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

Elijah Pelofske∗, Vincent Urias1, and Lorie M. Liebrock2,1

1Sandia National Laboratories
2New 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 150,000 function re-write GPT output text blocks (that are potentially correct source code), approximately 50,000 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
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 test 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 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 function 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 function 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 utility used in the compilation) 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 utility used in the compilation) 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 utility used in the compilation) 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 utility used in the compilation) 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, 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 [63, 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. 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 Composability of Correct Function Rewrites

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 154,433 in total, but only 55,162 of those could be compiled with all compiler settings. Importantly, 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 2.3), 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 auxiliary 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 10685 SIGSEGV errors, 559 SIGABRT errors, 9 SIGILL errors, 3 SIGFPE errors, and 13 SIGBUS errors. Across all
<table>
<thead>
<tr>
<th>Metric</th>
<th>sha1_final</th>
<th>sha1_init</th>
<th>sha1_update</th>
<th>sha1_transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Rewrites</td>
<td>34,299</td>
<td>40,716</td>
<td>45,269</td>
<td>34,149</td>
</tr>
<tr>
<td>Metric 1</td>
<td>12,431</td>
<td>8,729</td>
<td>21,893</td>
<td>12,109</td>
</tr>
<tr>
<td>Metric 2</td>
<td>10</td>
<td>17</td>
<td>29</td>
<td>5</td>
</tr>
<tr>
<td>Metric 3*</td>
<td>10,842</td>
<td>7,672</td>
<td>19,906</td>
<td>10,214</td>
</tr>
<tr>
<td>Metric 4</td>
<td>1,105</td>
<td>922</td>
<td>696</td>
<td>1,568</td>
</tr>
<tr>
<td>Metric 5</td>
<td>6</td>
<td>21</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>Metric 6*</td>
<td>9,142</td>
<td>7,222</td>
<td>17,556</td>
<td>1,102</td>
</tr>
<tr>
<td>Metric 7*</td>
<td>1,700</td>
<td>450</td>
<td>2,350</td>
<td>9,112</td>
</tr>
<tr>
<td>Metric 8</td>
<td>10</td>
<td>0</td>
<td>39</td>
<td>3</td>
</tr>
<tr>
<td>Metric 9</td>
<td>99</td>
<td>29</td>
<td>156</td>
<td>209</td>
</tr>
<tr>
<td>Metric 10</td>
<td>15</td>
<td>6</td>
<td>39</td>
<td>12</td>
</tr>
<tr>
<td>Metric 11</td>
<td>133</td>
<td>75</td>
<td>398</td>
<td>111</td>
</tr>
<tr>
<td>Metric 13*</td>
<td>13 groups: 3-1014-5-1-5-1-1-1-1-1-1-1-1-1</td>
<td>31 groups: 19-14-20-3-2-3-1-4-1-2-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1</td>
<td>57 groups: 426-129-32-61-2-2-3-5-7-23-5-5-7-6-10-23-10-1-1-1-1-1-1-2-8-1-10-1-1-1-3-2-2-4-1-2-3-1-1-1-5-1-1-1-1-1-2-1-3-2-1-1-1-1-1-1-1</td>
<td>8 groups 1-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td>Metric 14*</td>
<td>611</td>
<td>357</td>
<td>1,519</td>
<td>8,743</td>
</tr>
<tr>
<td>Metric 15</td>
<td>0</td>
<td>13</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Metric 16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Metric 17</td>
<td>143</td>
<td>12</td>
<td>698</td>
<td>5</td>
</tr>
<tr>
<td>Metric 18</td>
<td>110</td>
<td>8</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>Metric 19</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Metric 20</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Metric 21</td>
<td>383</td>
<td>238</td>
<td>213</td>
<td>220</td>
</tr>
<tr>
<td>Metric 22</td>
<td>955</td>
<td>696</td>
<td>1,182</td>
<td>1,353</td>
</tr>
<tr>
<td>Metric 23</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>36</td>
</tr>
<tr>
<td>Metric 24</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Metric 25</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Metric 26</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Metric 27</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Metric 28</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Metric 29</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Metric 30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Metric 31</td>
<td>611</td>
<td>234</td>
<td>401</td>
<td>225</td>
</tr>
<tr>
<td>Metric 32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Metric 33</td>
<td>614</td>
<td>241</td>
<td>406</td>
<td>227</td>
</tr>
<tr>
<td>Metric 34</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Metric 35</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>Metric 36</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Metric 37</td>
<td>115</td>
<td>304</td>
<td>55</td>
<td>17</td>
</tr>
<tr>
<td>Metric 38</td>
<td>1</td>
<td>12</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Metric 39</td>
<td>658</td>
<td>683</td>
<td>207</td>
<td>1206</td>
</tr>
<tr>
<td>Metric 40</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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 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, 2228264. 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 function re-write produces a hash of length 160, and a different function re-write produces hashes of length 2228264 (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 4 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 5 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. Note that, counter-intuitively, more SHA-1 rewrites can actually cause more connections to be revealed by the compilers, and thus fewer unique function clusters be found by Metric 13.

So as to compare the various inference temperatures, GPT models used, and prompts, we use three high level metrics (that do not depend on the clustering of the various versions produced by different compiler settings); Metric 1 (was able to be compiled for all compiler settings), Metric 3 (compiled binary was output-verified for all compiler settings), Metric 7 (the compiled binary was output-verified for all compiler settings, and had source code distance greater than 0 compared to the original) as outlined in Section 2.2. For all three of these summary metrics, the larger they are the better the model and parameter choice is performing. Metric 7 is the most fundamental and important measure of code quality, and also being measurably different from the original source code. Figure 1 shows these three metrics as a function of inference temperature (across all 10 prompts and 3 GPT models). Figure 2 shows these three metrics as a function of the 10 different prompts. Figure 3 shows these three metrics as a function of the 3 different GPT models.
There are several clear findings from these aggregate performance metrics. Figure 1 shows that the proportion of compilable functions increases consistently as the temperature gets lower. However, Metric 7 peaks at a temperature of 0.4 to 0.2 - this shows that some stochasticity in the token choice is required for the generated code to be different from the original code, but too high stochasticity makes the code quality decrease as well. Figure 2 shows that the best performing prompt to maximize Metric 7 was Prompt number 7, and the worst prompt to maximize Metric 7 was Prompt number 5. Finally, Figure 3 shows that Llama-2-70b-chat-hf performed the best.

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 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).

```c
void sha1_init ( SHA1_CTX * ctx ) {
    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] = 0x6ED9BA11;
    ctx->k[2] = 0x8F1BBDCD;
    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] = 0x6ED9BA11;
            ctx->k[2] = 0x8F1BBDCD;
            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) {
    };
    for (int i = 16; i < 80; ++i, m[i] = (m[i - 3] ^ m[i - 8] ^ m[i - 14] ^ m[i - 16]));
    m[16] = (m[16] << 1) | (m[16] >> 31);
    for (int i = 0; i < 20; ++i) {
        WORD t = ROTLEFT(a, 5) + ((b & c) + (~b & d)) + e + ctx->k[i] + 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) + a + ctx->k[i] + 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[i] + 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[i] + m[i];
    a = 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[4] += e;
free(a);
}
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;
    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 :
6a9993a364706d6d6a3d257f175026d69d6d696d6
8d0d3e412b3265a4a8d4a1f9120d5e56d7203
3f17c56b54d9d677e44bf8107d1b953b1589
469756679b23333335f7203c7c3532440758a57

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 < 64) {
            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;
    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:
99fb15b4449bfac0469735d6e4d46f10e928b3d5a7d73536216e
9b78cc2ba938a199109990afa8f32bd3f7ae9

Compiled Binary output for clang with optimization level 1:
6ac40abe5f0953273b935aae424c87f71eef27a7f
2bcb9e9a57c3b773d23d93923990985cd235e
8dbb65d6e3a90fcb9397d80320abcd673a73003
d3e66c4075c151f334eb40064e4f1c5b5e4a

Compiled Binary output for clang with optimization level z:
99fb15b4449bfac0469735d6e4d46f10e928b3d5a7d73536216e
1ec6639ad22746767073b8ae2277f741100919b69f
d77db9cb453a9d58a95a7677b0f50f56e0
1ec6639ad22746767073b8ae2277f741100919b69f

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.
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, 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.

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.
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.
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.

Code Listing 4 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 clustering). In each of these functions, the code functions 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 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 5 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.
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 code re-writes - 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 5: 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


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-


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


A Hash Function Test Vectors

The four test vectors that are tested against:

- abc
- abcdedecedfedefghghjihjikjklklnlmnomnopq
- 1,000,000 repeats of the character a
- 5555555555555555555555555555555555555555555555555555555555555555

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

- a9993e364706816aba3e25717850c26c9cd0d89d
- 84983e441c3bd26ebaae4aa1f95129e5e54670f1
- 34aa973cd4c4a4f61eeb2bdaad27f316534016f
- 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)
  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 = datalen; i < 56; ++i) {
    ctx->data[i] = 0x80;
    while (i < 64)
      ctx->data[i] = 0x00;
    sha1_transform(ctx, ctx->data);
    memset(ctx->data, 0, 56);
  }

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[], size_t len){
  WORD i;
  i = ctx->datalen;
  if (ctx->datalen < 56) {
    ctx->data[ctx->datalen] = 0x80;
    while (i < 56)
      ctx->data[i] = 0x00;
  }
  for (i = ctx->datalen; i < 64; ++i)
    ctx->data[i] = 0x80;
  while (i < 64)
    ctx->data[i] = 0x00;
  sha1_transform(ctx, ctx->data);
  memcpy(hash, ctx->data, 32);
}
```
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.

Listing 10: Reference SHA-1 implementation header file

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

for (int i = 0; i < len; i++)
printf("%02 x", buf[i]);
pass = pass && !memcmp(hash3, buf, SHA1_BLOCK_SIZE);
sha_init(ctx);
sha_update(ctx, test4, strlen(test4));
sha_final(ctx, buf);
len = sizeof(buf);
printf("\n");
for (int i = 0; i < len; i++)
printf("%02 x", buf[i]);
pass = pass && !memcmp(hash4, buf, SHA1_BLOCK_SIZE);
return (pass);
}

int main() {
sha_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

List of markdown and miscellaneous code identifiers that are checked for after the triple back-tick output. This list is delineated by commas.

c, c++, python, ruby, sql, java, go, css, perl, hpp, rust, php, md, markdown, ts, lua, bash, ss, csh, kotlin, xml, matlab, vbnet, sh, yami, 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, javascript, json..., assembly, #!, awk, [c], temp, triple backquotes, uint32_t, [triple backquotes], uint8_t, document, backquote, instructions, bitlen, uint, word, byte, size_t, const, fsharp, csh, typename, std::size_t, state[], xxd, "hello, world!", data[], data[], csv, shellscript, erb, plaintext, ocl, shell

Note that, in nearly all of these cases the markdown language identifier was not correctly identifying the language that was generated, and actually include unrelated utility Linux commands or general programming keywords. Nevertheless, we do attempt to remove these language identifiers if they exist in order to increase the parse-ability of the GPT output. Each of these markdown identifiers is checked against the first line of strings in the parsed output, with all capitalization having been removed. Additionally, cases such as miscellaneous spaces before the markdown identifiers are checked for, as well as extra ancillary backticks; if any of these cases are found, the ancillary is removed.

D  Additional Examples of Fully Correct SHA-1 Function Rewrites

void sha_init(SHA1_CTX *ctx) {
    ctx->dalen = 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] = 0x6ED9BA86;
    ctx->k[2] = 0x8F1BBDC7;
    ctx->k[3] = 0xCA62C1D6;
    ctx->data[ctx->dalen] = 0;
    ctx->bitlen = 0;
    ctx->state[0] = cts->state[0] | 0x8000000000000000;
    ctx->k[0] = ctx->k[0] | 0x8000000000000000;
    ctx->k[1] = ctx->k[1] | 0x8000000000000000;
}

void sha_update(SHA1_CTX *ctx, const BYTE *data, size_t len) {
    BYTE *new_data = ctx->data;
    size_t new_dalen = ctx->dalen;
    size_t new_bitlen = ctx->bitlen;
    size_t new_state = ctx->state;
    size_t new_k = ctx->k;
    for (i = 0; i < len; i++) {
        new_data[new_dalen] = data[i];
        new_dalen += 4;
        if (new_dalen == 64)
            sha_transform(ctx, new_data);
    }

void sha_transform(SHA1_CTX *ctx) {
    for (int i = 0; i < len; i++)
        printf("%02 x", buf[i]);
    pass = pass && !memcmp(hash3, buf, SHA1_BLOCK_SIZE);
    sha_init(ctx);
    sha_update(ctx, test4, strlen(test4));
    sha_final(ctx, buf);
    len = sizeof(buf);
    printf("\n");
    for (int i = 0; i < len; i++)
        printf("%02 x", buf[i]);
    pass = pass && !memcmp(hash4, buf, SHA1_BLOCK_SIZE);
    return (pass);
}

int main() {
    sha_test();
    return (0);
}
new_bitlen += 512;
new_datalen = 0;
}

tctx->datalen = new_datalen;
tctx->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; i < 16; ++i, 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);
}
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;
}
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[64] = ctx->bitlen >> 8;
ctx->data[65] = ctx->bitlen & 0xff;
ctx->data[66] = ctx->bitlen >> 16;
ctx->data[67] = ctx->bitlen & 0xffff;
ctx->data[68] = ctx->bitlen >> 24;
ctx->data[69] = ctx->bitlen & 0xffffff;
ctx->data[70] = ctx->bitlen >> 32;
ctx->data[71] = ctx->bitlen & 0xffffffff;
sha1_transform(ctx, ctx->data);
for (i = 0; i < 4; ++i) {
  hash[i] = ((ctx->state[0] >> (24 - i * 8)) & 0xffffffff);
  hash[i + 4] = ((ctx->state[1] >> (24 - i * 8)) & 0xffffffff);
  hash[i + 8] = ((ctx->state[2] >> (24 - i * 8)) & 0xffffffff);
  hash[i + 12] = ((ctx->state[3] >> (24 - i * 8)) & 0xffffffff);
  hash[i + 16] = ((ctx->state[4] >> (24 - i * 8)) & 0xffffffff);
}
}

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_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 += 16;
                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 k[4] = {0x5A827999, 0x6ED9eba1, 0x8f1bbcdc, 0xca62c1d6};
    for (i = 0, j = 0; i < 40; ++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[4] += e;
}

void sha1_final(SHA1_CTX *ctx, BYTE hash[]){
    WORD i, j, k, l, m, n;
    i = ctx->datalen;
    if (i < 60) {  
        ctx->data[i++] = 0x80;
        while (i < 60)
            ctx->data[i++] = 0x00;
    } else {  
        ctx->data[i++] = 0x80;
        while (i < 60)
            ctx->data[i++] = 0x00;
        sha1_transform(ctx, ctx->data);
        memset(ctx->data, 0, 56);
    }
    ctx->bitlen += ctx->datalen * 8;
    ctx->data[56] = ctx->bitlen;
    ctx->data[64] = ctx->bitlen >> 8;
    ctx->data[65] = ctx->bitlen >> 16;
    ctx->data[66] = ctx->bitlen >> 24;
    ctx->data[67] = ctx->bitlen >> 32;
    ctx->data[68] = ctx->bitlen >> 40;
    }
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

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 and prompt number 3. The comments in the source code were generated by the GPT model.

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. 

```c
void sha1_final (SHA1_CTX *ctx, BYTE hash[]) {  
  WORD i;  
  if (ctx->datalen < 56) {  
    ctx->data[i++] = 0x00;  
    while (i < 56)  
      ctx->data[i++] = 0x00;  
  }  
  ctx->datalen = ctx->datalen;  
  for (i = 0; i < 5; i++)  
    hash[i] = (ctx->state[i] >> ((24 - (4 * i) - i * 8)) & 0x000000ff);  
  while (ctx->datalen < 56)  
    sha1_transform (ctx, ctx->data);  
  ctx->datalen = 0;  
  for (i = 0; i < 4; i++)  
    hash[i] = (ctx->state[i] >> ((24 - (4 * i) - i * 8)) & 0x000000ff);  
  sha1_transform (ctx, ctx->data);  
}
```
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.

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.

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_transform(SHA1_CTX *ctx, const char *data) {
    WORD a, b, c, d, e, i, j, k, m[80];
    for (i = 0, j = 0; i < 16; ++i, j += 4) {
        m[i] = m[i] >> 16;
    }
    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;
    }
    ctx->state[0] += a;
    ctx->state[1] += b;
    ctx->state[2] += c;
    ctx->state[4] += e;
}

Compiled Binary output:
5f51e0f5416b5d4635d2709e677e32360
54a826b037c646e446f8e623d46227316534018f
0e5656f4b64479f40eeae97c9ab3c5f25c0

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 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)

37
#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;
    }
}

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;
}

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.
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 clang with optimization levels 2, 3, fast resulted in a binary that correctly produces SHA-1 hashes for all 4 test vectors.

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.

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_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];
  if (*datalen_pointer == 64) {
    sha1_transform(ctx, temp_data);
    bitlen_pointer += 512;
    *datalen_pointer = 0;
  }
  *data_pointer = temp_data[k];
  ++(*datalen_pointer);
  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)
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];
a = d;
b = e;
c = ROTLEFT(b, 30);
d = b;
e = a;
}
for (; i < 40; ++i) {
t = ROTLEFT(a, 5) ^ (b ^ c ^ d) + e + ctx->k[1] + m[i];
a = d;
b = e;
c = ROTLEFT(b, 30);
d = b;
e = a;
}
for (; i < 60; ++i) {
t = ROTLEFT(a, 5) ^ ((b & c) ^ (b & d) ^ (c & d)) + e + ctx->k[2] + m[i];
a = d;
b = e;
c = ROTLEFT(b, 30);
d = b;
e = a;
}
for (; i < 80; ++i) {
t = ROTLEFT(a, 5) ^ (b ^ c ^ d) + e + ctx->k[3] + m[i];
a = d;
b = e;
c = ROTLEFT(b, 30);
d = b;
e = a;
}
ctx->state[0] += a;
ctx->state[1] += b;
ctx->state[2] += c;
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
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 0.4 with prompt number 2.

```c
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->bitlen += ctx->datalen * 8;
}
```

Compiled Binary output:
73038396beca31768b71a26bc64f9f5d1ea0b81e
4029913a19bf89bfd628764c745cec4ba38f8ed0
251aaa6cb2ee37df9df4a3249854a34071902e7
3e2d4c8ba2fef255c2d3ba42d7fcd29bc16a8a8d

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.

```c
void sha1_final ( SHA1_CTX *ctx , BYTE hash [] )
{
    size_t datalen = ctx->datalen ;
    if (datalen < 56) {
        ctx->data[ctx->datalen++] = 0x80;
        while (datalen < 56)
            ctx->data[ctx->datalen++] = 0x00;
    } else {
        ctx->data[ctx->datalen++] = 0x80;
        while (datalen < 64)
            ctx->data[ctx->datalen++] = 0x00;
        sha1_transform(ctx, ctx->data);
        memset(ctx->data, 0, 56);
    }
    ctx->bitlen += datalen * 8;
    ctx->data[59] = ctx->bitlen >> 32;
    ctx->data[60] = ctx->bitlen >> 40;
    ctx->data[61] = ctx->bitlen >> 48;
    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;
        i ++;
    }
}
```

Compiled Binary output:
9c80d0f72a11fe6f3919c20ced200a71ea6a9a93
b5e6e74a4b308239af8f0390c65cfb1aa2cdb465
14fd67290c0cb4d3a620d176d040de48ffb5d147
1b37789ce56800733b3ef0b8f0329c467
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 -> 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 = 0x800885F0;
    // Process the input data
    for (i = 0, j = 0; i < 16; ++i, j += 4) {
    }
    // Perform the rotation and addition operations
    for (; i < 80; ++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;
    }
    // Update the hash value
    w0 += m[0];
    w1 += m[1];
    w2 += m[2];
    w3 += m[3];
    w4 += m[4];
}

Compiled Binary output:
b61026ab101287aa83cabd16b9379693f0f8b3ec
ca7e0a10ad1ac98e20c02f7c15e1a47a1b7bf02f
0308edb90fc3b923f8744554143881914286a132
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;
        }
    }
}

void sha1_transform ( SHA1_CTX *ctx , const BYTE data [] )
{
    static const BYTE k[] = { 0xa5827999 , 0x6ed9eba1 , 0x8f1bbcdc , 0xca62c1d6 };
    WORD w[80];
    for (size_t i = 0, j = 0; i < 16; ++i, j += 4) {
    }
    for (size_t i = 16; i < 80; ++i) {
        w[i] = (w[i - 3] ^ w[i - 8] ^ w[i - 14] ^ w[i - 16]);
    }
    for (size_t i = 0, j = 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 [4] += e;
}
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.

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.
Listing 36: Incorrect SHA-1 function `sha1_final` 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.

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 h[6];
    WORD bitlen;
    WORD va = malloc(68 * 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 ^ 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(va);
            ctx->hitlen = ctx->datalen * 8;
            ctx->data[63] = ctx->hitlen;
            ctx->data[62] = ctx->hitlen >> 8;
            ctx->data[61] = ctx->hitlen >> 16;
            ctx->data[60] = ctx->hitlen >> 24;
            ctx->data[59] = ctx->hitlen >> 32;
            ctx->data[58] = ctx->hitlen >> 40;
            ctx->data[57] = ctx->hitlen >> 48;
            ctx->data[56] = ctx->hitlen >> 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) | (0x00000010 << 24));
    ctx->bitlen = (0x00000000 | (0x00000001 << 8) | (0x00000010 << 16) | (0x00000010 << 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] = (0x8f1bbcd3 | (0x8f1bbcd4 << 8) | (0x8f1bbcd5 << 16) | (0x8f1bbcd6 << 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)
    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_LEFT (c, 30);  // Overflow
        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] = 0x00000000;
        while (i < 56)
            ctx->data[i] = 0x00000000;
    } else {
        ctx->data[i] = 0x00000000;
        while (i < 64)
            ctx->data[i] = 0x00000000;
        sha1_obfuscate(ctx, ctx->data);
        memset(ctx->data, 0, 56);
    }
    ctx->bitslen -= ctx->datalen;
    ctx->data[63] = ctx->bitslen >> 8;
    ctx->data[62] = ctx->bitslen >> 16;
    ctx->data[61] = ctx->bitslen >> 24;
    ctx->data[59] = ctx->bitslen >> 32;
    ctx->data[58] = ctx->bitslen >> 40;
    ctx->data[57] = ctx->bitslen >> 48;
    ctx->data[56] = ctx->bitslen >> 56;
    sha1_obfuscate(ctx, ctx->data);
    for (i = 0, j = 4; ++i)
        hash[i] = ((ctx->state[0] >> (24 - i * 8)) & 0x000000ff) | hash[i]
            | ((ctx->state[1] >> (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[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] = 0x8c3d2e1f;
    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) ^ (b & d) ^ (c & d));
        a = d;
        d = c;
        c = b & e;
        b = e;
    }
    for (; i < 60; ++i){
        t = a + (b ^ c ^ d);
        d = c;
        c = b;
    }
    ctx->state[0] = a;
    ctx->state[1] = b;
    ctx->state[2] = c;
    ctx->state[4] = e;
}

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.

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 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.