clang` are executed with all available optimization levels.

The motivation for the test cases that detect binaries whose output is unstable based on different compiler optimization settings is that a reasonably large number of these cases were found in the GPT function re-writes. In particular, many of the function re-writes have undefined behaviors. This then causes the compiler to have some freedom in how to interpret the undefined source code, and this can result in compiler optimization instability (also known as undefined behavior), which is a well-studied aspect of the C language [64–73]. Additionally, these tests are performed to categorize in what ways the output changes based on the compiler settings because there have been examples of vulnerabilities introduced by compilers [64, 74–76], and therefore it is of considerable interest to determine what is being affected by these compiler optimization unstable GPT function re-writes when different optimization levels are applied. It is difficult to systematically categorize undefined behavior and undefined syntax in a piece of C code, but what we found is that the binaries with compiler optimization instability often threw various compiler warnings including incorrect C syntax, such as incorrect type conversions. We leave more extensive analysis of undefined C syntax produced from GPT models to future research.

The hash output correctness is measured by the testing source code (given in Appendix B). Specifically, the hash data is written to an array and is intended to be in a specific index range of that array. The GPT modified source code may write out of bounds, but only the intended portion of that array is checked for algorithmic correctness of the hash function and writes outside of that array are not checked. The produced text from the compiled binaries are encoded and then decoded as utf-8 strings.

All compilation and execution was performed on Ubuntu, with `Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz` CPUs. The compiler versions are `gcc (Ubuntu 10.5.0-1ubuntu1 22.04) 10.5.0` and `Ubuntu clang version 14.0.0-1ubuntu1.1`. All code was compiled using C standard C11. The use the compilers on the fixed hardware platforms allowed for consistent binaries to be compiled and compared. The Valgrind version used for all tests is `Valgrind-3.18.1`.

2.2.1 Correct Function Rewrite Composability

The last step in generating full cryptographic function variants is composing the source code functions that were determined to be correct and compilable under all optimization settings. This is done by randomly selecting a representative source code function from each of the unique meta-clusters computed by **Metric 13** in Section 2.2. Then, all combinations of the full SHA-1 implementations are enumerated through, where we replace each of the four component functions with a re-written unique version. These re-written versions now have distinctly different source code from the original implementation for all of the component functions. These re-written versions are then processed by the same ensemble of tests that were performed on the single-function replacement tests in Section 2.2.

3 Results

The aggregated test statistics defined in Section 2.2 are all reported, for each component function of the reference implementation, in Table 1. These metrics include a wide range of different inference temperature settings, the 10 different prompts, and are also aggregated from all 3 GPT models. The most consequential metrics are the total number of *attempted* function re-writes, which is 137, 466 in total, but only 46, 962 of those could be compiled with all compiler settings. Fundamentally, this means that more than half of the GPT outputs could not be parsed as valid C code (using the markdown-style prompts as described in Sections 2.1 and 2.2). Note that the total number of function re-writes entry would be the exact same count (and exactly equal to the total number of GPT queries made) if not for the last parsing step (step number 7 in Section 2.2), where we occasionally get *extra* function re-writes from ancillary text produced by the GPT models.

Table 1 shows that there are a large number of GPT function re-writes that are instances of very flawed C code. For example, Metrics 31 and 33 in the table total over 1,000 instances, and both of these are measures that show fundamental issues with the code implementation.

The function `sha1_init` defines several constants that initialize the SHA-1 algorithm. Metrics 12 and 13 in Table 1 show quantify correct function re-writes, and one of the interesting questions to ask specifically about the function re-writes of `sha1_init` is whether any of those functions define constants that are not the same as the original implementation (which is given in Appendix B). The answer is that all of the constants defined in these variants were correct, and not different compared to the original source code. Many of these variants add ancillary constants, or define data-structures that are not used, but fundamentally the correct constants are always defined.

In regards to the versions that produced hash functions that were seemingly valid hash functions (characterized by Metric 39), but not correct to the reference algorithm, these examples are very likely not cryptographically secure (as in, they are very likely to have fundamental algorithmic weaknesses beyond implementation risks such as side channel attacks). The scope of this study is not to thoroughly evaluate these instances. However, these function re-write instances were unexpected results of these experiments, and they are indicators that open source GPT models can be tools for the proliferation of (incorrect) versions of important algorithms such as cryptographic algorithms. If such examples were minimally evaluated and found to have the basic requirements of hash functions, these could then be used (at fairly low cost) in new malware variants, thus obfuscating the functionality of the software. Even if these algorithms are insecure, they can be generated at scale, and therefore pose a risk to the community of cybersecurity analysts to the increase of availability of such tools. Therefore, these instances warrant future research.

Across all source code variants and compiler settings, the binary execution fatal errors that were encountered were 9369 `SIGSEGV` errors, 562 `SIGABRT` errors, 9 `SIGILL` errors, 3 `SIGFPE` errors, and 17 `SIGBUS` errors. Across all

Metric	sha1_final	sha1_init	sha1_update	sha1_transform
Function Rewrites	30,322	36,343	40,422	30,379
Metric 1	10,680	7,349	18,517	10,416
Metric 2□	10	17	24	4
Metric 3*	9,094	6,465	16,779	8,799
Metric 4□	1,104	770	634	1,316
Metric 5□	6	21	2	19
Metric 6*	7,690	6,025	14,667	1,060
Metric 7*	1,404	440	2,112	7,739
Metric 8□	10	0	38	3
Metric 9□	99	28	152	190
Metric 10□	15	5	14	12
Metric 11□	134	55	257	104
Metric 12*	48-23-17-18-17-18-54-22-17-17-21-17-22	116-53-38-39-38-39-118-55-56-56-59-56-61	121-76-73-74-71-74-149-74-69-69-70-69-70	67-29-18-24-20-24-90-19-17-17-19-17-19
Metric 13*	13 groups: 3-839-5-1-57-1-1-1-1-1-1-1-1	32 groups: 19-14-20-3-2-2-3-1-1-4-1-2-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1	58 groups: 416-126-31-60-2-2-3-5-7-23-4-5-7-6-10-22-10-1-1-1-1-2-8-1-9-1-1-3-2-2-4-1-2-3-1-1-1-5-1-1-1-1-1-1-2-1-1-3-2-1-1-1-1-1-1-1-1	8 groups 1-1-1-1-1-1-1-1
Metric 14*	490	346	1,298	7,454
Metric 15□	0	13	1	0
Metric 16□	0	0	0	0
Metric 17□	142	12	655	5
Metric 18□	108	8	70	0
Metric 19□	1	0	0	0
Metric 20□	1	0	0	0
Metric 21□	381	107	212	220
Metric 22□	953	675	1,078	1,101
Metric 23□	0	3	1	30
Metric 24□	0	0	1	0
Metric 25□	3	1	1	1
Metric 26□	0	0	0	0
Metric 27□	1	0	2	0
Metric 28□	0	0	0	0
Metric 29□	2	0	1	2
Metric 30□	0	0	0	0
Metric 31□	611	215	377	221
Metric 32□	0	0	0	0
Metric 33□	614	222	381	223
Metric 34□	0	3	1	0
Metric 35□	0	9	0	14
Metric 36□	0	0	0	0
Metric 37□	113	170	54	17
Metric 38□	1	12	3	0
Metric 39□	655	662	190	954
Metric 40□	2	0	1	0

Table 1: SHA-1 C code rewriting metrics (across all GPT models, prompts, and inference temperatures). * denotes function variant metrics that are correct re-writes □ denotes function variant counts that have an implementation flaw of some type causing code instability, compiler optimization instability, infinite loops, critical errors, or not correct SHA-1 implementations.

source code variants and compiler settings 943 of the compiled binaries reached an apparent infinite loop state, as determined by a timeout check of 10 seconds.

The different hash string output lengths, as determined by Metric 40, across all of the function re-writes and compiled binaries are; $40, 0, 160, 15728680$. The vast majority of the generated hashes have length of 40, but one function re-write produces hashes (for some compiler optimization settings) of length 0 (meaning an empty string), one case with a hash of length 160, and a different function re-write produces hashes of length 15728680 (for some compiler optimization settings), a majority of these characters are zeroes. Note that the testing code (Code Listing 12) is set up to read a specified number of indices from the array in which the hash data is computed.

Figure 1 shows compiled binary visualizations for 4 of the compiled SHA-1 binaries. These examples include re-writes that execute SHA-1 algorithms correctly, and re-writes that are incorrect and cause the output to be incorrect. These visualizations were generated using the tool binocle [77], with consistent binary data layout dimensions of the visual window and all default visualization settings otherwise. Note that these visualizations necessarily include the entirety of the testing code (the exact syntax of which is shown in Code Listing 12), and the only differences between each of the binaries is at most one of the four SHA-1 component functions being changed. There are clearly some variations that can be seen in these compiled binaries, but overall their structure is quite similar.

Figure 2 contain renderings of several of the connected components of the graphs produced by the correct SHA-1 re-write clustering, which are summarized by Metrics 12 and 13 in Table 1. Connections (e.g., paths and connected regions) of these graphs indicate that for some compiler optimization setting, an identical hash was produced for another piece of code (potentially the same code) with another compiler optimization setting. These connected graph components thus show function re-writes that are actually performing the same computation once compiled, even though the exact syntax of the code may be different. These graphs are all undirected. These graphs were generated using the Python 3 libraries Networkx [78] and Matplotlib [79, 80], and drawn using the kamada-kawai layout algorithm.

3.1 Function Composability Results

A total of $13 \cdot 32 \cdot 58 \cdot 8 = 193024$ different composed SHA-1 codebase re-writes were generated. This comes from all of the possible combinations of the unique and correct versions of the function re-writes, as determined by the compiled binary clustering of Metric 13 in Table 1. Of these 193024 versions, a vast majority are entirely correct, new, and unique variants of the SHA-1 codebase where they are all compilable, output-verified, they have no out-of-bounds writes, no integer overflows, no memory leaks, and are compiler optimization stable. Interestingly, 464 of these function re-writes however could not be compiled, but the underlying cause was due to re-definitions of C functions (specifically, functions with names that are not in the original SHA-1 source code) with conflicting type information. These ancillary functions that were produced could be removed in an automated system and then these conflicting definition errors would be mitigated, but here we did not apply this further step. Of the versions that could be compiled, using the SHA-256 checksums of the compiled binaries it was determined that all of these variants are all unique (e.g., there are not duplicates) under all compiler optimization settings using both gcc and clang, and are not the same as the original source code. Code Listings 1, 13, 14 are three examples of composed SHA-1 codebase re-writes that are fully correct and unique versions of the SHA-1 source code.

3.2 Example Function Variants and Compiled Binary Outputs

This section lists a number of concrete function re-write examples produced by the GPT models that exhibit a wide range of the behaviors that are summarized in Table 1. These include function re-writes that have memory leaks, that generate partially correct SHA-1 hashes, that have compiler optimization instability, among many other interesting software flaws. To conserve the total amount of space used for providing these functions, single instances will be listed in this section, and then many additional examples are given in Appendix D and Appendix E. The source code examples include syntax highlighting for C keywords, and if the compiled binary produced output, the text of the generated hashes are shown below the source code. The text of the hashes are color-coded such that black text denotes the hexadecimal character matches the corresponding character in the (correct) SHA-1 hash, and otherwise the character is red. The output of the hashes are ordered sequentially starting at test 1, and ending at test 4 (the test vectors are given in Appendix A).

```
void sha1_init(SHA1_CTX *ctx){
    ctx->data[0] = ctx->data[1] = ctx->data[2] = ctx->data[3] = ctx->data[4] = 0x00;
    ctx->data[5] = ctx->data[6] = ctx->data[7] = ctx->data[8] = ctx->data[9] = ctx->data[10] = ctx->data[11] = ctx->data[12] = ctx->data[13] = ctx->data[14]
          = ctx->data[15] = 0xFF;
    ctx->data[16] = ctx->data[17] = ctx->data[18] = ctx->data[19] = ctx->data[20] = ctx->data[21] = ctx->data[22] = ctx->data[23] = ctx->data[24] = ctx->
        data[25] = ctx->data[26] = ctx->data[27] = ctx->data[28] = ctx->data[29] = ctx->data[30] = 0x00;
    ctx->data[31] = ctx->data[32] = ctx->data[33] = ctx->data[34] = ctx->data[35] = ctx->data[36] = ctx->data[37] = ctx->data[38] = ctx->data[39] = ctx->
          data[40] = ctx->data[41] = ctx->data[42] = ctx->data[43] = ctx->data[44] = ctx->data[45] = ctx->data[46] = ctx->data[47] = ctx->data[48] = ctx->
          data[49] = ctx->data[50] = ctx->data[51] = ctx->data[52] = ctx->data[53] = ctx->data[54] = ctx->data[55] = 0xFF;
    ctx->datalen = 0;
    ctx->bitlen = 0;
    // initialize with values defined in the original implementation
    ctx->state[0] = 0x67452301;
    ctx->state[1] = 0xEFCDAB89;
    ctx->state[2] = 0x98BADCFE;
    ctx->state[3] = 0x10325476;
    ctx->state[4] = 0xc3d2e1f0;
    ctx->k[0] = 0x5a827999;
    ctx->k[1] = 0x6ed9eba1;
    ctx->k[2] = 0x8f1bbcdc;
    ctx->k[3] = 0xca62c1d6;
}
void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len){
        size_t i;
        for (i = 0; i < len; ++i) {
                ctx->data[ctx->datalen] = data[i];
                ctx->datalen++;
                if (ctx->datalen == 64) {
                        sha1_transform(ctx, ctx->data);
                        ctx->bitlen += 512;
                        ctx->datalen = 0;
                        ctx->k[0] = 0x5a827999;
                        ctx->k[1] = 0x6ed9eba1;
                        ctx->k[2] = 0x8f1bbcdc;
                        ctx->k[3] = 0xca62c1d6;
                }
        }
}
void sha1_transform(SHA1_CTX *ctx, const BYTE data[]){
    WORD a = ctx->state[0];
    WORD b = ctx->state[1];
    WORD c = ctx->state[2];
    WORD d = ctx->state[3];
    WORD e = ctx->state[4];
    WORD *m = (WORD *) malloc(80 * sizeof(WORD));
    for (int i = 0, j = 0; i < 16; ++i, j += 4){
        m[i] = (data[j] << 24) + (data[j + 1] << 16) + (data[j + 2] << 8) + (data[j + 3]);
    }
    for (int i = 16; i < 80; ++i){
        m[i] = (m[i - 3] ^ m[i - 8] ^ m[i - 14] ^ m[i - 16]);
        m[i] = (m[i] << 1) | (m[i] >> 31);
    }
    for (int i = 0; i < 20; ++i){
        WORD t = ROTLEFT(a, 5) + ((b & c) ^ (~b & d)) + e + ctx->k[0] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (int i = 20; i < 40; ++i){
        WORD t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + ctx->k[1] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (int i = 40; i < 60; ++i){
        WORD t = ROTLEFT(a, 5) + ((b & c) ^ (b & d) ^ (c & d))  + e + ctx->k[2] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (int i = 60; i < 80; ++i){
        WORD t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + ctx->k[3] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    ctx->state[0] += a;
    ctx->state[1] += b;
    ctx->state[2] += c;
    ctx->state[3] += d;
    ctx->state[4] += e;
    free(m);
}
void sha1_final(SHA1_CTX *ctx, BYTE hash[]){
    WORD i;
    i = ctx->datalen;
    if (ctx->datalen < 56) {
        ctx->data[i++] = 0x80;
        while (i < 56)
            ctx->data[i++] = 0x00;
    } else {
        ctx->data[i++] = 0x80;
        while (i < 64)
            ctx->data[i++] = 0x00;
        sha1_transform(ctx, ctx->data);
        memset(ctx->data, 0, 56);
    }
    ctx->bitlen += ctx->datalen * 8;
    ctx->data[63] = ctx->bitlen;
    ctx->data[62] = ctx->bitlen >> 8;
    ctx->data[61] = ctx->bitlen >> 16;
    ctx->data[60] = ctx->bitlen >> 24;
    ctx->data[59] = ctx->bitlen >> 32;
    ctx->data[58] = ctx->bitlen >> 40;
    ctx->data[57] = ctx->bitlen >> 48;
    ctx->data[56] = ctx->bitlen >> 56;
    sha1_transform(ctx, ctx->data);
    for (i = 0; i < 4; ++i) {
        BYTE temp1 = ctx->state[0] >> (24 - i * 8);
        BYTE temp2 = ctx->state[1] >> (24 - i * 8);
        BYTE temp3 = ctx->state[2] >> (24 - i * 8);
        BYTE temp4 = ctx->state[3] >> (24 - i * 8);
        BYTE temp5 = ctx->state[4] >> (24 - i * 8);
        hash[i] = temp1 & 0x000000ff;
        hash[i + 4] = temp2 & 0x000000ff;
        hash[i + 8] = temp3 & 0x000000ff;
        hash[i + 12] = temp4 & 0x000000ff;
        hash[i + 16] = temp5 & 0x000000ff;
    }
}
```

Listing 1: **Fully correct SHA-1 codebase GPT model re-write example Number 1. Each of these four functions were manually selected from the correct and unique re-writes of that function across all of the GPT model outputs. Minimal formatting of the raw parsed GPT output has been applied so as to reduce the amount of whitespace.**

```
void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len) {
    size_t i;
    for (i = 0; i < len; ++i) {
        ctx->data[ctx->datalen] = data[i];
        ctx->datalen = (ctx->datalen + 1) % 64;
        if (ctx->datalen == 0) {
            sha1_transform(ctx, ctx->data);
            ctx->bitlen += 512;
            ctx->datalen = 63;
        }
    }
}

Compiled Binary output:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
3f1b7c80b54ad4d677b4dbbf81dd21b952391589
04575f6b701b0333133f720bc5c1353844075b57
```

Listing 2: Example GPT function re-write of sha1_update produced from Llama-2-70b-chat-hf with inference temperature 0.9 and prompt number 2. This function re-write is an instance of the function category found by the Metric 17 definition, in this case where the generated hashes were correct for 3 out of the 4 test vectors. The output for the four test vectors are given below the source code function.

```
void sha1_final(SHA1_CTX *ctx, BYTE hash[])
{
        WORD i;
        if (ctx->datalen < 56) {
                ctx->data[i++] = 0x80;
                while (i < 56)
                        ctx->data[i++] = 0x00;
        }
        else {
                ctx->data[i++] = 0x80;
                while (i < 64)
                        ctx->data[i++] = 0x00;
                sha1_transform(ctx, ctx->data);
                memset(ctx->data, 0, 56);
        }
        ctx->bitlen += ctx->datalen * 8;
        ctx->data[63] = ctx->bitlen;
        ctx->data[62] = ctx->bitlen >> 8;
        ctx->data[61] = ctx->bitlen >> 16;
        ctx->data[60] = ctx->bitlen >> 24;
        ctx->data[59] = ctx->bitlen >> 32;
        ctx->data[58] = ctx->bitlen >> 40;
        ctx->data[57] = ctx->bitlen >> 48;
        ctx->data[56] = ctx->bitlen >> 56;
        sha1_transform(ctx, ctx->data);
        for (i = 0; i < 4; ++i) {
                hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
        }
}

Compiled Binary output for gcc with optimization level 0:
f1e2460eabace92790b22ff4f510265147c19a11
99c7bc21b9ab381e981099bfa8f2052b33fa7629
34aa973cd4c4daa4f61eeb2bdbad27316534016f
99c7bc21b9ab381e981099bfa8f2052b33fa7629

Compiled Binary output for clang with optimization level 1:
6ac40abea5f9533272983adba42c4b714eef27a7
28c0eb8a570cfbd7371d9df87263990d96dcd3da
88bb5e1de3a99fcb9597ea582cbacfd67ab73003
d93e65ac73cf1514f359d5d6909e47c5c56b914a

Compiled Binary output for clang with optimization level z:
99fb15b4449bf9c584693a76c1a0081140af2166
e1c6639d22274b76073b4ee227674110091b9bbf
6270d8ac7fc1834ba8b1cb6a78a06a4c7094bc56
e1c6639d22274b76073b4ee227674110091b9bbf
```

Listing 3: Example GPT function re-write of sha1_final produced from Mistral-7B-Instruct-v0.1 with inference temperature 0.5 and prompt number 4. This function re-write is an instance of the function category found by the Metric 18 definition, meaning that for some of the test vectors the generated hashes are correct, but the output changes depending on what compiler optimizations are used. The generated checksums for the four test vectors are given below the source code function, in this case for three specific compiler optimization settings (3 sets of 4 hexadecimal strings). The hash for the third test vector was correct for all gcc optimization settings and correct for clang with optimization level 0. All higher optimization levels used with clang cause the compiled binary to generate incorrect checksums.

```
void sha1_init(SHA1_CTX *ctx){
        ctx->datalen = 0;
        ctx->bitlen = 0;
        ctx->state[0] = 0x67452301;
        ctx->state[1] = 0xEFCDAB89;
        ctx->state[2] = 0x98BADCFE;
        ctx->state[3] = 0x10325476;
        ctx->state[4] = 0xc3d2e1f0;
        ctx->k[0] = 0x5a827999;
        ctx->k[1] = 0x6ed9eba1;
        ctx->k[2] = 0x8f1bbcdc;
        ctx->k[3] = 0xca62c1d6;
        ctx->state[0] = ctx->state[0] >> 32;
        ctx->state[1] = ctx->state[1] >> 32;
        ctx->state[2] = ctx->state[2] >> 32;
        ctx->state[3] = ctx->state[3] >> 32;
        ctx->state[4] = ctx->state[4] >> 32;
        ctx->k[0] = ctx->k[0] >> 32;
        ctx->k[1] = ctx->k[1] >> 32;
        ctx->k[2] = ctx->k[2] >> 32;
        ctx->k[3] = ctx->k[3] >> 32;
}

gcc with optimization level 0:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
34aa973cd4c4daa4f61eeb2bdbad27316534016f
04575f6b701b0333133f720bc5c1353844075b57

gcc with optimization level 1:
875d917812fd049087475128224cc666d663db07
7c449d2dfa4434b572a9481a188f6ca440558a14
aabc79811c6723281f12092cf3f82364b59a0f83
43b8a7cdc2f60c8ae3cb9683e2d14656f9eb58fa

gcc with optimization level fast:
875d917812fd049087475128224cc666d663db07
7c449d2dfa4434b572a9481a188f6ca440558a14
aabc79811c6723281f12092cf3f82364b59a0f83
43b8a7cdc2f60c8ae3cb9683e2d14656f9eb58fa

clang with optimization level 0:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
34aa973cd4c4daa4f61eeb2bdbad27316534016f
04575f6b701b0333133f720bc5c1353844075b57

clang with optimization level 1:
78c394b7494f507e812551817a7ac5ccf34476b6
a858c6bfc15f835b51bb231349969599c4750f60
d4fe7bfee5b66615fb257e7d6cd57ce7b696d49f
1b8537b7290c655d64cba67dd3951dbefcee3513

clang with optimization level fast:
78c394b7494f507e812551817a7ac5ccf34476b6
a858c6bfc15f835b51bb231349969599c4750f60
d4fe7bfee5b66615fb257e7d6cd57ce7b696d49f
1b8537b7290c655d64cba67dd3951dbefcee3513
```

Listing 4: This function re-write is an example where the algorithmic correctness of the compiled binary is dependent on the compiler optimization setting that is used (this is categorized by Metric 5). Specifically, using optimization level 0 with both gcc and clang result in the binary correctly producing SHA-1 hashes, but any higher optimization levels cause the binary to produce incorrect SHA-1 hashes. The incorrect hashes are the same for all higher levels of gcc optimization (optimization levels 1 and fast are shown as examples), and the incorrect hashes for the compiled clang binaries are different from the incorrect gcc compiled binaries (optimization levels 1 and fast are also shown for clang as examples). Notably, visually the incorrect hashes seem to have reasonably high entropy and do not show obvious low-entropy correlations. This function re-write was generated by Mistral-7B-Instruct-v0.1 with inference temperature 0.4 with prompt number 4.

```
void sha1_final(SHA1_CTX *ctx, BYTE *hash){
        WORD i;
        i = ctx->datalen;
        if (ctx->datalen < 56) {
                ctx->data[i++] = 0x80;
                while (i < 56)
                        ctx->data[i++] = 0x00;
        }
        else {
                ctx->data[i++] = 0x80;
                while (i < 64)
                        ctx->data[i++] = 0x00;
                sha1_transform(ctx, ctx->data);
                memset(ctx->data, 0, 56);
        }
        ctx->bitlen += ctx->datalen * 8;
        ctx->data[63] = ctx->bitlen;
        ctx->data[62] = ctx->bitlen >> 8;
        ctx->data[61] = ctx->bitlen >> 16;
        ctx->data[60] = ctx->bitlen >> 24;
        ctx->data[59] = ctx->bitlen >> 32;
        ctx->data[58] = ctx->bitlen >> 40;
        ctx->data[57] = ctx->bitlen >> 48;
        ctx->data[56] = ctx->bitlen >> 56;
        sha1_transform(ctx, ctx->data);
        for (i = 0; i < 4; ++i) {
                hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x0000007f;
                hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x0000007f;
                hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x0000007f;
                hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x0000007f;
                hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x0000007f;
        }
}

Compiled Binary output:
29193e364706016a3a3e25717850426c1c50581d
04183e441c3b526e3a2e4a21795129656546707 1
342a173c54445a24761e6b2b5b2d27316534016f
04575f6b701b0333133f720b4541353844075b57
```

Listing 5: Example function re-write, that does not produce correct SHA-1 hashes, and where at least one of the generated hashes is very close to the correct SHA-1 hash, in this case by being incorrect by only two hexadecimal characters. This function re-write was generated by Mistral-7B-Instruct-v0.1 with inference temperature 0.9 with prompt number 2.

```
void sha1_transform(SHA1_CTX *ctx, const BYTE data[]) {
    WORD a, b, c, d, e, i, j, t, m[80];
    // Initialize variables
    a = ctx->state[0];
    b = ctx->state[1];
    c = ctx->state[2];
    d = ctx->state[3];
    e = ctx->state[4];
    // Process input data
    for (i = 0; i < 16; ++i) {
        m[i] = (data[j] << 24) + (data[j + 1] << 16) + (data[j + 2] << 8) + (data[j + 3]);
        j += 4;
    }
    for (; i < 80; ++i) {
        m[i] = (m[i - 3] ^ m[i - 8] ^ m[i - 14] ^ m[i - 16]);
        m[i] = (m[i] << 1) | (m[i] >> 31);
    }
    // Perform transformations
    for (i = 0; i < 20; ++i) {
        t = ROTLEFT(a, 5) + ((b & c) ^ (~b & d)) + e + ctx->k[0] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 40; ++i) {
        t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + ctx->k[1] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 60; ++i) {
        t = ROTLEFT(a, 5) + ((b & c) ^ (b & d) ^ (c & d))  + e + ctx->k[2] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 80; ++i) {
        t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + ctx->k[3] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    // Update state
    ctx->state[0] += a;
    ctx->state[1] += b;
    ctx->state[2] += c;
    ctx->state[3] += d;
    ctx->state[4] += e;
}

Compiled Binary output for gcc with optimization level s:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
34aa973cd4c4daa4f61eeb2bdbad27316534016f
04575f6b701b0333133f720bc5c1353844075b57

Compiled Binary output for clang with optimization level 1:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
34aa973cd4c4daa4f61eeb2bdbad27316534016f
04575f6b701b0333133f720bc5c1353844075b57

Compiled Binary output for clang with optimization level 2:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
34aa973cd4c4daa4f61eeb2bdbad27316534016f
04575f6b701b0333133f720bc5c1353844075b57

Compiled Binary output for clang with optimization level 3:
da4968eb2e377c1f884e8f5283524bebe74ebdbd
313908d89e04cb0b2c0bc8e96de12aaa473a8dbe
c22cca10baa841079e00d5b4eb5d539d1d85e6d1
21c639842699549fbcf755c80c61f0e74228a80a

Compiled Binary output for clang with optimization level s:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
34aa973cd4c4daa4f61eeb2bdbad27316534016f
04575f6b701b0333133f720bc5c1353844075b57

Compiled Binary output for clang with optimization level fast:
da4968eb2e377c1f884e8f5283524bebe74ebdbd
313908d89e04cb0b2c0bc8e96de12aaa473a8dbe
c22cca10baa841079e00d5b4eb5d539d1d85e6d1
21c639842699549fbcf755c80c61f0e74228a80a

Compiled Binary output for clang with optimization level z:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
34aa973cd4c4daa4f61eeb2bdbad27316534016f
04575f6b701b0333133f720bc5c1353844075b57
```

Listing 6: Example function re-write of sha1_transform where for some compiler optimization settings the compiled binary, when executed, results in a fatal error, but for at least one other optimization setting the compiled binary correctly produces SHA-1 hashes. For 6 of the compiler optimization settings (specifically gcc with levels 0, 1, 2, 3, fast and clang with level 0), the compiled binary threw a Signals.SIGSEGV fatal error. The binaries compiled using gcc with optimization level s, and clang with optimization levels 1, 2, s, z correctly generated SHA-1 hashes. This was produced by Llama-2-70b-chat-hf with prompt number 2 and inference temperature 0.5.

```
void sha1_init(SHA1_CTX *ctx)
{
        ctx->datalen = 0;
        ctx->bitlen = 0;
        ctx->state[0] = 0x67452301;
        ctx->state[1] = 0xEFCDAB89;
        ctx->state[2] = 0x98BADCFE;
        ctx->state[3] = 0x10325476;
        ctx->state[4] = 0xc3d2e1f0;
        ctx->k[0] = 0x5a827999;
        ctx->k[1] = 0x6ed9eba1;
        ctx->k[2] = 0x8f1bbcdc;
        ctx->k[3] = 0xca62c1d6;
        ctx->state[0] ^= ctx->k[0];
        ctx->state[1] ^= ctx->k[1];
        ctx->state[2] ^= ctx->k[2];
        ctx->state[3] ^= ctx->k[3];
        ctx->state[4] ^= ctx->k[4];
}

Compiled Binary output:
4275ecaf350971015ed63376c29d8d8783187bb1
34a4a1173f53d9d7e2915ded83654efac8393b2c
a50d266b64be8917a269d7556861e34a5b0bbff6
c21443d878583e49864a7cb4920a305c744a6ab9
```

Listing 7: Example of a function re-write where the generated hashes have some of the basic requirements of a valid hash function, and have the same hexadecimal character length as SHA-1 hashes (40 characters). This code did not produce different output based on different compiler optimization settings. This is an example of a case found by Metric 39. This function re-write was generated by Mistral-7B-Instruct-v0.1 with inference temperature 0.6 with prompt number 4. Additionally, this function re-write had at least one detected out-of-bounds error by the automated memory sanitizer check.

```
void sha1_final(SHA1_CTX *ctx, BYTE hash[]){
    WORD i;
    BYTE *padded_data = malloc(ctx->datalen + 56);
    for (i = 0; i < ctx->datalen; ++i) {
        padded_data[i] = ctx->data[i];
    }
    for (i = ctx->datalen; i < 56; ++i) {
        padded_data[i] = 0x00;
    }
    ctx->bitlen += ctx->datalen * 8;
    padded_data[ctx->datalen] = ctx->bitlen;
    padded_data[ctx->datalen + 1] = ctx->bitlen >> 8;
    padded_data[ctx->datalen + 2] = ctx->bitlen >> 16;
    padded_data[ctx->datalen + 3] = ctx->bitlen >> 24;
    padded_data[ctx->datalen + 4] = ctx->bitlen >> 32;
    padded_data[ctx->datalen + 5] = ctx->bitlen >> 40;
    padded_data[ctx->datalen + 6] = ctx->bitlen >> 48;
    padded_data[ctx->datalen + 7] = ctx->bitlen >> 56;
    sha1_transform(ctx, padded_data);
    memset(padded_data, 0, ctx->datalen + 56);
    for (i = 0; i < 4; ++i) {
        hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
    }
}

Compiled binary output:
1242a46dffbfd1aa1f50cd0c0a9cec5fc296260a
ca1dc89ab706c485052661c66847fcbd7c17bde8
892014653d4dbd65184c2d28296349950c99ab5f
ae61f5a6bcbd06fe71c136cfa785b9ec10dc403e
```

Listing 8: Incorrect SHA-1 function implementation of sha1_final which causes a detected memory leak using Valgrind (for a binary compiled using optimization level 0 for either gcc or clang or both). Note that the compiled binary output for this example function does not change when different compiler optimization levels are applied. All of these output hash digests are very far away from the correct SHA-1 hash digests. This example was produced by Mistral-7B-Instruct-v0.1 using inference temperature of 0.6 and prompt number 4.

Code Listing 1 shows a complete SHA-1 re-written codebase example that is fully correct, and all of the component functions are distinct from the original source code (as determined by the compiled binary hashing clustering). In each of these functions, the code functions are selected arbitrarily from the entirety of the correct (and unique) function re-writes to serve as representative, and interesting, examples of the GPT code-writing results. Code Listings 13 and 14 in Appendix D shows two more example SHA-1 codebases that were completely re-written and unique.

Code Listing 2 shows a specific function re-write where three of the output checksums are correct, but one of them is not correct. Code Listing 3 shows a similar function re-write, except in that case the generated output varied based on the compiler optimization settings. Code Listing 4 shows an example of a function re-write which is correct for all 4 test vectors, if the code is compiled using specific optimization settings, and otherwise the output checksums are not at all correct.

Code Listing 5 shows a function re-write where the output checksums are not correct (e.g. not SHA-1 hashes), but are incorrect by only a few hexadecimal characters.

Code Listing 6 shows a function re-write that is compiler optimization unstable where for some settings the compiled binary correct produces SHA-1 hashes, for other settings it crashes in a fatal error, and still for other settings does produce output but the output is not correct SHA-1 hexadecimal hashes.

Code Listing 7 shows an example function re-write that produces incorrect checksums, but that have the interesting property of appearing to be good hashes (e.g. no apparent dependence on the input, and reasonably high entropy hexadecimal strings). Note that this specific example is not necessarily a good hash function – the notable thing is that the function re-write was quite minimal, and yet the compiled binary produced hexadecimal strings that were not obviously terrible checksums.

Code Listing 8 shows a function re-write example where the compiled binary had a detected memory leak by Valgrind. Interestingly, the compiled binary was able to be executed, and hexadecimal output was produced, however it was not correct SHA-1 hashes.

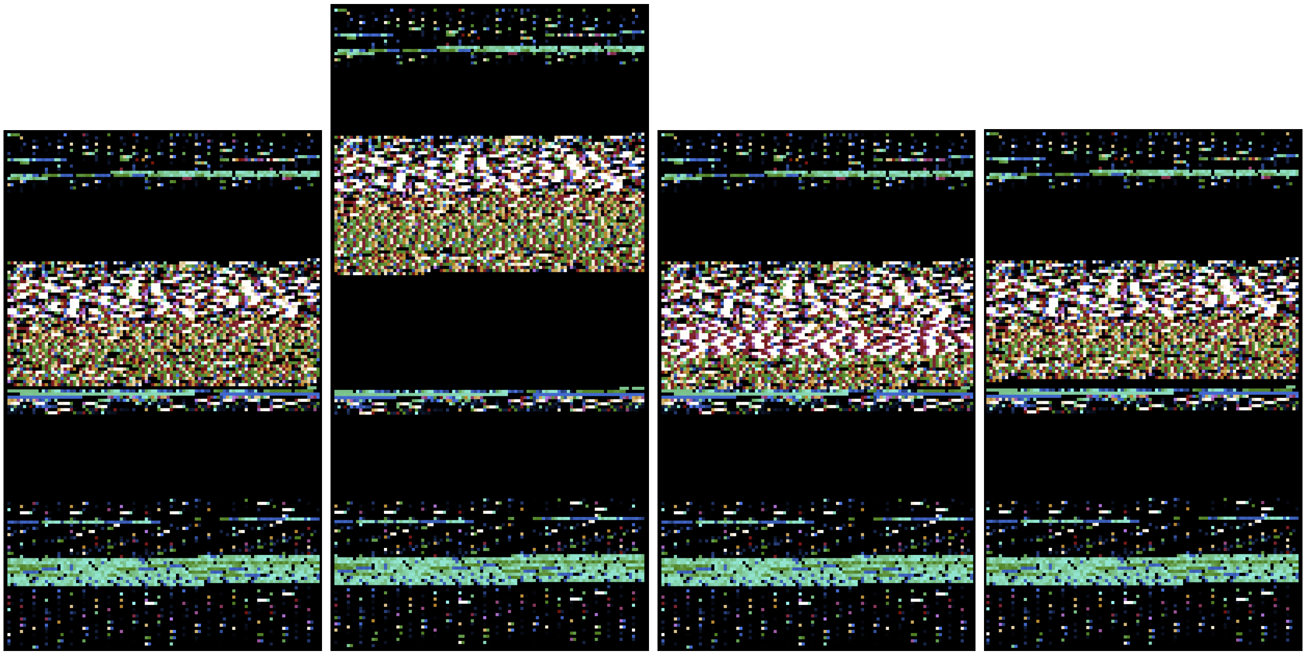

Figure 1: Visualized examples of compiled SHA-1 binaries using binocle. In order from top left to right; a single function-rewrite that is fully correct, 2 function rewrites where all test cases fail (and the outputs are not close to the correct SHA-1 hashes), and 1 function re-write that resulted in some of the test vectors producing a correct SHA-1 hash but failing for at least one test vector (and was not compiler optimization unstable). All of these example binaries were compiled using clang with an optimization level of 0. These binaries were arbitrarily selected as representative examples.

4 Discussion and Conclusion

This study shows that open source GPT models, without customized fine-tuning, can be used to construct correct algorithmic invariant implementations of a cryptographic hash function, in particular SHA-1. Code Listings 1, 13, 14 explicitly show three examples of re-written SHA-1 source code that maintain the same correct functionality of the original C source code. However, the success rate of function re-writes in terms of algorithmic correctness, or even being compilable, is quite low. Meaning that to assess a current GPT model for code re-writing capabilities, one needs to produce a large distribution of function re-writes and thoroughly test the characteristics of that output.

Fundamentally, using GPT models to produce source code, or as shown in this study to produce source code re-writes, is a cybersecurity risk for the integrity and stability of software development. The suite of tests that have been applied to these SHA-1 function re-writes has shown that GPT re-written functions can contain critical software flaws, some of which could be hard to detect without proper software validation and testing. GPT models have been shown to be incredibly effective at producing these SHA-1 re-writes at scale, but this capability of GPT models should serve as a research tool for testing interesting and useful versions of software, not as a solution for writing correct software. The tendency for GPT produced source code to contain bugs has been observed in several previous studies [48, 81–83], meaning the findings of this study are consistent with the existing literature. The remarkable finding here is that these faulty code implementations were generated as a byproduct of GPT models with prompts requesting *accurate* code function re-writes, as opposed to other studies where the aim was to induce the generation of faulty code from GPT models [84]. However, the other very notable finding of this study is the dramatic variety of code re-writes that were produced. While many of these re-writes were incorrect or contained serious software flaws, this suggests that GPT models offer a unique capability of producing large amounts of highly variable source code - which could be used for studying properties of computer code (namely, security properties), or fuzzing computer code. The code re-writing procedure described in this study should also be applied to implementations of other cryptographic algorithms to determine how well the GPT models perform at rewriting potentially even more complex code than this SHA-1 implementation.

This type of generative machine learning code re-writing could also be tested on malware source code, and will likely be used for this purpose in the wild. In the case of malware, the measurement of correctness is more difficult to capture - and potentially more dangerous in the case of undefined behavior. However, this study is an interesting

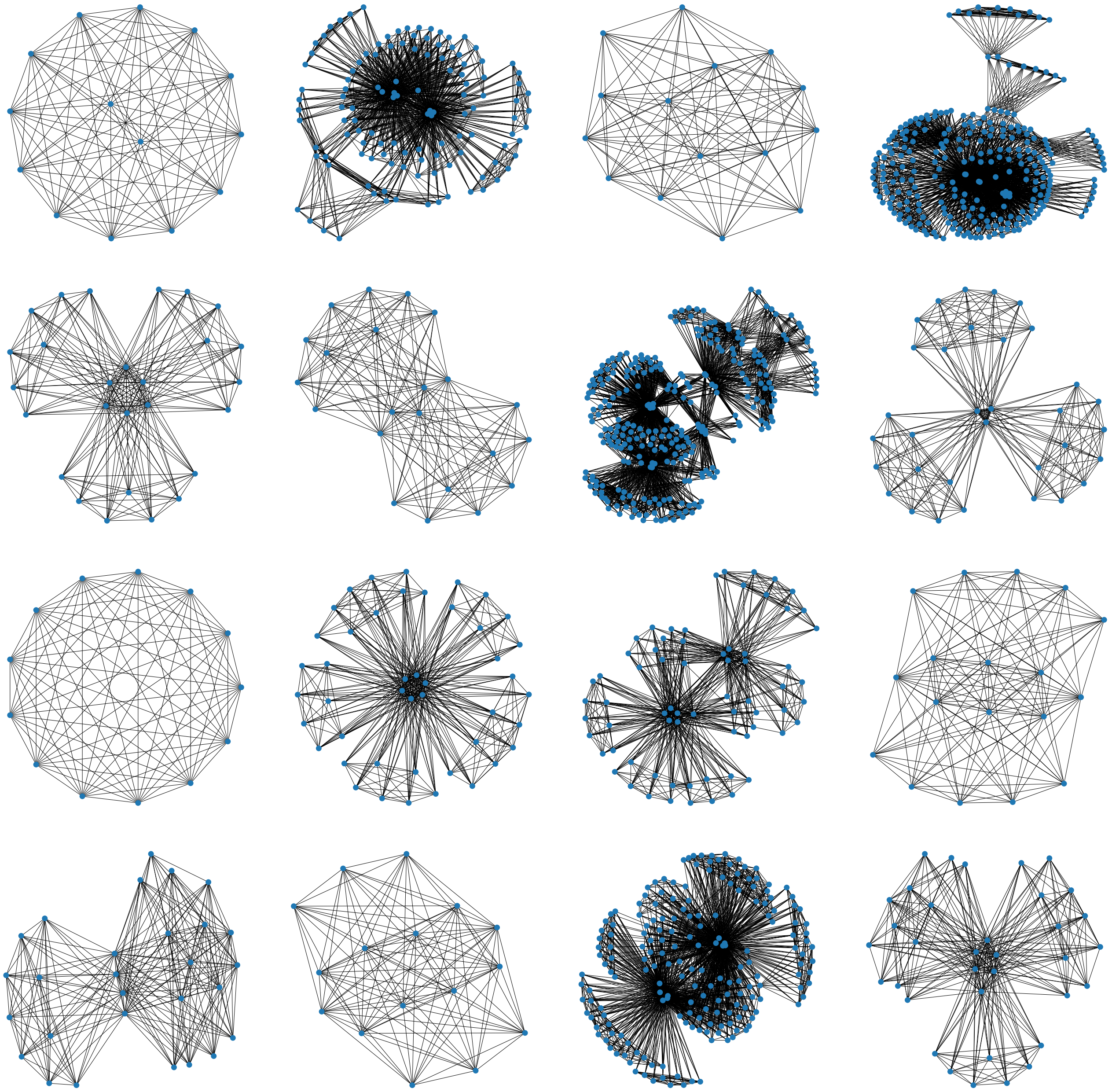

Figure 2: Graph renderings of various example connected components from the compiled binary clustering procedure; specifically for the GPT function re-writes of the SHA-1 C code where the compiled binary correctly produced SHA-1 hashes (and did not have fatal errors, or compiler optimization instability). Each node (blue) represents a tuple of a single SHA-1 component function re-write whose source code had Levenshtein character distance greater than 0 compared to the original source code (after repeated whitespace and code comments were removed) and one of the 13 compiler optimization settings used (either gcc or clang, with varying optimization levels). In other words, each node represents a single compiled binary that correctly executed the SHA-1 algorithm. Each edge represents the SHA-256 checksum of the compiled binary being equal for the two compiled binaries that the edge connects. These networks are not the comprehensive clustering of the correct SHA-1 rewrites, but they do represent a majority of the graphs that were produced. Notably, four of these graphs which are noticeably larger and more densely connected than the other graphs correspond to the graphs of function re-writes that are equivalent to the original source code due to the syntax changes made by the GPT models being relatively minimal. Each of these graphs are single connected components from the overall binary hashing clustering procedure, which is described in Section 2.2 and the summary statistics for are shown in Table 1.

first step in this direction - especially because cryptographic algorithms are commonly used in malware [85–92], for example when encrypting web traffic for command and control, or for disruption in the case of ransomware. Meaning that even GPT model re-writes of relatively small functions could further propagate malware variants.

The prompting of GPT models unintentionally producing hash function implementations that fit the basic properties of hash functions, but are not correct to the actual algorithm (in this case SHA-1), is also very notable (Code Listings 7, 27, 28, 29, 30, 31, 32, 33 are explicit examples of these instances, but there are other instances of this occurring in other function re-write examples namely where the output is compiler optimization unstable). Fundamentally, being able to easily produce hash function implementations using generative transformer models that are valid hash functions (but not necessarily secure *cryptographic* hash functions), will make malware analysis more challenging. Not only does this change the source code, and therefore signatures of the compiled binaries, but it also makes automated cryptographic algorithm detection in binaries [86–92] significantly more challenging since these rely on finding known cryptographic constants used in standard cryptographic algorithms. Future work should analyze how secure these incorrect (e.g., non-SHA-1) hash function implementations are, for example using cryptographic fuzzing tools [93–95]. More detailed analysis of the GPT rewritten code, specifically the compiled binaries, can be performed in future research - namely things such as memory usage, compute time efficiency, and detecting other potential security flaws such as side channel attack susceptibility.

The 10 prompts that were used in this study are selected to be reasonable hand-crafted prompts that attempt in, different ways, to extract useful source code. However, GPT prompts are effectively another hyperparameter that can be tuned to get good performance, and good natural language prompts may not be very obvious. There are a couple of proposed techniques [96, 97] for automatically constructing good natural language prompts, and it is likely that such methods could be used to get even better performance for GPT code re-writing.

Future research on source code re-writing using GPT models should be focused on domains where producing a large amount of different source code is useful. The challenge in utilizing GPT models for this is testing for correctness (although, perhaps there are situations where the source code does not need to be an exactly correct computation invariant of the original source code). The clearest case where this could be used is in pre-computing a large number of versions of known malware (where the source code, or compile-able disassembled code, is known) and then computing signatures of those binaries that are used for standard dynamic or static malware detection (such as fuzzy hashing [98, 99]). This set of signatures of the artificially created versions of the malware could then be used to detect those versions of the malware in the wild if a developer were to ever produce those versions. This capability is especially promising since it can be done in an entirely automated system − this does not require human developers to produce different versions of source code. This pre-computation of malware signatures would also help preemptively combat the likely future of GPT produced malware.

Another possible application for producing a large number of correct variants of source code is in optimizing the source code for a specific purpose - such as speed, or reduced memory usage. Because GPT models can generate a wide variety of code reasonably fast, they could be prompted to generate re-written and novel versions of code that have some desired characteristic such as executing faster. As seen in this study, it is likely that such code re-writes will not always be correct or adhere to the given prompt, but for a sufficiently large number of samples of GPT produced code, some versions could have the required properties.

Lastly, another aspect of this study which could be expanded on is the correctness analysis of the function re-writes which passed all of the SHA-1 tests. In principle, a cryptographic implementation passing these tests means that with very high likelihood the implementation is correct. However, we have also seen the numerous ways in which the GPT produced code can be incorrect, and therefore it is plausible, if unlikely, that the functions we have found to be re-written correctly actually contain interesting edge-case flaws.

5 Acknowledgments

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under contract DE-NA0003525. This written work is authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan.

References

[1] Mark Chen et al. *Evaluating Large Language Models Trained on Code.* 2021. arXiv: `2107.03374 [cs.LG]`.

[2] Ashish Vaswani et al. *Attention Is All You Need.* 2023. arXiv: `1706.03762 [cs.CL]`.

[3] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models.* 2023. arXiv: `2307.09288 [cs.CL]`.

[4] Tom B. Brown et al. *Language Models are Few-Shot Learners.* 2020. arXiv: `2005.14165 [cs.CL]`.

[5] OpenAI et al. *GPT-4 Technical Report.* 2023. arXiv: `2303.08774 [cs.CL]`.

[6] Tri Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness.* 2022. arXiv: `2205.14135 [cs.LG]`.

[7] Jianlin Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding.* 2023. arXiv: `2104.09864 [cs.CL]`.

[8] Noam Shazeer. *Fast Transformer Decoding: One Write-Head is All You Need.* 2019. arXiv: `1911.02150 [cs.NE]`.

[9] *Secure Hash Standard.* 2002. URL: `https://csrc.nist.gov/pubs/fips/180-2/final`.

[10] Nicky Mouha and Christopher Celi. *A Vulnerability in Implementations of SHA-3, SHAKE, EdDSA, and Other NIST-Approved Algorithms.* Cryptology ePrint Archive, Paper 2023/331. `https://eprint.iacr.org/2023/331`. 2023. DOI: `10.1007/978-3-031-30872-7_1`. URL: `https://eprint.iacr.org/2023/331`.

[11] Varun Satheesh and Dillibabu Shanmugam. "Implementation Vulnerability Analysis: A case study on ChaCha of SPHINCS". In: *2020 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS).* 2020, pp. 97–102. DOI: `10.1109/iSES50453.2020.00032`.

[12] Alessandro Cilardo and Nicola Mazzocca. "Exploiting Vulnerabilities in Cryptographic Hash Functions Based on Reconfigurable Hardware". In: *IEEE Transactions on Information Forensics and Security* 8.5 (2013), pp. 810–820. DOI: `10.1109/TIFS.2013.2256898`.

[13] Nicky Mouha et al. "Finding Bugs in Cryptographic Hash Function Implementations". In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 870–884. DOI: `10.1109/TR.2018.2847247`.

[14] Marc Stevens et al. "The first collision for full SHA-1". In: *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part I 37.* Springer. 2017, pp. 570–596.

[15] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. "Finding collisions in the full SHA-1". In: *Advances in Cryptology–CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005. Proceedings 25.* Springer. 2005, pp. 17–36.

[16] Eli Biham et al. "Collisions of SHA-0 and Reduced SHA-1". In: *Advances in Cryptology–EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005. Proceedings 24.* Springer. 2005, pp. 36–57.

[17] Pierre Karpman, Thomas Peyrin, and Marc Stevens. "Practical free-start collision attacks on 76-step SHA-1". In: *Annual Cryptology Conference.* Springer. 2015, pp. 623–642.

[18] Marc Stevens. "New collision attacks on SHA-1 based on optimal joint local-collision analysis". In: *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32.* Springer. 2013, pp. 245–261.

[19] Gaëtan Leurent and Thomas Peyrin. "SHA-1 is a Shambles: First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust". In: *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, Aug. 2020, pp. 1839–1856. ISBN: 978-1-939133-17-5. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/leurent`.

[20] Yingbo Song et al. "On the infeasibility of modeling polymorphic shellcode". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security.* CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, 541–551. ISBN: 9781595937032. DOI: `10.1145/1315245.1315312`. URL: `https://doi.org/10.1145/1315245.1315312`.

[21] *clang: a C language family frontend for LLVM.* `https://clang.llvm.org`.

[22] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: `10.1109/CGO.2004.1281665`.

[23] Chris Lattner. "LLVM and Clang: Next generation compiler technology". In: *The BSD conference.* Vol. 5. 2008, pp. 1–20.

[24] Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: *USENIX ATC 2012.* 2012. URL: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker.

[25] Nicholas Nethercote and Julian Seward. “Valgrind: A program supervision framework”. In: *Electronic notes in theoretical computer science* 89.2 (2003), pp. 44–66. DOI: 10.1016/S1571-0661(04)81042-9.

[26] Nicholas Nethercote. “Dynamic binary analysis and instrumentation”. In: (2004). DOI: 10.48456/tr-606.

[27] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *SIGPLAN Not.* 42.6 (2007), 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: https://doi.org/10.1145/1273442.1250746.

[28] Nicholas Nethercote and Julian Seward. “How to shadow every byte of memory used by a program”. In: *Proceedings of the 3rd International Conference on Virtual Execution Environments.* VEE ’07. San Diego, California, USA: Association for Computing Machinery, 2007, 65–74. ISBN: 9781595936301. DOI: 10.1145/1254810.1254820. URL: https://doi.org/10.1145/1254810.1254820.

[29] Julian Seward and Nicholas Nethercote. “Using Valgrind to Detect Undefined Value Errors with Bit-Precision.” In: *USENIX Annual Technical Conference, General Track.* 2005, pp. 17–30.

[30] Hyeokdong Kwon et al. *Novel Approach to Cryptography Implementation using ChatGPT.* Cryptology ePrint Archive, Paper 2023/606. https://eprint.iacr.org/2023/606. 2023. URL: https://eprint.iacr.org/2023/606.

[31] Pedro Valero-Lara et al. *Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation.* 2023. arXiv: 2309.07103 [cs.SE].

[32] Alexey Svyatkovskiy et al. “IntelliCode compose: code generation using transformer”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, 1433–1443. ISBN: 9781450370431. DOI: 10.1145/3368089.3417058. URL: https://doi.org/10.1145/3368089.3417058.

[33] Aishwarya Narasimhan, Krishna Prasad Agara Venkatesha Rao, and Veena M B. *CGEMs: A Metric Model for Automatic Code Generation using GPT-3.* 2021. arXiv: 2108.10168 [cs.AI].

[34] Luis Perez, Lizi Ottens, and Sudharshan Viswanathan. *Automatic Code Generation using Pre-Trained Language Models.* 2021. arXiv: 2102.10535 [cs.CL].

[35] Shailja Thakur et al. “VeriGen: A Large Language Model for Verilog Code Generation”. In: *ACM Trans. Des. Autom. Electron. Syst.* (2024). Just Accepted. ISSN: 1084-4309. DOI: 10.1145/3643681. URL: https://doi.org/10.1145/3643681.

[36] Immanuel Trummer. “CodexDB: synthesizing code for query processing from natural language instructions using GPT-3 codex”. In: *Proc. VLDB Endow.* 15.11 (2022), 2921–2928. ISSN: 2150-8097. DOI: 10.14778/3551793.3551841. URL: https://doi.org/10.14778/3551793.3551841.

[37] Jia Li et al. *Large Language Model-Aware In-Context Learning for Code Generation.* 2023. arXiv: 2310.09748 [cs.SE].

[38] Pantazis Deligiannis et al. *Fixing Rust Compilation Errors using LLMs.* 2023. arXiv: 2308.05177 [cs.SE].

[39] Anisha Agarwal et al. *Copilot Evaluation Harness: Evaluating LLM-Guided Software Programming.* 2024. arXiv: 2402.14261 [cs.SE].

[40] Márk Lajkó, Viktor Csuvik, and László Vidács. “Towards JavaScript program repair with generative pre-trained transformer (GPT-2)”. In: *Proceedings of the Third International Workshop on Automated Program Repair.* APR ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, 61–68. ISBN: 9781450392853. DOI: 10.1145/3524459.3527350. URL: https://doi.org/10.1145/3524459.3527350.

[41] Francisco Ribeiro et al. “GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering.* SLE 2023. Cascais, Portugal: Association for Computing Machinery, 2023, 111–124. ISBN: 9798400703966. DOI: 10.1145/3623476.3623522. URL: https://doi.org/10.1145/3623476.3623522.

[42] Hammond Pearce et al. “Examining Zero-Shot Vulnerability Repair with Large Language Models”. In: *2023 IEEE Symposium on Security and Privacy (SP).* 2023, pp. 2339–2356. DOI: 10.1109/SP46215.2023.10179324.

[43] Márk Lajkó et al. “Fine-tuning gpt-2 to patch programs, is it worth it?” In: *International Conference on Computational Science and Its Applications.* Springer. 2022, pp. 79–91.

[44] Ryosuke Ishizue et al. “Improved Program Repair Methods using Refactoring with GPT Models”. In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1.* SIGCSE 2024. Port-

land, OR, USA: Association for Computing Machinery, 2024, 569–575. ISBN: 9798400704239. DOI: 10.1145/3626252.3630875. URL: https://doi.org/10.1145/3626252.3630875.

[45] Matthew Jin et al. "InferFix: End-to-End Program Repair with LLMs". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, 1646–1656. ISBN: 9798400703270. DOI: 10.1145/3611643.3613892. URL: https://doi.org/10.1145/3611643.3613892.

[46] Yuan Huang et al. *Generative Software Engineering.* 2024. arXiv: 2403.02583 [cs.SE].

[47] Mohammed Latif Siddiq, Beatrice Casey, and Joanna C. S. Santos. *A Lightweight Framework for High-Quality Code Generation.* 2023. arXiv: 2307.08220 [cs.SE].

[48] Claudio Spiess et al. *Calibration and Correctness of Language Models for Code.* 2024. arXiv: 2402.02047 [cs.SE].

[49] Michele Tufano et al. *Unit Test Case Generation with Transformers and Focal Context.* 2021. arXiv: 2009.05617 [cs.SE].

[50] Qiuhan Gu. "LLM-Based Code Generation Method for Golang Compiler Testing". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, 2201–2203. ISBN: 9798400703270. DOI: 10.1145/3611643.3617850. URL: https://doi.org/10.1145/3611643.3617850.

[51] Vitor Guilherme and Auri Vincenzi. "An initial investigation of ChatGPT unit test generation capability". In: *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing.* SAST '23. Campo Grande, MS, Brazil: Association for Computing Machinery, 2023, 15–24. ISBN: 9798400716294. DOI: 10.1145/3624032.3624035. URL: https://doi.org/10.1145/3624032.3624035.

[52] Yutian Tang et al. "ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation". In: *IEEE Transactions on Software Engineering* (2024), pp. 1–19. DOI: 10.1109/TSE.2024.3382365.

[53] Vincent Li and Nick Doiron. *Prompting Code Interpreter to Write Better Unit Tests on Quixbugs Functions.* 2023. arXiv: 2310.00483 [cs.SE].

[54] Harrison Chase. *LangChain.* Oct. 2022. URL: https://github.com/langchain-ai/langchain.

[55] *LocalGPT.* https://github.com/PromtEngineer/localGPT.

[56] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.* 2021. arXiv: 2005.11401 [cs.CL].

[57] Hongjin Su et al. *One Embedder, Any Task: Instruction-Finetuned Text Embeddings.* 2023. arXiv: 2212.09741 [cs.CL].

[58] Jack Choquette et al. "NVIDIA A100 Tensor Core GPU: Performance and Innovation". In: *IEEE Micro* 41.2 (2021), pp. 29–35. DOI: 10.1109/MM.2021.3061394.

[59] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing.* 2020. arXiv: 1910.03771 [cs.CL].

[60] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[61] Albert Q. Jiang et al. *Mistral 7B.* 2023. arXiv: 2310.06825 [cs.CL].

[62] Lewis Tunstall et al. *Zephyr: Direct Distillation of LM Alignment.* 2023. arXiv: 2310.16944 [cs.LG].

[63] *crypto-algorithms.* https://github.com/B-Con/crypto-algorithms.

[64] RV Baev et al. "Preventing Vulnerabilities Caused by Optimization of Code with Undefined Behavior". In: *Programming and Computer Software* 48.7 (2022), pp. 445–454.

[65] Chris Hathhorn and Grigore Rosu. "Dealing With C's Original Sin". In: *IEEE Software* 36.5 (2019), pp. 24–28. DOI: 10.1109/MS.2019.2921226.

[66] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. "Defining the undefinedness of C". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, 336–345. ISBN: 9781450334686. DOI: 10.1145/2737924.2737979. URL: https://doi.org/10.1145/2737924.2737979.

[67] Xi Wang et al. "A Differential Approach to Undefined Behavior Detection". In: *ACM Trans. Comput. Syst.* 33.1 (2015). ISSN: 0734-2071. DOI: 10.1145/2699678. URL: https://doi.org/10.1145/2699678.

[68] Zefan Shen. "The Impact of Undefined Behavior on Compiler Optimization". In: *Proceedings of the 2021 European Symposium on Software Engineering.* ESSE '21. Larissa, Greece: Association for Computing Machinery, 2022, 45–50. ISBN: 9781450385060. DOI: 10.1145/3501774.3501781. URL: https://doi.org/10.1145/3501774.3501781.

[69] Wentao Li, Jianhua Sun, and Hao Chen. "Detecting Undefined Behaviors in CUDA C". In: *IEEE Access* 7 (2019), pp. 182559–182572. DOI: 10.1109/ACCESS.2019.2954143.

[70] Shaohua Li and Zhendong Su. "Finding Unstable Code via Compiler-Driven Differential Testing". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3.* ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, 238–251. ISBN: 9781450399180. DOI: 10.1145/3582016.3582053. URL: https://doi.org/10.1145/3582016.3582053.

[71] Juneyoung Lee et al. "Taming undefined behavior in LLVM". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, 633–647. ISBN: 9781450349888. DOI: 10.1145/3062341.3062343. URL: https://doi.org/10.1145/3062341.3062343.

[72] Manjeet Dahiya and Sorav Bansal. "Modeling undefined behaviour semantics for checking equivalence across compiler optimizations". In: *Hardware and Software: Verification and Testing: 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings 13.* Springer. 2017, pp. 19–34.

[73] Xi Wang et al. "Towards optimization-safe systems: analyzing the impact of undefined behavior". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, 260–275. ISBN: 9781450323888. DOI: 10.1145/2517349.2522728. URL: https://doi.org/10.1145/2517349.2522728.

[74] Vijay D'Silva, Mathias Payer, and Dawn Song. "The Correctness-Security Gap in Compiler Optimization". In: *2015 IEEE Security and Privacy Workshops.* 2015, pp. 73–87. DOI: 10.1109/SPW.2015.33.

[75] Michael J. Hohnka et al. "Evaluation of Compiler-Induced Vulnerabilities". In: *Journal of Aerospace Information Systems* 16.10 (2019), pp. 409–426. DOI: 10.2514/1.I010699. eprint: https://doi.org/10.2514/1.I010699. URL: https://doi.org/10.2514/1.I010699.

[76] Alexey R Nurmukhametov et al. "Application of compiler transformations against software vulnerabilities exploitation". In: *Programming and Computer Software* 41 (2015), pp. 231–236. DOI: 10.1134/S0361768815040052.

[77] *binocle*. https://github.com/sharkdp/binocle.

[78] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *7th Python in Science Conference SciPy'08.* Aug. 2008, pp. 11–15. URL: https://www.osti.gov/biblio/960616.

[79] Thomas A Caswell et al. *matplotlib/matplotlib.* Version v3.4.3. DOI: 10.5281/zenodo.5194481.

[80] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (June 2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55. URL: https://doi.org/10.1109/MCSE.2007.55.

[81] Gustavo Sandoval et al. *Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants.* 2023. arXiv: 2208.09727 [cs.CR].

[82] Hammond Pearce et al. *Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions.* 2021. arXiv: 2108.09293 [cs.CR].

[83] Manish Bhatt et al. *Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models.* 2023. arXiv: 2312.04724 [cs.CR].

[84] Fangzhou Wu, Xiaogeng Liu, and Chaowei Xiao. *DeceptPrompt: Exploiting LLM-driven Code Generation via Adversarial Natural Language Instructions.* 2023. arXiv: 2312.04730 [cs.CR].

[85] Kenan Begovic, Abdulaziz Al-Ali, and Qutaibah Malluhi. "Cryptographic ransomware encryption detection: Survey". In: *Computers & Security* 132 (2023), p. 103349.

[86] Dongpeng Xu, Jiang Ming, and Dinghao Wu. "Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping". In: *2017 IEEE Symposium on Security and Privacy (SP).* 2017, pp. 921–937. DOI: 10.1109/SP.2017.56.

[87] Li Jia et al. "A Neural Network-Based Approach for Cryptographic Function Detection in Malware". In: *IEEE Access* 8 (2020), pp. 23506–23521. DOI: 10.1109/ACCESS.2020.2966860.

[88] Felix Matenaar et al. "CIS: The Crypto Intelligence System for automatic detection and localization of cryptographic functions in current malware". In: *2012 7th International Conference on Malicious and Unwanted Software.* 2012, pp. 46–53. DOI: 10.1109/MALWARE.2012.6461007.

[89] Damjan Buhov et al. "Discovering Cryptographic Algorithms in Binary Code Through Loop Enumeration". In: *2017 International Conference on Software Security and Assurance (ICSSA).* 2017, pp. 80–86. DOI: 10.1109/ICSSA.2017.22.

[90] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. "Aligot: cryptographic function identification in obfuscated binary programs". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, 169–182. ISBN: 9781450316514. DOI: 10.1145/2382196.2382217. URL: https://doi.org/10.1145/2382196.2382217.

[91] Hassan Asghar et al. *SoK: Use of Cryptography in Malware Obfuscation*. Cryptology ePrint Archive, Paper 2022/1699. https://eprint.iacr.org/2022/1699. 2022. URL: https://eprint.iacr.org/2022/1699.

[92] Felix Leder, Peter Martini, and Andre Wichmann. "Finding and extracting crypto routines from malware". In: *2009 IEEE 28th International Performance Computing and Communications Conference*. 2009, pp. 394–401. DOI: 10.1109/PCCC.2009.5403858.

[93] Yuanhang Zhou et al. "CLFuzz: Vulnerability Detection of Cryptographic Algorithm Implementation via Semantic-aware Fuzzing". In: *ACM Trans. Softw. Eng. Methodol.* 33.2 (2023). ISSN: 1049-331X. DOI: 10.1145/3628160. URL: https://doi.org/10.1145/3628160.

[94] Hoyong Jin, Dohyeon An, and Taekyoung Kwon. "Differential Testing of Cryptographic Libraries with Hybrid Fuzzing". In: *International Conference on Information Security and Cryptology*. Springer. 2022, pp. 124–144.

[95] Max Ammann, Lucca Hirschi, and Steve Kremer. "Dy fuzzing: formal Dolev-Yao models meet cryptographic protocol fuzz testing". In: *45th IEEE Symposium on Security and Privacy*. 2024.

[96] Chengrun Yang et al. *Large Language Models as Optimizers*. 2023. arXiv: 2309.03409 [cs.LG].

[97] Omar Khattab et al. *DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines*. 2023. arXiv: 2310.03714 [cs.CL].

[98] Yuping Li et al. "Experimental Study of Fuzzy Hashing in Malware Clustering Analysis". In: *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*. Washington, D.C.: USENIX Association, Aug. 2015. URL: https://www.usenix.org/conference/cset15/workshop-program/presentation/li.

[99] Nitin Naik et al. "Fuzzy-Import Hashing: A Malware Analysis Approach". In: *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. 2020, pp. 1–8. DOI: 10.1109/FUZZ48607.2020.9177636.

A Hash Function Test Vectors

The four test vectors that are tested against:

- abc
- abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq
- 1,000,000 repeats of the character a
- 55555555555555555555555555555555555555555555555555555555

The correct SHA-1 hash digests, in hexadecimal, of these four test vectors is the following:

- a9993e364706816aba3e25717850c26c9cd0d89d
- 84983e441c3bd26ebaae4aa1f95129e5e54670f1
- 34aa973cd4c4daa4f61eeb2bdbad27316534016f
- 04575f6b701b0333133f720bc5c1353844075b57

These are the same test vectors defined in the original SHA-1 standard [9], but also with an additional test vector that we manually added to test against. The additional test vector is motivated by having a test vector with a character length equal to the second test vector to specifically check if some of the instances where the hash outputs were correct for some test vectors but not other were due to input character length similarities (specifically having the same character length as the second test vector).

B Source Code Reference Function Implementations for SHA-1

Code Listing 9 shows the reference source code implementation for the 4 functions that implement the SHA-1 algorithm. This implementation is based on the original NIST implementation, defined in ref. [9]. Note that this is from ref. [63], and the comments from the original were removed for these versions which were used in the prompts to the GPT models. Code Listing 11 defines the fixed library imports and the macro that are concatenated with the re-written source code functions in order to generate the complete source code.

Code listing 12 is adapted from [63], and includes one additional test vector that we manually added that has the same character length as one of the original test vectors. Importantly, the C code implementations passing these tests is an indication that the underlying source code is very likely a correct implementation of SHA-1. However, it is not certain that the implementation is correct – and in particular many more tests vectors would need to be applied to the source code in order to better verify its correctness.

Of these source codes, only the individual functions in Code Listing 9 are re-written and evaluated using the GPT models, since these contain the SHA-1 algorithmic syntax.

```c
#include <stdlib.h>
#include <memory.h>
#include "sha1.h"

#define ROTLEFT(a, b) ((a << b) | (a >> (32 - b)))

void sha1_transform(SHA1_CTX *ctx, const BYTE data[]){
        WORD a, b, c, d, e, i, j, t, m[80];
        for (i = 0, j = 0; i < 16; ++i, j += 4)
                m[i] = (data[j] << 24) + (data[j + 1] << 16) + (data[j + 2] << 8) + (data[j + 3]);
        for ( ; i < 80; ++i) {
                m[i] = (m[i - 3] ^ m[i - 8] ^ m[i - 14] ^ m[i - 16]);
                m[i] = (m[i] << 1) | (m[i] >> 31);
        }
        a = ctx->state[0];
        b = ctx->state[1];
        c = ctx->state[2];
        d = ctx->state[3];
        e = ctx->state[4];
        for (i = 0; i < 20; ++i) {
                t = ROTLEFT(a, 5) + ((b & c) ^ (~b & d)) + e + ctx->k[0] + m[i];
                e = d;
                d = c;
                c = ROTLEFT(b, 30);
                b = a;
                a = t;
        }
        for ( ; i < 40; ++i) {
                t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + ctx->k[1] + m[i];
                e = d;
                d = c;
                c = ROTLEFT(b, 30);
                b = a;
                a = t;
        }
        for ( ; i < 60; ++i) {
                t = ROTLEFT(a, 5) + ((b & c) ^ (b & d) ^ (c & d))  + e + ctx->k[2] + m[i];
                e = d;
                d = c;
                c = ROTLEFT(b, 30);
                b = a;
                a = t;
        }
        for ( ; i < 80; ++i) {
                t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + ctx->k[3] + m[i];
                e = d;
                d = c;
                c = ROTLEFT(b, 30);
                b = a;
                a = t;
        }
        ctx->state[0] += a;
        ctx->state[1] += b;
        ctx->state[2] += c;
        ctx->state[3] += d;
        ctx->state[4] += e;
}
void sha1_init(SHA1_CTX *ctx){
        ctx->datalen = 0;
        ctx->bitlen = 0;
        ctx->state[0] = 0x67452301;
        ctx->state[1] = 0xEFCDAB89;
        ctx->state[2] = 0x98BADCFE;
        ctx->state[3] = 0x10325476;
        ctx->state[4] = 0xc3d2e1f0;
        ctx->k[0] = 0x5a827999;
        ctx->k[1] = 0x6ed9eba1;
        ctx->k[2] = 0x8f1bbcdc;
        ctx->k[3] = 0xca62c1d6;
}
void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len){
        size_t i;
        for (i = 0; i < len; ++i) {
                ctx->data[ctx->datalen] = data[i];
                ctx->datalen++;
                if (ctx->datalen == 64) {
                        sha1_transform(ctx, ctx->data);
                        ctx->bitlen += 512;
                        ctx->datalen = 0;
                }
        }
}
void sha1_final(SHA1_CTX *ctx, BYTE hash[]){
        WORD i;
        i = ctx->datalen;
        if (ctx->datalen < 56) {
                ctx->data[i++] = 0x80;
                while (i < 56)
                        ctx->data[i++] = 0x00;
        }
        else {
                ctx->data[i++] = 0x80;
                while (i < 64)
                        ctx->data[i++] = 0x00;
                sha1_transform(ctx, ctx->data);
                memset(ctx->data, 0, 56);
        }
        ctx->bitlen += ctx->datalen * 8;
        ctx->data[63] = ctx->bitlen;
        ctx->data[62] = ctx->bitlen >> 8;
        ctx->data[61] = ctx->bitlen >> 16;
        ctx->data[60] = ctx->bitlen >> 24;
        ctx->data[59] = ctx->bitlen >> 32;
        ctx->data[58] = ctx->bitlen >> 40;
        ctx->data[57] = ctx->bitlen >> 48;
        ctx->data[56] = ctx->bitlen >> 56;
        sha1_transform(ctx, ctx->data);
        for (i = 0; i < 4; ++i) {
                hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
        }
}
```

Listing 9: **The 4 C functions that comprise the SHA-1 algorithm reference implementation. Note that this version is only the C syntax of the functions which are given as part of the GPT prompts, and does not include comments that were in the original version and does not include macros or library importants.**

```
#ifndef sha1_H
#define sha1_H

/*************************** HEADER FILES ***************************/
#include <stddef.h>

/****************************** MACROS ******************************/
#define sha1_BLOCK_SIZE 20              // SHA-1 outputs a 20 byte digest

/**************************** DATA TYPES ****************************/
typedef unsigned char BYTE;             // 8-bit byte
typedef unsigned int  WORD;             // 32-bit word, change to "long" for 16-bit machines

typedef struct {
        BYTE data[64];
        WORD datalen;
        unsigned long long bitlen;
        WORD state[5];
        WORD k[4];
} SHA-1_CTX;

/*********************** FUNCTION DECLARATIONS **********************/
void SHA-1_init(sha1_CTX *ctx);
void SHA-1_update(sha1_CTX *ctx, const BYTE data[], size_t len);
void SHA-1_final(sha1_CTX *ctx, BYTE hash[]);

#endif   // sha1_H
```

Listing 10: **Reference SHA-1 implementation header file**

```
#include <stdlib.h>
#include <memory.h>
#include "sha1.h"

#define ROTLEFT(a, b) ((a << b) | (a >> (32 - b)))
```

Listing 11: **Library imports and macro definition required for the SHA-1 source code function.**

```
#include <stdio.h>
#include <memory.h>
#include <string.h>
#include "sha1.h"

int sha1_test(){
        BYTE text1[] = {"abc"};
        BYTE text2[] = {"abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"};
        BYTE text3[] = {"aaaaaaaaaa"};
        BYTE text4[] = {"55555555555555555555555555555555555555555555555555555555"};
        BYTE hash1[SHA1_BLOCK_SIZE] = {0xa9,0x99,0x3e,0x36,0x47,0x06,0x81,0x6a,0xba,0x3e,0x25,0x71,0x78,0x50,0xc2,0x6c,0x9c,0xd0,0xd8,0x9d};
        BYTE hash2[SHA1_BLOCK_SIZE] = {0x84,0x98,0x3e,0x44,0x1c,0x3b,0xd2,0x6e,0xba,0xae,0x4a,0xa1,0xf9,0x51,0x29,0xe5,0xe5,0x46,0x70,0xf1};
        BYTE hash3[SHA1_BLOCK_SIZE] = {0x34,0xaa,0x97,0x3c,0xd4,0xc4,0xda,0xa4,0xf6,0x1e,0xeb,0x2b,0xdb,0xad,0x27,0x31,0x65,0x34,0x01,0x6f};
        BYTE hash4[SHA1_BLOCK_SIZE] = {0x04, 0x57, 0x5f, 0x6b, 0x70, 0x1b, 0x03, 0x33, 0x13, 0x3f, 0x72, 0x0b, 0xc5, 0xc1, 0x35, 0x38, 0x44, 0x07, 0x5b, 0
              x57};
        BYTE buf[SHA1_BLOCK_SIZE];
        int idx;
        SHA1_CTX ctx;
        int pass = 1;
        sha1_init(&ctx);
        sha1_update(&ctx, text1, strlen(text1));
        sha1_final(&ctx, buf);
        int len = sizeof(buf);
        printf("{'Test1': '");
        for(int i = 0; i < len; i++)
                printf("%02x", buf[i]);
        printf("'}\n");
        pass = pass && !memcmp(hash1, buf, SHA1_BLOCK_SIZE);

        sha1_init(&ctx);
        sha1_update(&ctx, text2, strlen(text2));
        sha1_final(&ctx, buf);
        len = sizeof(buf);
        printf("{'Test2': '");
        for(int i = 0; i < len; i++)
                printf("%02x", buf[i]);
        printf("'}\n");
        pass = pass && !memcmp(hash2, buf, SHA1_BLOCK_SIZE);

        sha1_init(&ctx);
        for (idx = 0; idx < 100000; ++idx)
           sha1_update(&ctx, text3, strlen(text3));
        sha1_final(&ctx, buf);
        len = sizeof(buf);
        printf("{'Test3': '");
        for(int i = 0; i < len; i++)
                printf("%02x", buf[i]);
        printf("'}\n");
        pass = pass && !memcmp(hash3, buf, SHA1_BLOCK_SIZE);

        sha1_init(&ctx);
        sha1_update(&ctx, text4, strlen(text4));
        sha1_final(&ctx, buf);
        len = sizeof(buf);
        printf("{'Test4': '");
        for(int i = 0; i < len; i++)
                printf("%02x", buf[i]);
        printf("'}\n");
        pass = pass && !memcmp(hash4, buf, SHA1_BLOCK_SIZE);

        return(pass);
}

int main(){
    sha1_test();
        return(0);
}
```

Listing 12: **Test code that evaluates the SHA-1 functions, with formatting for easy parsing of the hashes into Python syntax.**

C Checked Markdown Computer Code Language Identifiers

c, c++, Python, ruby, sql, java, go, css, perl, hpp, rust, php, md, markdown, ts, lua, bash, scss, csharp, kotlin, xml, matlab, vbnet, sh, yaml, vscode, arduino, objc, json, js, html, asp, console.log, text, txt, typescript, Makefile, asm, haskell, cpp, log, swift, #!lua, #!c, #!/bin/bash, #!/bin/sh, #!/usr/bin/perl, #! c++, #! sh

Note that, in many cases the markdown language identifier was not correctly identifying the language that was generated. Nevertheless, we do attempt to remove these language identifiers if they exist in order to increase the parse-ability of the GPT output.

D Additional Examples of Fully Correct SHA-1 Function Rewrites

```
void sha1_init(SHA1_CTX *ctx) {
        ctx->datalen = 0;
        ctx->bitlen = 0;
        ctx->state[0] = 0x67452301;
        ctx->state[1] = 0xEFCDAB89;
        ctx->state[2] = 0x98BADCFE;
        ctx->state[3] = 0x10325476;
        ctx->state[4] = 0xc3d2e1f0;
        ctx->k[0] = 0x5a827999;
        ctx->k[1] = 0x6ed9eba1;
        ctx->k[2] = 0x8f1bbcdc;
        ctx->k[3] = 0xca62c1d6;
        ctx->data[ctx->datalen] = 0;
        ctx->bitlen = 0;
        ctx->state[0] = ctx->state[0] | 0x8000000000000000;
        ctx->state[1] = ctx->state[1] | 0x8000000000000000;
        ctx->state[2] = ctx->state[2] | 0x8000000000000000;
        ctx->state[3] = ctx->state[3] | 0x8000000000000000;
        ctx->state[4] = ctx->state[4] | 0x8000000000000000;
        ctx->k[0] = ctx->k[0] | 0x8000000000000000;
        ctx->k[1] = ctx->k[1] | 0x8000000000000000;
        ctx->k[2] = ctx->k[2] | 0x8000000000000000;
        ctx->k[3] = ctx->k[3] | 0x8000000000000000;
}
void sha1_update(SHA1_CTX *ctx, const BYTE *data, size_t len) {
    size_t i;
    BYTE *new_data = ctx->data;
    size_t new_datalen = ctx->datalen;
    size_t new_bitlen = ctx->bitlen;
    unsigned long long *new_state = ctx->state;
    unsigned long long *new_k = ctx->k;
    for (i = 0; i < len; ++i) {
        new_data[new_datalen] = data[i];
        new_datalen++;
        if (new_datalen == 64) {
            sha1_transform(ctx, new_data);
            new_bitlen += 512;
            new_datalen = 0;
        }
    }
    ctx->datalen = new_datalen;
    ctx->bitlen = new_bitlen;
}
void sha1_transform(SHA1_CTX *ctx, const BYTE data[]){
    WORD a, b, c, d, e, i, j, t, m[80];
    for (i = 0, j = 0; i < 16; ++i, j += 4)
        m[i] = (data[j] << 24) + (data[j + 1] << 16) + (data[j + 2] << 8) + (data[j + 3]);
    for (; i < 80; ++i) {
        m[i] = (m[i - 3] ^ m[i - 8] ^ m[i - 14] ^ m[i - 16]);
        m[i] = (m[i] << 1) | (m[i] >> 31);
    }
    WORD k0 = ctx->k[0], k1 = ctx->k[1], k2 = ctx->k[2], k3 = ctx->k[3];
    WORD a0 = ctx->state[0], b0 = ctx->state[1], c0 = ctx->state[2], d0 = ctx->state[3], e0 = ctx->state[4];
    for (i = 0; i < 20; ++i) {
        t = ROTLEFT(a0, 5) + ((b0 & c0) ^ (~b0 & d0)) + e0 + k0 + m[i];
        e0 = d0;
        d0 = c0;
        c0 = ROTLEFT(b0, 30);
        b0 = a0;
        a0 = t;
    }
    for (; i < 40; ++i) {
        t = ROTLEFT(a0, 5) + (b0 ^ c0 ^ d0) + e0 + k1 + m[i];
        e0 = d0;
        d0 = c0;
        c0 = ROTLEFT(b0, 30);
        b0 = a0;
        a0 = t;
    }
    for (; i < 60; ++i) {
        t = ROTLEFT(a0, 5) + ((b0 & c0) ^ (b0 & d0) ^ (c0 & d0)) + e0 + k2 + m[i];
        e0 = d0;
        d0 = c0;
        c0 = ROTLEFT(b0, 30);
        b0 = a0;
        a0 = t;
    }
    for (; i < 80; ++i) {
        t = ROTLEFT(a0, 5) + (b0 ^ c0 ^ d0) + e0 + k3 + m[i];
        e0 = d0;
        d0 = c0;
        c0 = ROTLEFT(b0, 30);
        b0 = a0;
        a0 = t;
    }
    ctx->state[0] += a0;
    ctx->state[1] += b0;
    ctx->state[2] += c0;
    ctx->state[3] += d0;
    ctx->state[4] += e0;
}

void sha1_final(SHA1_CTX *ctx, BYTE hash[]){
        WORD i;
        i = ctx->datalen;
        if (ctx->datalen < 56) {
                ctx->data[i++] = 0x80;
                while (i < 56)
                        ctx->data[i++] = 0x00;
        } else {
                ctx->data[i++] = 0x80;
                while (i < 64)
                        ctx->data[i++] = 0x00;
                sha1_transform(ctx, ctx->data);
                memset(ctx->data, 0, 56);
        }
        ctx->bitlen += ctx->datalen * 8;
        ctx->data[62] = ctx->bitlen >> 8;
        ctx->data[63] = ctx->bitlen & 0xff;
        ctx->data[61] = ctx->bitlen >> 16;
        ctx->data[60] = ctx->bitlen & 0xff00;
        ctx->data[59] = ctx->bitlen >> 24;
        ctx->data[58] = ctx->bitlen & 0xff0000;
        ctx->data[57] = ctx->bitlen >> 32;
        ctx->data[56] = ctx->bitlen & 0xff000000;
        sha1_transform(ctx, ctx->data);
        for (i = 0; i < 4; ++i) {
                hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
        }
}
```

Listing 13: **Fully correct SHA-1 codebase GPT model re-write example Number 2. Each of these four functions were manually selected from the correct and unique re-writes of that function across all of the GPT model outputs.**

```
void sha1_init(SHA1_CTX *ctx){
        ctx->datalen = 0;
        ctx->bitlen = 0;
        ctx->state[0] = 0x67452301;
        ctx->state[1] = 0xEFCDAB89;
        ctx->state[2] = 0x98BADCFE;
        ctx->state[3] = 0x10325476;
        ctx->state[4] = 0xc3d2e1f0;
        ctx->k[0] = 0x5a827999;
        ctx->k[1] = 0x6ed9eba1;
        ctx->k[2] = 0x8f1bbcdc;
        ctx->k[3] = 0xca62c1d6;
        ctx->data[0] = 0x67452301;
        ctx->data[1] = 0xEFCDAB89;
        ctx->data[2] = 0x98BADCFE;
        ctx->data[3] = 0x10325476;
        ctx->data[4] = 0xc3d2e1f0;
        ctx->data[5] = 0x5a827999;
        ctx->data[6] = 0x6ed9eba1;
        ctx->data[7] = 0x8f1bbcdc;
        ctx->data[8] = 0xca62c1d6;
}
void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len) {
        size_t i = 0;
        switch (len) {
        case 1:
                ctx->data[ctx->datalen] = data[0];
                ctx->datalen++;
                break;
        case 2:
                ctx->data[ctx->datalen] = data[0];
                ctx->data[ctx->datalen + 1] = data[1];
                ctx->datalen += 2;
                break;
        case 3:


                ctx->data[ctx->datalen] = data[0];
                ctx->data[ctx->datalen + 1] = data[1];
                ctx->data[ctx->datalen + 2] = data[2];
                ctx->datalen += 3;
                break;
        default:
                while (i < len) {
                        ctx->data[ctx->datalen] = data[i];
                        ctx->datalen++;
                        if (ctx->datalen == 64) {
                                sha1_transform(ctx, ctx->data);
                                ctx->bitlen += 512;
                                ctx->datalen = 0;
                        }
                        i++;
                }
                break;
        }
}
void sha1_transform(SHA1_CTX *ctx, const BYTE data[]){
    WORD a = ctx->state[0], b = ctx->state[1], c = ctx->state[2], d = ctx->state[3], e = ctx->state[4];
    WORD m[80], t;
    WORD i, j;
    WORD k[4] = {0x5a827999, 0x6ed9eba1, 0x8f1bbcdc, 0xca62c1d6};
    for (i = 0, j = 0; i < 16; ++i, j+=4)
        m[i] = (data[j] << 24) + (data[j+1] << 16) + (data[j+2] << 8) + (data[j+3]);
    for (; i < 80; ++i) {
        m[i] = (m[i-3] ^ m[i-8] ^ m[i-14] ^ m[i-16]);
        m[i] = (m[i] << 1) | (m[i] >> 31);
    }
    for (i = 0; i < 20; ++i) {
        t = ROTLEFT(a, 5) + ((b & c) ^ (~b & d)) + e + k[0] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 40; ++i) {
        t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + k[1] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 60; ++i) {
        t = ROTLEFT(a, 5) + ((b & c) ^ (b & d) ^ (c & d))  + e + k[2] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 80; ++i) {
        t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + k[3] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    ctx->state[0] += a;
    ctx->state[1] += b;
    ctx->state[2] += c;
    ctx->state[3] += d;
    ctx->state[4] += e;
}
void sha1_final(SHA1_CTX *ctx, BYTE hash[]){
    WORD i, j, k, l, m, n;
    i = ctx->datalen;
    if (i < 56) {
        ctx->data[i++] = 0x80;
        while (i < 56)
            ctx->data[i++] = 0x00;
    } else {
        ctx->data[i++] = 0x80;
        while (i < 64)
            ctx->data[i++] = 0x00;
        sha1_transform(ctx, ctx->data);
        memset(ctx->data, 0, 56);
    }
    ctx->bitlen += ctx->datalen * 8;
    ctx->data[63] = ctx->bitlen;
    ctx->data[62] = ctx->bitlen >> 8;
    ctx->data[61] = ctx->bitlen >> 16;
    ctx->data[60] = ctx->bitlen >> 24;
    ctx->data[59] = ctx->bitlen >> 32;
    ctx->data[58] = ctx->bitlen >> 40;
    ctx->data[57] = ctx->bitlen >> 48;
    ctx->data[56] = ctx->bitlen >> 56;
    sha1_transform(ctx, ctx->data);
    for (i = 0; i < 4; ++i) {
        m = ctx->state[0];
        n = ctx->state[1];
        l = ctx->state[2];
        k = ctx->state[3];
        j = ctx->state[4];
        hash[i] = (m >> (24 - i * 8)) & 0xff;
        hash[i + 4] = (n >> (24 - i * 8)) & 0xff;
        hash[i + 8] = (l >> (24 - i * 8)) & 0xff;
        hash[i + 12] = (k >> (24 - i * 8)) & 0xff;
        hash[i + 16] = (j >> (24 - i * 8)) & 0xff;
    }
}
```

Listing 14: **Fully correct SHA-1 codebase GPT model re-write example Number 3. Each of these**

four functions were manually selected from the correct and unique re-writes of that function across all of the GPT model outputs.

E Additional Source Code Re-write Examples that have a Variety of Software Flaws

```
void sha1_final(SHA1_CTX *ctx, BYTE hash[]) // Old
{
        WORD i;
        i = ctx->datalen;   // Old
        if (ctx->datalen < 56) {
                ctx->data[i++] = 0x80;   // Old
                while (i < 56)
                        ctx->data[i++] = 0x00;   // Old
        } // Old
        else {
                ctx->data[i++] = 0x80;   // Old
                while (i < 64)
                        ctx->data[i++] = 0x00;   // Old
                sha1_transform(ctx, ctx->data);   // Old
                memset(ctx->data, 0, 56);   // Old
        }
        ctx->bitlen += ctx->datalen * 8;   // Old
        ctx->data[63] = ctx->bitlen;   // Old
        ctx->data[62] = ctx->bitlen >> 8;   // Old
        ctx->data[61] = ctx->bitlen >> 16;   // Old
        ctx->data[60] = ctx->bitlen >> 24;   // Old
        ctx->data[59] = ctx->bitlen >> 32;   // Old
        ctx->data[58] = ctx->bitlen >> 40;   // Old
        ctx->data[57] = ctx->bitlen >> 48;   // Old
        ctx->data[56] = ctx->bitlen >> 56;   // Old
        sha1_transform(ctx, ctx->data);   // Old
        for (i = 0; i < 4; ++i) {
                hash[i]      = (ctx->state[0] >> ((24 - (4*8) - i*8)) & 0
                        x000000ff);  // New
                hash[i+4]   = (ctx->state[1] >> ((24 - (4*8) - i*8)) & 0
                        x000000ff);  // New
                hash[i+8]   = (ctx->state[2] >> ((24 - (4*8) - i*8)) & 0
                        x000000ff);  // New
                hash[i+12] = (ctx->state[3] >> ((24 - (4*8) - i*8)) & 0
                        x000000ff);  // New
                hash[i+16] = (ctx->state[4] >> ((24 - (4*8) - i*8)) & 0
                        x000000ff);  // New
        }
}
```

```
gcc with optimization level 0:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
34aa973cd4c4daa4f61eeb2bdbad27316534016f
04575f6b701b0333133f720bc5c1353844075b57

gcc with optimization level fast:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
34aa973cd4c4daa4f61eeb2bdbad27316534016f
04575f6b701b0333133f720bc5c1353844075b57

clang with optimization level 0:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
34aa973cd4c4daa4f61eeb2bdbad27316534016f
04575f6b701b0333133f720bc5c1353844075b57

clang with optimization level 1:
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
```

Listing 15: This function re-write is an example where the algorithmic correctness of the compiled binary is dependent on the compiler optimization setting that is used (this is categorized by Metric 5). Specifically, gcc with optimization settings produce compiled binaries that correctly produce SHA-1 hashes. clang with optimization level 0 also produces correct SHA-1 binaries. However, clang at all higher optimization levels produces sequences of all 0's for all input test vectors. This function re-write was generated by Mistral-7B-Instruct-v0.1 with inference temperature 1.0 with prompt number 3. The comments in the source code were generated by the GPT model.

```
void sha1_final(SHA1_CTX *ctx, BYTE hash[])
{
        WORD i;
        ctx->datalen = ctx->datalen;
        if (ctx->datalen < 56) {
                ctx->data[ctx->datalen++] = 0x80;
                while (ctx->datalen < 56)
                        ctx->data[ctx->datalen++] = 0x00;
        }
        else {
                ctx->data[ctx->datalen++] = 0x80;
                while (ctx->datalen < 64)
                        ctx->data[ctx->datalen++] = 0x00;
                sha1_transform(ctx, ctx->data);
                ctx->datalen = 0;
                while (ctx->datalen < 56)
                        ctx->data[ctx->datalen++] = 0x00;
        }
        ctx->bitlen = ctx->datalen * 8;
        ctx->data[63] = ctx->bitlen;
        ctx->data[62] = ctx->bitlen >> 8;
        ctx->data[61] = ctx->bitlen >> 16;
        ctx->data[60] = ctx->bitlen >> 24;
        ctx->data[59] = ctx->bitlen >> 32;
        ctx->data[58] = ctx->bitlen >> 40;
        ctx->data[57] = ctx->bitlen >> 48;
        ctx->data[56] = ctx->bitlen >> 56;
        sha1_transform(ctx, ctx->data);
        for (i = 0; i < 4; ++i) {
                hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
        }
}
```

```
Compiled Binary output:
9c80d0f72a11fe6f3919c20ced200a71ea6a9a93
84983e441c3bd26ebaae4aa1f95129e5e54670f1
a72c016e8f148f5fcbd0481bc16f538d311a71dc
04575f6b701b0333133f720bc5c1353844075b57
```

Listing 16: Example GPT function re-write of sha1_final produced from Mistral-7B-Instruct-v0.1 with inference temperature 0.3 and prompt number 4. The output hash for the second and fourth test vector is correct, but all other hashes are not correct to SHA-1. This function re-write is an instance of the function category found by the Metric 17 definition. The output hashes for the four test vectors are given below the source code function - characters that are red denote the characters are incorrect compared to the correct SHA-1 hash, and characters that are black denote that they are correct for the SHA-1 hash.

```
void sha1_update (SHA1_CTX *ctx, const BYTE data[], size_t len)
{
    size_t i;
    for (i = 0; i < len; ++i)
    {
        ctx->data[ctx->datalen] = data[i];
        ctx->datalen++;
        if (ctx->datalen == 64)
        {
            sha1_transform(ctx, ctx->data);
            ctx->bitlen += 768;
            ctx->datalen = 0;
        }
    }
}

Compiled Binary output:
a9993e364706816aba3e25717850c26c9cd0d89d
84983e441c3bd26ebaae4aa1f95129e5e54670f1
9a325e3503b5afd5b61bc7bd5773a593ef596d58
04575f6b701b0333133f720bc5c1353844075b57
```

Listing 17: Example GPT function re-write of sha1_update produced from Mistral-7B-Instruct-v0.1 with inference temperature 1.0 and prompt number 3. The output hash for the first, second, and fourth test vector is correct, but all other hashes are not correct to SHA-1. This function re-write is an instance of the function category found by the Metric 17 definition, meaning that this function re-write causes the compiled binary to be incorrect for some inputs, but correct for other inputs. The output hashes for the four test vectors are given below the source code function.

```
// This function performs the SHA-1 message update operation
void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len)
{
    // Iterate through the input data
    for (size_t i = 0; i < len; ++i)
    {
        // Append the current byte to the message buffer
        ctx->data[ctx->datalen++] = data[i];
        // Check if we have reached the end of the buffer
        if (ctx->datalen == SHA1_BLOCK_SIZE)
        {
            // Perform the SHA-1 transform on the current buffer
            sha1_transform(ctx, ctx->data);
            // Reset the buffer pointer and update the bit length
            ctx->datalen = 0;
            ctx->bitlen += SHA1_BLOCK_SIZE * 8;
        }
    }
}


Compiled Binary output:
a9993e364706816aba3e25717850c26c9cd0d89d
bcabde9df92f44be489c45545a3736e5b92707d7
7b6e48f05ea3919b76a43fc5ca27fe71634dfb6b
71c2da8aa7eca91d00cbbdcd35a5db51bd03b6a7
```

Listing 19: Example GPT function re-write of sha1_update produced from zephyr-7b-alpha with inference temperature 0.5 and prompt number 1. This function re-write is an instance of the function category found by the Metric 17 definition. The output hashes for the four test vectors are given below the source code function.

```
void sha1_final(SHA1_CTX *ctx, BYTE hash[])
{
        WORD i;
        i = ctx->datalen;
        if (ctx->datalen < 56) {
                ctx->data[i++] = 0x80;
                while (i < 56)
                        ctx->data[i++] = 0x00;
        }
        else {
                ctx->data[i++] = 0x80;
                while (i < 64)
                        ctx->data[i++] = 0x00;
                sha1_transform(ctx, ctx->data);
                memset(ctx->data, 0, 56);
                ctx->data[55] = ctx->bitlen >> 40;
                ctx->data[54] = ctx->bitlen >> 32;
                ctx->data[53] = ctx->bitlen >> 24;
                ctx->data[52] = ctx->bitlen >> 16;
                ctx->data[51] = ctx->bitlen >> 8;
                ctx->data[50] = ctx->bitlen;
                sha1_transform(ctx, ctx->data);
        }
        ctx->bitlen += ctx->datalen * 8;
        ctx->data[63] = ctx->bitlen;
        ctx->data[62] = ctx->bitlen >> 8;
        ctx->data[61] = ctx->bitlen >> 16;
        ctx->data[60] = ctx->bitlen >> 24;
        ctx->data[59] = ctx->bitlen >> 32;
        ctx->data[58] = ctx->bitlen >> 40;
        ctx->data[57] = ctx->bitlen >> 48;
        ctx->data[56] = ctx->bitlen >> 56;
        sha1_transform(ctx, ctx->data);
        for (i = 0; i < 4; ++i) {
                hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
        }
}

Compiled Binary output:
a9993e364706816aba3e25717850c26c9cd0d89d
a99aa108dde88d02c74c55c6e4d2a80138f7e795
34aa973cd4c4daa4f61eeb2bdbad27316534016f
364faad2eb7ec72393d62a4351007d947925801f
```

Listing 18: Example GPT function re-write of sha1_final produced from Mistral-7B-Instruct-v0.1 with inference temperature 0.7 and prompt number 4. The generated hash for the first and third test vector is correct, but all other hashes are not correct to SHA-1. This function re-write is an instance of the function category found by the Metric 17 definition. The output hashes for the four test vectors are given below the source code function.

```
void sha1_transform(SHA1_CTX *ctx, const char *data) {
    WORD a, b, c, d, e, i, j, t, m[80];
    for (i = 0, j = 0; i < 16; ++i, j += 4) {
        m[i] = (data[j] << 24) + (data[j + 1] << 16) + (data[j + 2] << 8) +
               (data[j + 3]);
    }
    for (; i < 80; ++i) {
        m[i] = (m[i - 3] ^ m[i - 8] ^ m[i - 14] ^ m[i - 16]);
        m[i] = (m[i] << 1) | (m[i] >> 31);
    }
    a = ctx->state[0];
    b = ctx->state[1];
    c = ctx->state[2];
    d = ctx->state[3];
    e = ctx->state[4];
    for (i = 0; i < 20; ++i) {
        t = ROTLEFT(a, 5) + ((b & c) ^ (~b & d)) + e + ctx->k[0] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 40; ++i) {
        t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + ctx->k[1] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 60; ++i) {
        t = ROTLEFT(a, 5) + ((b & c) ^ (b & d) ^ (c & d))  + e + ctx->k[2] +
            m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 80; ++i) {
        t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + ctx->k[3] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    ctx->state[0] += a;
    ctx->state[1] += b;
    ctx->state[2] += c;
    ctx->state[3] += d;
    ctx->state[4] += e;
}

Compiled Binary output:
c64d3fcde20c5cd03142171e5ac47a87aa3c8ace
8349bd43e6a2450a29b016db3c6d4caccfcabee0
34aa973cd4c4daa4f61eeb2bdbad27316534016f
0fc5d456fa8444a7847a4eae671daba3c39f2362
```

Listing 20: Example GPT function re-write of sha1_transform produced from Mistral-7B-Instruct-v0.1 with inference temperature 0.7 and prompt number 1. This function re-write is an instance of the function category found by the Metric 17 definition. The output hashes for the four test vectors are given below the source code function.

```
#define MSB(x) (x >> (sizeof(x) * 8 - 1))
#define LSB(x) (x & 0xFF)
#define ABS(x) (x >= 0? x : -x)
#define MAX_MSG_SIZE 64
#define MIN_MSG_SIZE (MAX_MSG_SIZE >> 6)
#define MAGIC_KEYS(i) ctx->K[i] = (ctx->k[(i << 1) + 0] << 24) | (ctx->k[(i << 1) + 1] << 16) | (ctx->k[(i << 1) + 2] << 8) | (ctx->k[(i << 1) + 3] << 0)
#define SIZE_64BYTES (64 * 1 << 3)
#define SIZE_1024BYTES (1024 * 1 << 3)
#define SIZE_16384BYTES (16384 * 1 << 3)
#define SIZE_24576BYTES (24576 * 1 << 3)
#define SIZE_32768BYTES (32768 * 1 << 3)
#define SIZE_65536BYTES (65536 * 1 << 3)
#define SIZE_1048576BYTES (1048576 * 1 << 3)
#define SIZE_161061273808BYTES (161061273808 * 1 << 3)
#define SIZE_268435456BYTES (268435456 * 1 << 3)
#define SIZE_4294967296BYTES (4294967296 * 1 << 3)
#define SIZE_1024BYTES (1024 * 1 << 3)
#define MSG_SIZE (ctx->datalen << 3)
#define MSG_SIZE_1 (ctx->datalen)
#define MSG_SIZE_2 (ctx->datalen >> 1)
#define MSG_SIZE_3 (ctx->datalen >> 2)
#define MSG_SIZE_4 (ctx->datalen >> 3)
#define MSG_SIZE_5 (ctx->datalen >> 4)
#define MSG_SIZE_6 (ctx->datalen >> 5)
#define MSG_SIZE_7 (ctx->datalen >> 6)
#define MSG_SIZE_8 (ctx->datalen >> 7)
#define MSG_SIZE_9 (ctx->datalen >> 8)
#define MSG_SIZE_10 (ctx->datalen >> 9)
#define MSG_SIZE_11 (ctx->datalen >> 10)
#define MSG_SIZE_12 (ctx->datalen >> 11)
#define MSG_SIZE_13 (ctx->datalen >> 12)
#define MSG_SIZE_14 (ctx->datalen >> 13)
#define MSG_SIZE_15 (ctx->datalen >> 14)
```

```
#define MSG_SIZE_16 (ctx->datalen >> 15)
#define MSG_SIZE_17 (ctx->datalen >> 16)
#define MSG_SIZE_18 (ctx->datalen >> 17)
#define MSG_SIZE_19 (ctx->datalen >> 18)
#define MSG_SIZE_20 (ctx->datalen >> 19)
#define MSG_SIZE_21 (ctx->datalen >> 20)
#define MSG_SIZE_22 (ctx->datalen >> 21)
#define MSG_SIZE_23 (ctx->datalen >> 22)
#define MSG_SIZE_24 (ctx->datalen >> 23)
#define MSG_SIZE_25 (ctx->datalen >> 24)
#define MSG_SIZE_26 (ctx->datalen >> 25)
#define MSG_SIZE_27 (ctx->datalen >> 26)
#define MSG_SIZE_28 (ctx->datalen >> 27)
#define MSG_SIZE_29 (ctx->datalen >> 28)
#define MSG_SIZE_30 (ctx->datalen >> 29)
#define MSG_SIZE_31 (ctx->datalen >> 30)
#define MSG_SIZE_32 (ctx->datalen >> 31)
#define MSG_SIZE_33 (ctx->datalen >> 32)

void sha1_final(SHA1_CTX *ctx, BYTE hash[])
{
        WORD i;
        i = ctx->datalen;
        if (ctx->datalen < 56) {
                ctx->data[i++] = 0x80;
                for (; i < 56; ++i) {
                        ctx->data[i++] = 0x00;
                }
                sha1_transform(ctx, ctx->data);
                memset(ctx->data, 0, 56);
        }
        else {
                ctx->data[i++] = 0x80;
                for (; i < 64; ++i) {
                        ctx->data[i++] = 0x00;
                }
                sha1_transform(ctx, ctx->data);
                memset(ctx->data, 0, 56);
        }
        ctx->bitlen += ctx->datalen * 8;
        ctx->data[63] = ctx->bitlen;
         ctx->data[62] = ctx->bitlen >> 8;
         ctx->data[61] = ctx->bitlen >> 16;
         ctx->data[60] = ctx->bitlen >> 24;
         ctx->data[59] = ctx->bitlen >> 32;
         ctx->data[58] = ctx->bitlen >> 40;
         ctx->data[57] = ctx->bitlen >> 48;
         ctx->data[56] = ctx->bitlen >> 56;
         sha1_transform(ctx, ctx->data);
         for (i = 0; i < 4; ++i) {
                 hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000FF;
                 hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000FF;
                 hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000FF;
                 hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000FF;
                 hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000FF;
         }
}

Compiled Binary output:
0351b49320841d5af1321fa3a755edf68dfbb21b
84983e441c3bd26ebaae4aa1f95129e5e54670f1
f5b0fefa55f5c06755c7aded39d840326ccafa51
065c3bf21f3e3f6a8f2faac4f234c09a59d0da64
```

Listing 21: Example GPT function re-write of sha1_update produced from zephyr-7b-alpha with inference temperature 0.8 and prompt number 3. This function re-write is an instance of the function category found by the Metric 17 definition. The output hashes for the four test vectors are given below the source code function.

```
void sha1_init(SHA1_CTX *ctx) {
        goto init_state;
        ctx->datalen = 0;
        ctx->bitlen = 0;
init_state:
        ctx->state[0] = 0x67452301;
        ctx->state[1] = 0xEFCDAB89;
        ctx->state[2] = 0x98BADCFE;
        ctx->state[3] = 0x10325476;
        ctx->state[4] = 0xc3d2e1f0;
        ctx->k[0] = 0x5a827999;
        ctx->k[1] = 0x6ed9eba1;
        ctx->k[2] = 0x8f1bbcdc;
        ctx->k[3] = 0xca62c1d6;
}

Compiled Binary output:
a9993e364706816aba3e25717850c26c9cd0d89d
9dcb22c7c3939c0391c092bdf0ba57db55990c56
71279 9aa5bbacb9672c1360b52f2d4d316af1b2b
59504d8c7fcd0362972949d6eefa4a37e89d3b58
```

Listing 22: Example GPT function re-write of sha1_update produced from Llama-2-70b-chat-hf with inference temperature 0.5 and prompt number 2. This function re-write is an instance of the function category found by the Metric 17 definition. The output hashes for the four test vectors are given below the source code function. Note that this function rewrite uses a goto statement.

```
void sha1_init(SHA1_CTX *ctx){
        ctx->datalen = 0;
        ctx->bitlen = 0;
        ctx->state[0] = 0x67452301;
        ctx->state[1] = 0xEFCDAB89;
        ctx->state[2] = 0x98BADCFE;
        ctx->state[3] = 0x10325476;
        ctx->state[4] = 0xc3d2e1f0;
        ctx->k[0] = 0x5a827999;
        ctx->k[1] = 0x6ed9eba1;
        ctx->k[2] = 0x8f1bbcdc;
        ctx->k[3] = 0xca62c1d6;
        ctx->k[4] = 0x6cc51756;
        ctx->k[5] = 0xca62c1d7;
        ctx->k[6] = 0x6cc51757;
        ctx->k[7] = 0xca62c1d8;
        ctx->k[8] = 0x6cc51758;
        ctx->k[9] = 0xca62c1d9;
        ctx->k[10] = 0x6cc5175a;
        ctx->k[11] = 0xca62c1da;
        ctx->k[12] = 0x6cc5175b;
        ctx->k[13] = 0xca62c1db;
        ctx->k[14] = 0x6cc5175c;
        ctx->k[15] = 0xca62c1dc;}

Compiled Binary output for gcc with optimization level 0:
d213d4de859377e13668643aed3af1e46d2f5ea4
84983e441c3bd26ebaae4aa1f95129e5e54670f1
a24fd54b89b420bb8335de10c50b2981beadf7b0
04575f6b701b0333133f720bc5c1353844075b57
```

Listing 23: Example function re-write of sha1_init where under some compiler optimization settings the compiled binary, when executed, results in a fatal error, but for at least one other optimization setting the compiled binary correctly produces SHA-1 hashes. The compiled binaries with clang with optimization levels 1, s, and z result in fatal Signals.SIGSEGV errors. The output hashes are not correct SHA-1 hashes, and do not change under different optimization levels, when compiled with gcc. When compiled using clang with optimization levels 2, 3, fast resulted in a binary that correctly produces SHA-1 hashes for all 4 test vectors.

```
void sha1_final(SHA1_CTX *ctx, BYTE hash[]){
        WORD i;
        WORD* data = malloc(ctx->datalen);
        if (ctx->datalen < 56){
                ctx->data[i] += 128;
                /* pad each byte with 00 bytes until 56 bytes are reached */
                for (i = 1; i < 56; ++i){
                        ctx->data[i] = 0;
                }
        }
        else{
                ctx->data[i] += 128;
                /* pad each byte with 00 bytes until 64 bytes are reached */
                for (i = 1; i < 64; ++i)
                {
                        ctx->data[i] = 0;
                }
                /* store the context data into the allocator pointer */
                data = (WORD*)ctx->data;
                /* transform the data */
                sha1_transform(&ctx->state, data);
                /* reset the context data to 0 */
                memset(ctx->data, 0, ctx->datalen * sizeof(WORD));
                /* store the hash into the array */
                for (i = 0; i < 16; ++i){
                        hash[i] = (ctx->state[0] >> (24 - (i % 4) * 8)) & 0
                            x000000ff;
                }
        }
        free(data);
}
```

Listing 25: Incorrect SHA-1 function implementation of sha1_final which causes a detected memory leak using Valgrind (for a binary compiled using optimization level 0 for either gcc or clang or both). The comments produced by the GPT model are left in the output. In this case, the function re-write resulted in a fatal error (Signals.SIGABRT) for all compiler optimization settings. This example was produced by Mistral-7B-Instruct-v0.1 using inference temperature of 1.0 and prompt number 7.

```
void sha1_transform(SHA1_CTX *ctx, const unsigned char *data) {
    unsigned int a = ctx->k[0], b = ctx->k[1], c = ctx->k[2], d = ctx->k[3];
    unsigned char *m = (unsigned char *) malloc(80 * sizeof(unsigned char));
    for (int i = 0; i < 16; ++i) {
        m[i] = data[i * 4];
        m[i + 16] = data[i * 4 + 1];
        m[i + 16] = data[i * 4 + 2];
        m[i + 16] = data[i * 4 + 3];
    }
    for (int i = 16; i < 80; ++i) {
        m[i] = (m[i - 3] ^ m[i - 4] ^ m[i - 5]) ^ (m[i - 12] ^ m[i - 13] ^ m
                [i - 14]) ^ (m[i - 17] ^ m[i - 18] ^ m[i - 19]);
        m[i] <<= 1;
        m[i] &= 0xffffffff;
        m[i] |= (m[i] >> 31);
    }
    ctx->k[0] += a;
    ctx->k[1] += b;
    ctx->k[2] += c;
    ctx->k[3] += d;
}

Compiled binary output:
67452301efcdab8998badcfe10325476c3d2e1f0
67452301efcdab8998badcfe10325476c3d2e1f0
67452301efcdab8998badcfe10325476c3d2e1f0
67452301efcdab8998badcfe10325476c3d2e1f0
```

Listing 24: Incorrect SHA-1 function implementation of sha1_transform which causes a detected memory leak using Valgrind (for a binary compiled using optimization level 0 for either gcc or clang or both). In this case the compiled binary output does not change for different test vectors. The output does not change in response to different compiler optimization settings. This example was produced by Mistral-7B-Instruct-v0.1 using inference temperature of 1.0 and prompt number 9.

```
void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len){
    size_t k;
    BYTE *temp_data = malloc(len);
    BYTE *end_of_data = temp_data + len;
    BYTE *data_pointer = ctx->data;
    BYTE *datalen_pointer = &ctx->datalen;
    unsigned long long bitlen_register = ctx->bitlen;
    unsigned long long bitlen_pointer = &ctx->bitlen;
    unsigned int *state_pointer = &ctx->state[0];
    unsigned int *k_pointer = &ctx->k[0];
    for (k = 0; k < len; ++k) {
        temp_data[k] = data[k];
    }
    for (k = 0; k < len; ++k) {
        *data_pointer = temp_data[k];
        +++*datalen_pointer;
        if (*datalen_pointer == 64) {
            sha1_transform(ctx, temp_data);
            bitlen_pointer += 512;
            *datalen_pointer = 0;
        }
        data_pointer += 1;
        temp_data += 1;
    }
    free(temp_data);
}
```

Listing 26: Incorrect SHA-1 function implementation of sha1_update which has a detected memory leak using Valgrind (for a binary compiled using optimization level 0 for either gcc or clang or both). In this case, the function re-write resulted in a fatal error (Signals.SIGABRT) for all compiler optimization settings. This example function re-write was produced by zephyr-7b-alpha using inference temperature of 0.8 and prompt number 3.

```
void sha1_transform(SHA1_CTX *ctx, const BYTE data[])
{
    WORD a, b, c, d, e, i, j, t, m[80];
    for (i = 0, j = 0; i < 16; ++i, j += 4)
        m[i] = (data[j] << 24) | (data[j + 1] << 16) | (data[j + 2] << 8) | (data[j + 3]);
    for (; i < 80; ++i) {
        m[i] = (m[i - 3] ^ m[i - 8] ^ m[i - 14] ^ m[i - 16]);
        m[i] = m[i] >> 1 | m[i] << 31;
    }
    a = ctx->state[0];
    b = ctx->state[1];
    c = ctx->state[2];
    d = ctx->state[3];
    e = ctx->state[4];
    for (i = 0; i < 20; ++i) {
        t = ROTLEFT(a, 5) ^ ((b & c) ^ (~b & d)) + e + ctx->k[0] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 40; ++i) {
        t = ROTLEFT(a, 5) ^ (b ^ c ^ d) + e + ctx->k[1] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 60; ++i) {
        t = ROTLEFT(a, 5) ^ ((b & c) ^ (b & d) ^ (c & d)) + e + ctx->k[2] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    for (; i < 80; ++i) {
        t = ROTLEFT(a, 5) ^ (b ^ c ^ d) + e + ctx->k[3] + m[i];
        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = t;
    }
    ctx->state[0] += a;
    ctx->state[1] += b;
    ctx->state[2] += c;
    ctx->state[3] += d;
    ctx->state[4] += e;
}

Compiled Binary output:
44c16e0596a9e168efe4e21c255383833f542f71
8e57b34b97c19a64656bbd1bfd3258e15bcbc181
c7f40a8561316b63e6d8184906611e33203c8488
601675c0e3b31a9489289831af8a5a4743f90dfb
```

Listing 27: Example of a function re-write where the generated hashes have some of the basic requirements of a valid hash function, and have the same hexadecimal character length as SHA-1 hashes (40 characters). This code

did not produce different output based on different compiler optimization settings. This is an example of a case found by Metric 39. The hash output for the 4 test vectors is given below the source code. This function re-write was generated by Mistral-7B-Instruct-v0.1 with inference temperature 0.4 with prompt number 2.

```
void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len)
{
        WORD i;
        for (i = 0; i < len; ++i) {
                ctx->data[ctx->datalen] = data[i];
                ctx->datalen++;
                if (ctx->datalen == 64) {
                        sha1_transform(ctx, ctx->data);
                        ctx->bitlen += 512;
                        ctx->datalen = 0;
                }
        }
        ctx->state[0] ^= ctx->k[0];
        ctx->state[1] ^= ctx->k[1];
        ctx->state[2] ^= ctx->k[2];
        ctx->state[3] ^= ctx->k[3];
        ctx->state[4] ^= ctx->k[4] ^ ctx->bitlen;
        ctx->bitlen += ctx->datalen * 8;
}

Compiled Binary output:
73038396beca31768b71a26bc64f9f5d1ea0b81e
4029913a19bf89bfd628764c745cec4ba38f8ed0
251aaa6cb2ee37df9df4a32429854a34071902e7
3e2d4c8ba2fef255c2d3ba42d7fcd29bc16a8a8d
```

Listing 28: Example of a function re-write where the generated hashes have some of the basic requirements of a valid hash function, and have the same hexadecimal character length as SHA-1 hashes (40 characters). This code did not produce different output based on different compiler optimization settings. This is an example of a case found by Metric 39. The hash output for the 4 test vectors is given below the source code. This function re-write was generated by Mistral-7B-Instruct-v0.1 with inference temperature 0.9 with prompt number 1.

```
void sha1_final(SHA1_CTX *ctx, BYTE hash[])
{
    size_t datalen = ctx->datalen;
    if (datalen < 56) {
        ctx->data[datalen++] = 0x80;
        while (datalen < 56)
            ctx->data[datalen++] = 0x00;
    } else {
        ctx->data[datalen++] = 0x80;
        while (datalen < 64)
            ctx->data[datalen++] = 0x00;
        sha1_transform(ctx, ctx->data);
        memset(ctx->data, 0, 56);
    }
    ctx->bitlen += datalen * 8;
    ctx->data[63] = ctx->bitlen;
    ctx->data[62] = ctx->bitlen >> 8;
    ctx->data[61] = ctx->bitlen >> 16;
    ctx->data[60] = ctx->bitlen >> 24;
    ctx->data[59] = ctx->bitlen >> 32;
    ctx->data[58] = ctx->bitlen >> 40;
    ctx->data[57] = ctx->bitlen >> 48;
    ctx->data[56] = ctx->bitlen >> 56;
    sha1_transform(ctx, ctx->data);
    size_t i = 0;
    while (i < 4) {
        hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
        i++;
    }
}

Compiled Binary output:
9c80d0f72a11fe6f3919c20ced200a71ea6a9a93
b5e6e74a4b308239af8f0390c65cfb1aa2cdb465
14fd67290c0cb4d3a620d176d040de48ffb5d147
1b37789ce58d800733383af03d871f4aff329c447
```

Listing 29: Example of a function re-write where the generated hashes have some of the basic requirements of a valid hash function, and have the same hexadecimal character length as SHA-1 hashes (40 characters). This code did not produce different output based on different compiler optimization settings. This is an example of a case found by Metric 39. The hash output for the 4 test vectors is given below the source code. This function re-write was generated by Llama-2-70b-chat-hf with inference temperature 0.8 with prompt number 7.

```
void sha1_init(SHA1_CTX *ctx) {
  int x = 0, y = 0, z = 0, w = 0;
  ctx->datalen = x;
  ctx->bitlen = y;
  ctx->state[0] = 0x12345678;
  ctx->state[1] = 0x90123456;
  ctx->state[2] = 0x78901234;
  ctx->state[3] = 0x56789012;
  ctx->state[4] = 0x34567890;
  ctx->k[0] = 0x99827530;
  ctx->k[1] = 0x31628974;
  ctx->k[2] = 0x54213478;
  ctx->k[3] = 0x75632148;
}

Compiled Binary output:
988eed2df1d6022c014b63d8c071d890658f648e
43108df3de63bbc044b4ffe4d981043b51cbe8fc
2db8738884d0768ba8e871953caff4a193f1d548
40ae75db48ba6836766d261d0c19519129d3a356
```

Listing 30: Example of a function re-write where the generated hashes have some of the basic requirements of a valid hash function, and have the same hexadecimal character length as SHA-1 hashes (40 characters). This code did not produce different output based on different compiler optimization settings. This is an example of a case found by Metric 39. The hash output for the 4 test vectors is given below the source code. This function re-write was generated by Llama-2-70b-chat-hf with inference temperature 1.0 with prompt number 3.

```
void sha1_transform(SHA1_CTX *ctx, const BYTE data[]){
    // Use a different naming convention for the variables
    WORD w0, w1, w2, w3, w4, i, j, t, m[80];
    // Initialize the hash value
    w0 = 0x67452301;
    w1 = 0xEFCDAB89;
    w2 = 0x98BADCFE;
    w3 = 0x10325476;
    w4 = 0xc3d2e1f0;
    // Process the input data
    for (i = 0, j = 0; i < 16; ++i, j += 4)
        m[i] = (data[j] << 24) + (data[j + 1] << 16) + (data[j + 2] << 8) +
              (data[j + 3]);
    for (; i < 80; ++i) {
        m[i] = (m[i - 3] ^ m[i - 8] ^ m[i - 14] ^ m[i - 16]);
        m[i] = (m[i] << 1) | (m[i] >> 31);
    }
    // Update the hash value
    w0 += m[0];
    w1 += m[1];
    w2 += m[2];
    w3 += m[3];
    w4 += m[4];

    // Perform the rotation and addition operations
    for (i = 0; i < 20; ++i) {
        t = ROTLEFT(w0, 5) + (w1 & w2) ^ (~w1 & w3) + w4 + ctx->k[0] + m[i];
        w4 = w3;
        w3 = w2;
        w2 = w1;
        w1 = w0;
        w0 = t;
    }
    for (; i < 40; ++i) {
        t = ROTLEFT(w0, 5) + (w1 ^ w2 ^ w3) + w4 + ctx->k[1] + m[i];
        w4 = w3;
        w3 = w2;
        w2 = w1;
        w1 = w0;
        w0 = t;
    }
    for (; i < 60; ++i) {
        t = ROTLEFT(w0, 5) + (w1 & w2) ^ (w1 & w3) ^ (w2 & w3) + w4 + ctx->k
              [2] + m[i];
        w4 = w3;
        w3 = w2;
        w2 = w1;
        w1 = w0;
        w0 = t;
    }
    for (; i < 80; ++i) {
        t = ROTLEFT(w0, 5) + (w1 ^ w2 ^ w3) + w4 + ctx->k[3] + m[i];
        w4 = w3;
        w3 = w2;
        w2 = w1;
        w1 = w0;
        w0 = t;
    }
    // Update the context state
    ctx->state[0] += w0;
    ctx->state[1] += w1;
    ctx->state[2] += w2;
    ctx->state[3] += w3;
    ctx->state[4] += w4;
}

Compiled Binary output:
b61026ab10128 7aa83cabd16b9379693f0f8b3ec
ca7e0a10ad1ac98e20c02f7c15e1a47a1b7bf02f
0308edb90fc3b923f874455414388191 4286a132
1e083c2ef69c11b3ad1bd0fa58ad306631b13af4
```

Listing 31: Example of a function re-write where the generated hashes have some of the basic requirements of a valid hash function, and have the same hexadecimal character length as SHA-1 hashes (40 characters). This code did not produce different output based on different compiler optimization settings. This is an example of a case found by Metric 39. The hash output for the 4 test vectors is given below the source code. This function re-write was generated by Llama-2-70b-chat-hf with inference temperature 0.8 with prompt number 8.

```
void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len)
{
    size_t i, j = 0;

    for (i = 0; i < len; ++i) {
        ctx->data[ctx->datalen / 2 % 32] = data[i], ctx->datalen *= 2;
        if (! (ctx->datalen) % 16384) {
            sha1_transform(ctx, ctx->data);
            ctx->bitlen += (64 * 64) * 8;
            ctx->datalen = 0;
        }
    }
}

Compiled Binary output:
c988768035c5cac9cc9f2c42a6678fedb952568c
3514f45d596e8d1d422e6929ed5e5a4b21f609ec
80b00421bf8a0f727206dc0365065c7902f73006
87adef43ed4e855c7cdc6fb1c71e9914fd4ad7a4
```

Listing 32: Example of a function re-write where the generated hashes have some of the basic requirements of a valid hash function, and have the same hexadecimal character length as SHA-1 hashes (40 characters). This code did not produce different output based on different compiler optimization settings. This is an example of a case found by Metric 39. The hash output for the 4 test vectors is given below the source code. This function re-write was generated by zephyr-7b-alpha with inference temperature 1.0 with prompt number 2.

```
void sha1_transform(SHA1_CTX *ctx, const BYTE data[])
{
        static const BYTE k[] = { 0x5a827999, 0x6ed9eba1, 0x8f1bbcdc, 0
              xca62c1d6 };
        WORD w[80];
        for (size_t i = 0, j = 0; i < 16; ++i, j += 4)
                w[i] = (data[j] << 24) + (data[j + 1] << 16) + (data[j + 2]
                      << 8) + (data[j + 3]);
        for (size_t i = 16; i < 80; ++i) {
                w[i] = (w[i - 3] ^ w[i - 8] ^ w[i - 14] ^ w[i - 16]);
                w[i] = (w[i] << 1) | (w[i] >> 31);
        }
        register WORD a, b, c, d, e;
        a = ctx->state[0];
        b = ctx->state[1];
        c = ctx->state[2];
        d = ctx->state[3];
        e = ctx->state[4];
        for (size_t i = 0; i < 20; ++i) {
                register WORD t;
                t = ROTLEFT(a, 5) + ((b & c) ^ (~b & d)) + e + k[0] + w[i];
                e = d;
                d = c;
                c = ROTLEFT(b, 30);
                b = a;
                a = t;
        }
        for (size_t i = 20; i < 40; ++i) {
                register WORD t;
                t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + k[1] + w[i];
                e = d;
                d = c;
                c = ROTLEFT(b, 30);
                b = a;
                a = t;
        }
        for (size_t i = 40; i < 60; ++i) {
                register WORD t;
                t = ROTLEFT(a, 5) + ((b & c) ^ (b & d) ^ (c & d))  + e + k
                      [2] + w[i];
                e = d;
                d = c;
                c = ROTLEFT(b, 30);
                b = a;
                a = t;
        }
        for (size_t i = 60; i < 80; ++i) {
                register WORD t;
                t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + k[3] + w[i];
                e = d;
                d = c;
                c = ROTLEFT(b, 30);
                b = a;
                a = t;
        }
        ctx->state[0] += a;
        ctx->state[1] += b;
        ctx->state[2] += c;
        ctx->state[3] += d;
        ctx->state[4] += e;
}

Compiled Binary output:
d48e2bccb02d6417dfe6f50c44b65960ad31be52
0abffb62cc76ba8e46c2a8cf2aef8b64dc06e981
99c6573c2c7443d0bf3d954ffb2bf361d49c969f
92d60c6f24460ce6eea5efdab1cd515db1e4f426
```

Listing 33: Example of a function re-write where the generated hashes have some of the basic requirements of a valid hash function, and have the same hexadecimal character length as SHA-1 hashes (40 characters). This code did not produce different output based on different compiler optimization settings. This is an example of a case found by Metric 39. The hash output for the 4 test vectors is given below the source code. Function re-write was generated by zephyr-7b-alpha with inference temperature 0.8 with prompt number 3.

```
typedef unsigned int WORD;

void sha1_transform(SHA1_CTX *ctx, const BYTE data[])
{
    int i, j, k, t;
    BYTE m[80];

    for (i = 0, j = 0; i < 16; ++i, j += 4)
    {
        m[i] = data[j];
        m[i+16] = data[j+1];
        m[i+32] = data[j+2];
        m[i+48] = data[j+3];
    }

    for (; i < 80; ++i)
    {
        m[i] = m[i-8] ^ m[i-14] ^ m[i-16];
        m[i] = (m[i] << 1) | (m[i] >> 31);
    }

    t = ctx->state[0];
    t = t + m[0];
    t = t + m[16];
    t = t + m[32];
    t = t + m[48];
    t = t + m[64];
    t = t + m[48];
    t = t + m[64];
    t = t + m[76];
    t = t + m[76];

    ctx->state[0] = t;
    ctx->state[1] = ctx->state[0] & ctx->state[1];
    ctx->state[1] = ctx->state[1] ^ (ctx->state[1] & ctx->state[2]);
    ctx->state[2] = ctx->state[2] ^ (ctx->state[2] & ctx->state[3]);
    ctx->state[3] = ctx->state[3] ^ (ctx->state[3] & ctx->state[1]);
    ctx->state[4] = ctx->state[0] & ctx->state[1];
    ctx->state[5] = ctx->state[2];
    ctx->state[6] = ctx->state[3];
    ctx->state[7] = ctx->state[4];
}

Compiled Binary output:
674527c0674523008888888881032547667452300
674525606745210088888888810325476674521 00
684b2924600000008888888881032547660000000
674527e86745230088888888810325476674523 00
```

Listing 34: Example of a function re-write where the generated hashes have some of the basic requirements of a valid hash function, and have the same hexadecimal character length as SHA-1 hashes (40 characters). This code did not produce different output based on different compiler optimization settings. This is an example of a case found by Metric 39. The hash output for the 4 test vectors is given below the source code. This function re-write was generated by Mistral-7B-Instruct-v0.1 with inference temperature 0.6 with prompt number 4. **Note that the generated hash checksums in this example contain clear repeated sequences of characters, making it not a good hash function.**

```
void sha1_final(SHA1_CTX *ctx, BYTE hash[])
{
    WORD i;
    i = ctx->datalen;
    if (ctx->datalen < 56) {
        ctx->data[i++] = 0x80;
        while (i < 56)
            ctx->data[i++] = 0x00;
    }
    else {
        ctx->data[i++] = 0x80;
        while (i < 64)
            ctx->data[i++] = 0x00;
        sha1_transform(ctx, ctx->data);
        memset(ctx->data, 0, 56);
    }
    ctx->bitlen += ctx->datalen * 8;
    ctx->data[63] = ctx->bitlen;
    ctx->data[62] = ctx->bitlen >> 8;
    ctx->data[61] = ctx->bitlen >> 16;
    ctx->data[60] = ctx->bitlen >> 24;
    ctx->data[59] = ctx->bitlen >> 32;
    ctx->data[58] = ctx->bitlen >> 40;
    ctx->data[57] = ctx->bitlen >> 48;
    ctx->data[56] = ctx->bitlen >> 56;
    sha1_transform(ctx, ctx->data);
    for (WORD j = 0; j < 4; ++j) {
        WORD k = 0;
        for (WORD n = 0; n < 3; ++n) {
            k = (k << 5) | (ctx->state[j] >> (24 - (3 - n) * 8));
            k = (k << 5) | (ctx->state[j + 1] >> (24 - (3 - n) * 8));
            k = (k << 5) | (ctx->state[j + 2] >> (24 - (3 - n) * 8));
            k = (k << 5) | (ctx->state[j + 3] >> (24 - (3 - n) * 8));
            k = (k << 5) | (ctx->state[j + 4] >> (24 - (3 - n) * 8));
        }
        hash[j]      = (k >> (24 - j * 8)) & 0x000000ff;
        hash[j + 4]  = (k >> (24 - j * 8)) & 0x000000ff;
        hash[j + 8]  = (k >> (24 - j * 8)) & 0x000000ff;
        hash[j + 12] = (k >> (24 - j * 8)) & 0x000000ff;
        hash[j + 16] = (k >> (24 - j * 8)) & 0x000000ff;
    }
}

Compiled Binary output:
bbff7e3bbbff7e3bbbff7e3bbbff7e3bbbff7e3b
effffe3beffffe3beffffe3beffffe3beffffe3b
eb6ffe3beb6ffe3beb6ffe3beb6ffe3beb6ffe3b
ffbffe3bffbffe3bffbffe3bffbffe3bffbffe3b
```

Listing 35: Example of a function re-write where the generated hashes have some of the basic requirements of a valid hash function, and have the same hexadecimal character length as SHA-1 hashes (40 characters). This code did not produce different output based on different compiler optimization settings. This is an example of a case found by Metric 39. The hash output for the 4 test vectors is given below the source code. This function re-write was generated by Mistral-7B-Instruct-v0.1 with inference temperature 0.6 with prompt number 4. **Note that the generated hash checksums in this example contain clear repeated sequences of characters, making it not a good hash function.**

```c
void sha1_final(SHA1_CTX *ctx, BYTE hash[]) {
    unsigned int a, b, c, d, e;
    unsigned char i;
    a = ctx->datalen;
    while (a-- > 0) {
        b = ctx->data[i++];
        c = ctx->data[i++];
        d = ctx->data[i++];
        e = ctx->data[i++];
        a = (a + 1) % 64;
        hash[a] = (b << 24) | (c << 16) | (d << 8) | e;
    }
    memset(ctx->data, 0, 56);
    ctx->bitlen += ctx->datalen * 8;
    for (i = 0; i < 4; ++i) {
        hash[i] = (ctx->state[i] >> (24 - i * 8)) & 0x000000ff;
    }
}
```

Listing 36: Incorrect SHA-1 function sha1_update implementation that results in a compiled binary, for all compiler settings, that results in an apparent infinite loop. This source code was generated by Llama-2-70b-chat-hf with an inference temperature of 1.0 and prompt number 2. Example 2 of an infinite loop case.

```c
void sha1_update(SHA1_CTX* ctx, const BYTE data[], size_t len) {
    size_t idx, jdx;
    BYTE* pData = (BYTE*)malloc(len);
    memcpy(pData, data, len);
    for (idx = 0; idx < len; ++idx) {
        ctx->data[ctx->datalen] = pData[idx];
        ctx->datalen++;
        if (ctx->datalen == 64) {
            sha1_transform(ctx, ctx->data);
            ctx->bitlen += 512;
            ctx->datalen = 0;
            free(pData);
            pData = (BYTE*)malloc(len - idx);
            memcpy(pData, data + idx, len - idx);
            jdx = 0;
            for (; jdx < len - idx; ++jdx) {
                ctx->data[ctx->datalen] = pData[jdx];
                ctx->datalen++;
                if (ctx->datalen == 64) {
                    sha1_transform(ctx, ctx->data);
                    ctx->bitlen += 512;
                    ctx->datalen = 0;
                    free(pData);
                    pData = (BYTE*)malloc(len - idx - jdx);
                    memcpy(pData, data + idx + jdx, len - idx - jdx);
                    jdx = 0;
                }
            }
            free(pData);
            break;
        }
    }
    free(pData);
}
```

Listing 37: Incorrect SHA-1 function sha1_update implementation that results in a compiled binary, for all compiler settings, that crashes in a fatal error (specifically a SIGABRT error). This source code was generated by zephyr-7b-alpha with an inference temperature of 0.7 and prompt number 3. This function re-write also had an error detected by Valgrind (Invalid free() / delete / realloc()).

```
void sha1_final(SHA1_CTX *ctx, BYTE hash[]) {
    WORD i, j;
    WORD *data = ctx->data;
    WORD state[5];
    WORD k[4];
    WORD bitlen;
    WORD *m = malloc(80 * sizeof(WORD));
    for (i = 0; i < 5; ++i)
        state[i] = ctx->state[i];
    for (i = 0; i < 4; ++i)
        k[i] = ctx->k[i];
    bitlen = ctx->bitlen;
    for (i = 0; i < ctx->datalen; ++i) {
        data[ctx->datalen] = ctx->data[i];
        ctx->datalen++;
        if (ctx->datalen == 64) {
            for (j = 0; j < 80; ++j) {
                m[j] = (data[j] << 24) | (data[j + 1] << 16) | (data[j + 2]
                      << 8) | (data[j + 3] & 0xff);
                if (j < 16)
                    m[j] ^= (m[j - 3] ^ m[j - 8] ^ m[j - 14] ^ m[j - 16]) <<
                           1;
                m[j] ^= (m[j - 3] ^ m[j - 8] ^ m[j - 14] ^ m[j - 16]) >> 31;
            }
            for (i = 0; i < 20; ++i) {
                WORD a = state[0];
                WORD b = state[1];
                WORD c = state[2];
                WORD d = state[3];
                WORD e = state[4];
                WORD t = (a << 5) | (a >> 31) | ((b & c) ^ (~b & d) | e) | (
                      ctx->k[0] & m[i]);
                e = d;
                d = c;
                c = (b << 30) | (b >> 2);
                b = a;
                a = t;
            }
            for ( ; i < 40; ++i) {
                WORD a = state[0];
                WORD b = state[1];
                WORD c = state[2];
                WORD d = state[3];
                WORD e = state[4];
                WORD t = (a << 5) | (a >> 31) | ((b & c) ^ (b & d) ^ (c & d)
                      ) | (ctx->k[1] & m[i]);
                e = d;
                d = c;
                c = (b << 30) | (b >> 2);
                b = a;
                a = t;
            }
            for ( ; i < 60; ++i) {
                WORD a = state[0];
                WORD b = state[1];
                WORD c = state[2];
                WORD d = state[3];
                WORD e = state[4];
                WORD t = (a << 5) | (a >> 31) | ((b & c) ^ (b ^ c ^ d) | (
                      ctx->k[2] & m[i]));
                e = d;
                d = c;
                c = (b << 30) | (b >> 2);
                b = a;
                a = t;
            }
            for ( ; i < 80; ++i) {
                WORD a = state[0];
                WORD b = state[1];
                WORD c = state[2];
                WORD d = state[3];
                WORD e = state[4];
                WORD t = (a << 5) | (a >> 31) | ((b & c) ^ (b ^ c ^ d) | (
                      ctx->k[3] & m[i]));
                e = d;
                d = c;
                c = (b << 30) | (b >> 2);
                b = a;
                a = t;
            }
            for (i = 0; i < 4; ++i) {
                hash[i]      = (state[0] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 4]  = (state[1] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 8]  = (state[2] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 12] = (state[3] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 16] = (state[4] >> (24 - i * 8)) & 0x000000ff;
            }
            free(m);
            ctx->bitlen += ctx->datalen * 8;
            ctx->data[63] = ctx->bitlen;
            ctx->data[62] = ctx->bitlen >> 8;
            ctx->data[61] = ctx->bitlen >> 16;
            ctx->data[60] = ctx->bitlen >> 24;
            ctx->data[59] = ctx->bitlen >> 32;
            ctx->data[58] = ctx->bitlen >> 40;
            ctx->data[57] = ctx->bitlen >> 48;
            ctx->data[56] = ctx->bitlen >> 56;
            sha1_transform(ctx, ctx->data);
        }
    }
}
```

Listing 38: Incorrect SHA-1 function sha1_final implementation that results in a compiled binary, for all compiler settings, that crashes in a fatal error (specifically a SIGSEGV error). This source code was generated by zephyr-7b-alpha with an inference temperature of 0.2 and prompt number 3.

```
void sha1_init(SHA1_CTX *ctx) {
        ctx->datalen = (0x00000000 | (0x00000001 << 8) | (0x00000010 << 16)
              | (0x00000100 << 24));
        ctx->bitlen = (0x00000000 | (0x00000001 << 8) | (0x00000010 << 16) |
               (0x00000100 << 24));
        ctx->state[0] = (0x67452301 | (0x67452302 << 8) | (0x67452303 << 16)
               | (0x67452304 << 24));
        ctx->state[1] = (0xEFCDAB89 | (0xEFCDAB90 << 8) | (0xEFCDAB91 << 16)
               | (0xEFCDAB92 << 24));
        ctx->state[2] = (0x98BADCFE | (0x98BADCFD << 8) | (0x98BADCFE << 16)
               | (0x98BADCFD << 24));
        ctx->state[3] = (0x10325476 | (0x10325477 << 8) | (0x10325478 << 16)
               | (0x10325479 << 24));
        ctx->state[4] = (0xc3d2e1f0 | (0xc3d2e1f1 << 8) | (0xc3d2e1f2 << 16)
               | (0xc3d2e1f3 << 24));
        ctx->k[0] = (0x5a827999 | (0x5a82799a << 8) | (0x5a82799b << 16) |
              (0x5a82799c << 24));
        ctx->k[1] = (0x6ed9eba1 | (0x6ed9eba2 << 8) | (0x6ed9eba3 << 16) |
              (0x6ed9eba4 << 24));
        ctx->k[2] = (0x8f1bbcdc | (0x8f1bbcdD << 8) | (0x8f1bbcdE << 16) |
              (0x8f1bbcdF << 24));
        ctx->k[3] = (0xca62c1d6 | (0xca62c1d7 << 8) | (0xca62c1d8 << 16) |
              (0xca62c1d9 << 24));
}
```

Listing 39: Incorrect SHA-1 function sha1_init implementation that results in a compiled binary, for all compiler settings, that crashes in a fatal error (specifically a SIGSEGV error). This source code was generated by Llama-2-70b-chat-hf with an inference temperature of 0.9 and prompt number 4.

```
void sha1_transform(SHA1_CTX *ctx, const BYTE data[]) {
    WORD a, b, c, d, e, i, j, t, m[79];
    for (i = 0, j = 0; i < 16; ++i, j += 4)
        m[i] = (data[j] << 24) | (data[j+1] << 16) | (data[j+2] << 8) | (
              data[j+3]);
    for (i = 0; i < 76; ++i) {
        t = ROL_AND(a, b) ^ ROL_XOR(~b, d) + e + ctx->k[0] + m[i];
        e = ROL_XOR(d, c) + m[79 - i];
        d = ROL_XOR(c, b);
        c = ROL_LEFT(b, 30);
        m[i] = t | ((t >> 31) << 5);
        a = t;
    }
    for (i = 0; i < 32; ++i) {
        a += b;
    }
}
```

Listing 40: Incorrect SHA-1 function sha1_transform implementation that is compiler optimization unstable; for some settings, the binary compiles, but for other settings, the binary does not compile. This function, when substituted in the original SHA-1 source code, could only be compiled using clang with an optimization level higher than 0; all gcc settings, and clang with optimization level 0, were not able to compile the binary. This source code was generated by Mistral-7B-Instruct-v0.1 with an inference temperature of 1.0 and prompt number 2.

```
void sha1_final(SHA1_CTX *ctx, BYTE hash[]){
        WORD i;
        i = ctx->datalen;
        if (ctx->datalen < 56) {
                ctx->data[i++] = 0x80;
                while (i < 56)
                        ctx->data[i++] = 0x00;
        }
        else {
                ctx->data[i++] = 0x80;
                while (i < 64)
                        ctx->data[i++] = 0x00;
                sha1_obfuscate(ctx, ctx->data);
                memset(ctx->data, 0, 56);
        }
        ctx->bitlen += ctx->datalen * 8;
        ctx->data[63] = ctx->bitlen;
        ctx->data[62] = ctx->bitlen >> 8;
        ctx->data[61] = ctx->bitlen >> 16;
        ctx->data[60] = ctx->bitlen >> 24;
        ctx->data[59] = ctx->bitlen >> 32;
        ctx->data[58] = ctx->bitlen >> 40;
        ctx->data[57] = ctx->bitlen >> 48;
        ctx->data[56] = ctx->bitlen >> 56;
        sha1_obfuscate(ctx, ctx->data);
        for (i = 0; i < 4; ++i) {
                hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
        }
}

void sha1_obfuscate(SHA1_CTX *ctx, BYTE data[]){
        WORD i;
        for (i = 0; i < 80; ++i) {
                data[i] = (data[i - 3] ^ data[i - 8] ^ data[i - 14] ^ data[i - 16]) & 0xFF;
                data[i] = (data[i] << 1) | (data[i] >> 31);
        }
        ctx->state[0] = data[0];
        ctx->state[1] = data[1];
        ctx->state[2] = data[2];
        ctx->state[3] = data[3];
        ctx->state[4] = data[4];
}
```

Listing 41: Incorrect SHA-1 function sha1_final implementation that is compiler unstable; for some settings, the binary compiles, but for other settings, the binary does not compile. This function, when substituted in the original SHA-1 source code, could be compiled using gcc (using any optimization setting), but could not be compiled using any optimization setting in clang. Note that this implementation includes an additional function called sha1_obfuscate that the GPT model produced, which is not a component of the original SHA-1 source code in any way. This source code was generated by Mistral-7B-Instruct-v0.1 with an inference temperature of 0.5 and prompt number 2.

```
void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len){
    size_t i;
    for (i = 0; i < len; ++i) {
        ctx->data[ctx->datalen] = data[i];
        ctx->datalen++;
        switch (ctx->datalen) {
            case 64:
                sha1_transform(ctx, ctx->data);
                ctx->bitlen += 512;
                ctx->datalen = 0;
                continue;
            default:
                break;
        }
    }
next_iteration:
    // continue with the next iteration of the loop
}
```

Listing 42: Incorrect SHA-1 function sha1_update implementation that is compiler unstable; for some settings, the binary compiles, but for other settings, the binary does not compile. This function, when substituted into the original SHA-1 source code, could be compiled using gcc (using any optimization setting), but could not be compiled using any optimization setting in clang. This source code was generated by Mistral-7B-Instruct-v0.1 with an inference temperature of 0.4 and prompt number 4; this text includes the comments generated by the GPT model.

```
void sha1_init(SHA1_CTX *ctx) {
  ctx->datalen = 0;
  ctx->bitlen = 0;
  ctx->state[0] = 0x67452301;
  ctx->state[1] = 0xEFCDAB89;
  ctx->state[2] = 0x98BADCFE;
  ctx->state[3] = 0x10325476;
  ctx->state[4] = 0xc3d2e1f0;
  ctx->k[0] = 0x5a827999;
  ctx->k[1] = 0x6ed9eba1;
  ctx->k[2] = 0x8f1bbcdc;
  ctx->k[3] = 0xca62c1d6;
  // Initialize the state array with the given values
  for (int i = 0; i < 5; i++) {
    ctx->state[i] = 0x67452301 + (i * 0x10000000);
  }
  // Initialize the key array with the given values
  for (int i = 0; i < 4; i++) {
    ctx->k[i] = 0x5a827999 + (i * 0x10000000);
  }
}

Compiled binary output:
5ff45a0abee9f51d59a5cab9b366d181d6036819
e5a4bbb47f54d5000d3785fa323a6a7be218c1c0
ea8a8524a73f852afc4e9577849f4608cbdf3f85
8b70116f6e5feb596fcc511a33ca3e32d660eee4
```

Listing 43: Example of a function re-write that has a detected integer overflow flaw by the memory sanitizer check. The output does not change under different compiler optimization settings. This example was generated by Llama-2-70b-chat-hf with an inference temperature of 0.6 and prompt number 1.

```
void sha1_transform(SHA1_CTX *ctx, const BYTE data[]){
        WORD a = ctx->state[0], b = ctx->state[1], c = ctx->state[2], d =
              ctx->state[3], e = ctx->state[4];
        WORD t, m[80];
        size_t i = 0;
        for (i = 0; i < 16; ++i){
                m[i] = data[i*4];
                m[i] += data[(i*4) + 1]*256;
                m[i] += data[(i*4) + 2]*65536;
                m[i] += data[(i*4) + 3]*16777216;
        }
        for (; i < 80; ++i){
                m[i] = (m[i - 3] ^ m[i - 8] ^ m[i - 14] ^ m[i - 16]);
        }
        for (i = 0; i < 20; ++i){
                t = a + ((b ^ c) ^ (d & e));
                e = d;
                d = c;
                c = b;
                b = e | t;
                a = t;
        }
        for (; i < 40; ++i){
                t = a + ((b ^ c ^ d) ^ e);
                e = d;
                d = c;
                c = b;
                b = e | t;
                a = t;
        }
        for (; i < 60; ++i){
                t = a + ((b & c) ^ (b & d) ^ (c & d));
                a = d;
                d = c;
                c = b & e;
                b = e;
                e = t;
        }
        for (; i < 80; ++i){
                t = a + (b ^ c ^ d);
                a = d;
                d = c;
                c = b;
        }
        ctx->state[0] = a;
        ctx->state[1] = b;
        ctx->state[2] = c;
        ctx->state[3] = d;
        ctx->state[4] = e;
}

Compiled binary output:
0a4000c00a4000c00a4000c00a4000c00aa00000
0180000001800000018000000180000000000000
0000000000000000000000000000000000000000
0180000001800000018000000180000000000000
```

Listing 44: Example of a function re-write that has a detected integer overflow flaw by the memory sanitizer check. The output does not change under different compiler optimization settings. This example was generated by Mistral-7B-Instruct-v0.1 with an inference temperature of 1.0 and prompt number 7.

```
void sha1_init(SHA1_CTX *ctx) {
        ctx->datalen = 0;
        ctx->bitlen = 0;
        ctx->state[0] = 0x67452301;
        ctx->state[1] = 0xEFCDAB89;
        ctx->state[2] = 0x98BADCFE;
        ctx->state[3] = 0x10325476;
        ctx->state[4] = 0xc3d2e1f0;
        ctx->k[0] = 0x5a827999;
        ctx->k[1] = 0x6ed9eba1;
        ctx->k[2] = 0x8f1bbcdc;
        ctx->k[3] = 0xca62c1d6;
        // Obfuscated code to maintain the same functionality
        int i = 0;
        while (i < 5) {
                ctx->state[i] = (0x67452301 << (i * 8)) + (0xEFCDAB89 >> (i
                      * 8));
                ctx->k[i] = (0x5a827999 << (i * 8)) + (0x6ed9eba1 >> (i * 8)
                      );
                i++;
        }
        ctx->state[5] = 0xc3d2e1f0;
        ctx->k[4] = 0xca62c1d6;
}

Compiled binary output gcc with optimization level 0:
ca21665cf6c122ecc3be3b0aa37618403587f273
ca94ebd807dcbc3e989159d2ce11c2dc67b3e4bb
0455fb47ac91174c354a608c7776d54334770fe8
a587f9eb0d90443e99902763c5b14ceb48f95463

Compiled binary output gcc with optimization level 1:
f8ebaca7115ef6e9e1d125414213c6937fff7b56
2eaea1efb0550a5f6ea5f055d87cad1e0625d203
f87e351068dcd32680d4766ae4c144a487ad7b20
6508e1290b08f31b30cd01114482e22435962684

Compiled binary output clang with optimization level 0:
ca21665cf6c122ecc3be3b0aa37618403587f273
ca94ebd807dcbc3e989159d2ce11c2dc67b3e4bb
0455fb47ac91174c354a608c7776d54334770fe8
a587f9eb0d90443e99902763c5b14ceb48f95463

Compiled binary output clang with optimization level 2:
6269dd06d753f92010d9520d42239975c3d5dd73
25717503f460cc828f44bd855767afebc18517e5
67474bda460547bcdbef4c54c8c10fd36dfeede1
e7a86fb7710dd7066f0c1c6e4ab7f64d91a0b5a5

Compiled binary output clang with optimization level z:
ca21665cf6c122ecc3be3b0aa37618403587f273
ca94ebd807dcbc3e989159d2ce11c2dc67b3e4bb
0455fb47ac91174c354a608c7776d54334770fe8
a587f9eb0d90443e99902763c5b14ceb48f95463
```

Listing 45: Example of a function re-write that has a detected integer overflow flaw by the memory sanitizer check. The output hashes change depending on the compiler optimization settings that are used - and the output is always wrong. The generated output from the compiled binaries are given below the source code. The output hashes for the binaries compiled with gcc are the same for all optimization levels greater than 0, and are different for optimization level 0 (the output for optimization level 0 and 1 are given as examples). The hash outputs for the binaries compiled using optimization level 0 with gcc and clang are the same. This example was generated by Llama-2-70b-chat-hf with an inference temperature of 1.0 and prompt number 2. Note that the comment that the GPT model inserted with this source code is not correct.

```
void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len) {
    size_t i;
    for (i = 0; i < len; ++i) {
        ctx->data[ctx->datalen] = data[i];
        ctx->datalen++;
        if (ctx->datalen == 64) {
            sha1_transform(ctx, ctx->data);
            ctx->bitlen += 512;
            ctx->datalen = 0;
        }
    }
    size_t j = 0;
    while (j < 1000000000) {
        j++;
    }
    size_t k = 0;
    while (k < 1000000000) {
        k++;
    }
    size_t l = 0;
    while (l < 1000000000) {
        l++;
    }
    size_t m = 0;
    while (m < 1000000000) {
        m++;
    }
    size_t n = 0;
    while (n < 1000000000) {
        n++;
    }
    size_t o = 0;
    while (o < 1000000000) {
        o++;
    }
    size_t p = 0;
    while (p < 1000000000) {
        p++;
    }
    size_t q = 0;
    while (q < 1000000000) {
        q++;
    }
    size_t r = 0;
    while (r < 1000000000) {
        r++;
    }
    size_t s = 0;
    while (s < 1000000000) {
        s++;
    }
    size_t t = 0;
    while (t < 1000000000) {
        t++;
    }
    size_t u = 0;
    while (u < 1000000000) {
        u++;
    }
    size_t v = 0;
    while (v < 1000000000) {
        v++;
    }
    size_t w = 0;
    while (w < 1000000000) {
        w++;
    }
    size_t x = 0;
    while (x < 1000000000) {
        x++;
    }
    size_t y = 0;
    while (y < 1000000000) {
        y++;
    }
    size_t z = 0;
    while (z < 1000000000) {
        z++;
    }
}
```

Listing 46: Example of a function re-write that was found to be the single case found by Metric 24. This means that for some optimization settings an apparent infinite loop was encountered, but for other optimization settings the compiled binary was correct. Interestingly, this instance is actually a case where the infinite loop check of 10 seconds failed for the non-optimized compiled binaries, but for the optimized binaries the binaries executed within the required timeframe. This is because the code, shown above, is entirely correct, but the GPT model added several large for loops that make the compiled binary execute much longer than necessary. This code was generated by zephyr-7b-alpha with an inference temperature 0.4 and prompt number 4.