Exploring and Improving Ciphertext-to-English Encryption(C2E) with Arithmetic Coding and GPT-2

Matt Xu
UW–Madison

Hao Zhang
UW–Madison

ABSTRACT

Ciphertext-to-English(C2E) encryption, as we call it, is a type of format-transforming encryption that transforms gibberish-looking ciphertext into readable English sentences, which appear less suspicious and conceal the existence of encrypted communication. We follow the methods introduced in a recent paper by Bauer et al. that achieve C2E using arithmetic coding and language models such as GPT-2 and build a functional C2E tool with the core ideas in their paper and our customization. In addition, we provide an extended version of our tool called HIDE-N-ARTICLE, which allows us to embed generated sentences inside an article to further enhance concealment.

1 INTRODUCTION

Encryption is an effective way to prevent adversaries from learning the content of communications as ciphertexts generated by secure encryption algorithms are computationally difficult for adversaries to directly extract secrets. However, ciphertexts often take the form of random gibberish and look nothing like normal human communication. Directly using them as messages can easily expose the existence of encrypted communication to adversaries who may have the power to compromise security in various ways without breaking the encryption. Therefore, we seek to achieve a type of format-transforming encryption, which we call Ciphertext-to-English(C2E), that produces encryption output in readable English sentences that do not raise suspicion and still provide security and correctness. During our research, we find a paper by Bauer et al. [3] that achieves the C2E functionality we desire by using arithmetic coding with GPT-2 [16]. In their design, an arithmetic decoder with GPT-2 as the token and frequency source transforms ciphertext into English sentences, which get processed into covert messages. When recipients obtain the covert messages, an arithmetic encoder, with the same GPT-2 model, will recover the same ciphertexts for decryption. After exploring the plausibility of the mechanisms described in their work and areas of improvement, we present the following contributions.

- We provide a functional C2E tool that employs arithmetic coding with GPT-2 following the ideas from Bauer et al. and includes our customization in them for usability.
- We provide an extension to our C2E tool called HIDE-N-ARTICLE, which allows us to embed generated sentences in articles for better concealment.
- We also provide analyses of the sentences generated and evaluate the performance of our tools.

The paper is organized as follows. In Section 2, we provide backgrounds of C2E including the threat model and related works. In Section 3, we detail our development process of the C2E tool in various stages. Section 3 also covers the functionalities of our tool, important findings during implementation, and descriptions of our customization. In Section 4, we elaborate on the ideas behind HIDE-N-ARTICLE and describe its mechanisms. In Section 5, we evaluate our tools' speed, usability, and security. We explore the future of our work in Section 6 and conclude in Section 7.

2 BACKGROUND AND RELATED WORK

Our motivation for studying C2E originates from the following threat model. We consider the scenario in which two people want to communicate secrets via online chatting, email, or commentary platform where communications are mostly regular English sentences or phrases. They face two different types of adversaries.

Authorities Adversaries of this type monitor and censor all traffic within the platform looking for secrets and other information of interest. They have no prior knowledge about which communication contains hidden information, but they try to detect suspicious communications by examining the content. When they have a target, they are powerful enough to take any action against either communicator to learn the secret or interfere with their communication.

Eavesdroppers This type of adversary knows that the communicators are sending secrets to each other and can eavesdrop on any messages in their communication. However, unlike *Authorities*, *Eavesdroppers* cannot take meaningful actions against the communicators. Their goal is simply to learn the secret, and we do not consider *Eavesdroppers* who work with *Authorities* since we can directly classify them as *Authorities*.

The goal of the communicators is to establish communication for sharing secrets between each other on the platforms described earlier without alerting the *Authorities* or revealing the secrets to *Eavesdroppers*. Since we assume communication platforms to be filled with English sentences, we should achieve the best concealment if we disguise our ciphertexts as English sentences. Thus, we begin our research in Ciphertext-to-English encryption and discover the work by Bauer et al.

The mechanisms of ciphertext transformation using arithmetic coding and the GPT-2 language model are not the only ideas presented in their work. In fact, Bauer et al. [3] have a complete design for passing covert messages on public platforms, which has additional mechanisms for authenticated encryption, post-processing

Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

for public platforms, and covert text detection for valid recipients. They also provide security analysis using several attack strategies for distinguishing covert messages from normal messages and show that their design can produce covert messages that are hard to detect for make machine detectors. Since we have limited time and resources, we decide to focus on exploring the core mechanisms in their work and continue our research from it in our direction. Nonetheless, parts of our customizations take inspiration from mechanisms by Bauer et al., and we also make assumptions about our work based on their results. We will note and acknowledge them as they appear in the rest of the paper.

Besides the design by Bauer et al., there are also other related works that aim at achieving similar goals or provide possible directions for achieving them. Some of them are closely related to C2E, while others are distinct from it.

Format Preserving Encryption Format-preserving encryption (FPE) encrypts plaintexts of a specific format into ciphertexts of the same format, such as credit card numbers and SSN, which are fixed-length strings of pure numbers [4]. We find it difficult to perform FPE on English sentences, but FPE is what inspires us to research FTE and C2E.

Format Transforming Encryption Format-transforming encryption(FTE) refers to encryption where the format of the output cipher text can be configurable. For example, Dyer et al introduce an FTE that formats the cipher text according to a regex [6]. C2E, as mentioned earlier, is a type of FTE.

Steganography Steganography is the technique of hiding secret information in another piece of data, typically images. For example, Generative Steganography Network [19] can transform some pieces of raw data into real-looking images and retrieve the original data from the image. C2E and Steganography both aim to avoid detection, but C2E produces new data instead of modifying existing data. Our Hide-n-Article may share more similarities, but they remain different.

Honey Encryption Honey encryption is a type of data encryption that produces a cipher text, which, when decrypted with an incorrect key that is guessed by the attacker, presents a plausible-looking yet incorrect plain text password or encryption key [9]. Honey encryption may not be an immediate fit as a solution to our thread model, but shows a different direction and possibilities in concealing the existence of secrets.

3 ROAD TO A FUNCTIONAL C2E TOOL

Since it is possible to treat the arithmetic coding algorithms and GPT-2 models as separate components according to Bauer et al. [3], we plan our implementation in stages starting from obtaining a functional arithmetic coding implementation with basic models and encryption library to completing our tool with GPT-2 integrated. For flexibility and easier integration with GPT-2, we use Python for all components of our tool. As a side note, we sometimes use the terms "generated sentences" and "transformed ciphertext" interchangeably due to the nature of our output. They usually mean the same thing.

3.1 Basic Models and Encryption Library

We build two basic models using a list of token-like words and corresponding frequency values that we randomly generate to begin our experimental implementation. They are the most simple models for arithmetic coding based on what we have learned from the tutorials by mathematicalmonk [12]. Both models provide simple but necessary functions that support getting tokens with corresponding cumulative frequency values and getting the cumulative frequency values for tokens, which are crucial for arithmetic coding.

Since we need a reliable source for secure ciphertext, We build a simple encryption utility module that uses the PyCryptodome to perform AES encryption and decryption in CBC mode assuming the library to be secure [17] [11]. Different from the approaches in the paper [3], we attach the 16-byte initialization vector to the produced ciphertext and use the new string for transformation. We have neither the tag structures nor the authenticated encryption scheme used by Bauer et al. [3]. Our choice here will be discussed in Section 5.3. We use the same encryption utility and final ciphertext format during our implementation of the functional C2E tool.

3.2 Initial Implementation of Arithmetic Coding for Transformation

We start our initial implementation by trying to understand the usage of arithmetic coding in the pseudocode by Bauer et al. [3]. A copy of their pseudocode is included in Appendix B, and we will only cover the most relevant aspects in this section due to its sophistication. The algorithm, according to Bauer et al., is similar to the ones by Rubin [18] and Howard & Vitter [8], but they modify it and use it differently from its regular application. Instead of a regular encode-then-decode operation, they first use the arithmetic decoder with the ciphertext in bits as the input and attempt to directly decode it into a piece of text as the transformation. The arithmetic encoder can then encode the transformation to recover the ciphertext and completes a decode-then-encode process. To avoid filling the paper with lengthy and repetitive descriptions, we only discuss details in their arithmetic coding mechanisms as they become important in the implementation processes, and the first one is the stopping mechanism for the arithmetic decoder.

In a regular encode-then-decode setting, the decoder can stop when the next token decoded is a unique symbol that represents the end of the original text that has been encoded [12]. If we use the decoder first with ciphertext as input, however, there is no stop symbol since we do not have an "original text" and the input is essentially random bits, which means we need a different way to stop the decoding process. The stopping mechanism used by Bauer et al. [3], according to our understanding, is to count the number of bits that have been used and are no longer involved in the computation of the next token. Once the count is the same as the number of bits in the ciphertext, it can stop and the decoded text should produce the same ciphertext bits when it gets encoded.

We note the importance of the stopping mechanism because it is the biggest obstacle during the initial implementation process. After having an initial understanding of the pseudocode by Bauer et al., we try to implement it verbatim but fail due to confusion about some details in their pseudocode. Therefore, we switch to a more basic arithmetic coding algorithm from the online tutorials [12] and try to modify it for ciphertext transformation believing the idea should work for arithmetic coding in general. Regular encode-then-decode works fine with our basic models, but once we use ciphertext from our encryption library and perform decode-then-encode, we hit the stopping problem. We try stopping when the target frequency is half, stopping when internal parameters exceed the length of the ciphertext, and stopping when the current output can be encoded back to the ciphertext. None of them achieves a high ciphertext recovery rate from decoded output or desirable performance. From our failure in modifying a simple arithmetic coding algorithm, we conclude that the stopping mechanism must be carefully designed and integrated into the algorithm, and we have to return to the pseudocode by Bauer et al. for a functional implementation.

3.3 Functional Implementation of Arithmetic Coding for Transformation

After further study of the pseudocode by Bauer et al., we think there are errors in it and decided to contact the authors with our findings and confusion. Fortunately, we soon received their kind and detailed response, which answers our confusion and confirms errors in functions **Adjust** and **Rescale** with correction. We have included details about them in Appendix B. They also provide us with their Python implementation for arithmetic coding, and we are able to patch our previously failed implementation with the pieces in their code. After connecting the patched implementation with our encryption library and basic models, with a 32-byte coding range, we are finally able to achieve 100% recovery of ciphertexts from encoding the output of the decoder. In addition, we have not yet encountered the case described by Bauer et al. [3] that the encoding output does not match the original ciphertext.

Before moving on to the stage of integrating GPT-2, we also build a new model called "english" that rotates between different cumulative frequency tables for different sets of tokens. Details about this model are in Appendix A. We build this model to verify the plausibility of having different sets of tokens while encoding or emitting the next token when we can ensure consistent cumulative frequency distribution for each set of tokens. The result of the verification is crucial for GPT-2 integration since we expect it to predict different words for us to complete a readable sentence. By testing with the "english", we not only obtain the verification we need but also gain additional insights about arithmetic coding, which will be discussed in Section 6.

3.4 GPT-2 Integration and Tool Construction

The reason that we use GPT-2 as the language model is not simply because it is the same model used in the paper by Bauer et al. We have considered using GPT-3 [5], but our trial implementation indicates that GPT-3 is not suitable due to the following problems. (1) Even though the cost per query to GPT-3 is not expensive, the arithmetic decoder and encoder make calls for each token, and the cumulative cost for processing one ciphertext can be too much. (2) We can only obtain 5 possible next tokens, and they often contain useless tokens such as a single newline character, making it not usable for generating the cumulative frequency distribution needed. GPT-2 does not have the problems above, and its public availability allows us to conduct experiments freely.

As mentioned by Bauer et al. [3], GPT-2 generates logits that contain probability information about multiple tokens as the next possible token, and they can be used for generating our cumulative frequency. After a lengthy search for appropriate API and documentation, we find a piece of demo code by McCaffrey [2] that contain complete function calls for obtaining both logit values and corresponding token as strings. Using McCaffery's code as a source of API documentation, we implement our GPT-2 based model that interacts with the arithmetic decoder and encoder as follows.

- Before the decoding and encoding process, it initializes the internal GPT-2 module and takes a string during initialization as the initial seed. We currently require the initial seed to be an English word with no spaces. It will then run its **Next** function to complete initialization.
- After initialization, the **Next** function should be called with the token as the input after each time the encoder obtains the cumulative frequency of the token and each time the decoder obtains the token from searching with a target cumulative frequency. **Next** updates the current seed by appending the input token to the current seed. If it is called during model initialization, **Next** simply sets the empty current seed to the initial seed. Since the current seed is a field of the class, it keeps all the appended tokens and grows with each call to **Next**. **Next** feed the updated current seed to the GPT-2 model and obtain the logits of the top $k = 3000$ possible tokens. It will then extract tensor values from the logits and convert them into frequencies using the regular Softmax function with $\beta = 1$. After extracting token strings from the logits, **Next** builds the cumulative frequency table with token strings and their corresponding computed frequency for arithmetic coding operations.
- **GetFreq** and **GetToken** functions, using the cumulative frequency table constructed by **Next**, respond to queries for tokens' cumulative frequency and queries for tokens with corresponding cumulative frequency values.

With all the pieces in place, we start constructing our encryption and decryption tools that work as follows.

- The encryption tool asks users to input the secret message to be encrypted and a word to be used as the initial seed and the first word in the generated sentences. The user can also provide no initial seed and let the encryption tool choose one for them. It then encrypts the secret with AES in CBC mode, concatenates the generated *IV* and *CT* as a new ciphertext, and feeds the new ciphertext to the arithmetic decoder backed by our GPT-2 model with the initial seed. After decoding, the encryption tool concatenates the tokens in the list generated by the decoder into one string, which is printed as the transformed ciphertext to the user. Users can choose to regenerate after seeing the output, and the tool will restart the process from encryption with the same secret.
- The decryption tool receives the transformed ciphertext, parses it into a list of tokens and separates the first token from the list as the initial seed for the GPT-2 model used by

the arithmetic encoder. After encoding with the remaining tokens, it will extract *IV* and *CT* from the encoding result and try decryption. If decryption is successful, the tool will print the secret message back to the user.

We now have a functional C2E tool, but it also requires additional modification and customization to be usable.

3.5 Customization for Usability

We implement the following customization and obtain a functional tool with much better usability.

Ignored tokens and duplicates: Some tokens generated by GPT-2 such as "\\n" and end-of-text tokens break the output sentences and are hard to encode. In addition, for tokens such as parentheses, it is difficult to guarantee that corresponding symbols will reasonably appear in the output, and without them, the sentence will look incomplete. We modify the **Next** function to eliminate those types of tokens from the lists obtained from the GPT-2 logits before generating the cumulative frequency table so that they can be ignored by the decoder and encoder. They should still function correctly since the resulting cumulative frequency tables remain consistent and deterministic after the removal of those tokens.

We also realize that tokens with the same string representation will lead to the incorrect pairing of token and cumulative frequency value, crashing the arithmetic decoder. Therefore we also eliminate the duplicates that appear later together with the tokens to be ignored.

Rotational seeds Our tool decodes long ciphertext more slowly as the generated output gets longer, and we identify the reason to be the size of the seed, which only grows longer and makes GPT-2 spend more time generating logits in the **Next** function. We want to limit the size of the seed but also ensure that subsequent tokens do not deviate from the previously generated sentences.

Our solution is a mechanism called rotational seeds. The model will first sets a value $N > 0$ as the size of the rotational seeds. When the current seed gets updated, it will check its content and find out how many complete sentences it has generated through previous updates. If there are more than N sentences in the current seed, our mechanism will keep only the previous N sentences. For example, if $N = 2$, and the current seed has the sentences $[S_1, S_2, S_3]$ in the order of when they are generated and added to the seed, S_1 will be removed from the current seed because we only want to keep the previous 2 sentences, which are $[S2, S3]$. With such a mechanism, we can limit the current seed to be N number of sentences plus some additional words that do not form a complete sentence. Our intuition is that sentences should provide enough context for subsequent tokens not to deviate, so removing and keeping only complete sentences is the safest way to ensure the efficiency and accuracy of sentence generation. We provide evaluations on the effectiveness of our rotational seeds mechanism in Section 5.1, and details on how we implement it can be found in Appendix D.

Decode-till-the-end : We observe that many generated sentences end abruptly and look incomplete, we have not found any explicit mention of such an observation in the paper [3]. After some testing, we realize that it is also due to the previously discussed stopping issue. Even though we can correctly stop the decoder, the token that the decoder stops on is unpredictable due to the randomness in the ciphertext. The decoder may very likely stop in the middle of a sentence and produce incomplete output.

As a solution, we let the decoder continue after meeting the original stopping condition and output additional tokens until it outputs one of the sentence-end tokens:[. ? !]. Our encryption tool also tells users which part of the output is additionally decoded. This customization makes all generated sentences complete, but it also makes the encoder produce extra bytes after the original ciphertext. Luckily, the number of random bytes seems to be low because the decoder can usually finish the sentence after only a few more tokens. Therefore, we can apply heuristics to the encoder output to eliminate the extra bytes and retry decryption if needed. We include an example of this scenario in Appendix C. In addition, we can modify the additionally decoded part of the sentence however we want since what it encodes to does not influence the ciphertext.

Parsing strategy and checks during encryption When the encoder parses the input sentences, it heuristically adds a space to the beginning of the input to transform the first word to a regular token and obtains the token list with the AutoTokenizer from transformers [1] following the usage example by McCaffrey [2]. Since the GPT-2 model in our tool also uses the same AutoTokenizer, we can directly and correctly parse most of our generated sentences. In contrast, Bauer et al. [3] use a special top-N parsing strategy to parse non-prefix-free token sets. We believe the difference is due to implementation and production details. We evaluate the success rate of parsing in Section 5.2.

To match the parsing strategy, we also modify the encryption tool and pad the initial seed in the same way. In addition, after obtaining the encryption result, the tool will parse the output in the same way it would be parsed during decryption and compare the parsing result to the original list of tokens obtained by the decoder. If they match, we indicate to users that the generated sentences can be correctly decrypted. Otherwise, we show them an error and ask them to regenerate. We can regenerate automatically for mismatches, but we want to let users make the decisions.

3.6 Weaknesses in Generated Sentences

Even though our functional tools can generate readable sentences, they still possess weaknesses if we use them to avoid alerting adversaries or evading detection. Supported by our evaluation results in Section 5, we identify the following major weaknesses in the output of our current tool:

- Unless we use only short secrets, they are inevitably long, and We might not be able to put them on certain platforms as a whole due to size limits.
- If we use them as a standalone message, they have no context, making the odd and conspicuous.
- Also, being a standalone message allows adversaries to easily isolate them from other regular messages if they find it suspicious and attack the entire transformed ciphertext.
- Lastly, some platforms do not usually see AI-generated messages or posts. Therefore, being detected as AI-generated can easily raise suspicion.

The fragmentation approach from Bauer et al. [3] may solve some of the weaknesses, but we aim to overcome them in a different direction using our extension called HIDE-N-ARTICLE.

4 HIDE-N-ARTICLE

4.1 The Idea of Hiding in Articles

Instead of using the sentences from transformed ciphertexts as short messages or posts, we propose embedding them in the middle of an article that can be separately composed, and the recipients will receive the entire article to recover the ciphertext from it and decrypt for secrets. We believe such an approach can overcome the previously mentioned weaknesses in the following way:

- Articles are usually long, and their length variations make the final article with transformed ciphertext embedded continue to be reasonable.
- We can compose the rest of the article based on the generated sentences to establish a reasonable context, making it more ordinary and less conspicuous.
- Since only a small part of the transmission, which is the entire article, is the transformed ciphertext, it adds extra burdens to the adversaries who attempt to isolate it.
- There exist many tools that make AI write articles for you. Especially with the recent release of ChatGPT [15], we believe that everyone will soon find it normal to see an AI-generated article and do not suspect that our articles contain anything secretive.

To allow our output to be embedded in any article, we need to modify and extend our existing tool so that when the decryption tool receives the entire article as input, it can correctly extract the transformed ciphertext and obtain the secret through decryption.

4.2 The Implementation as an Extension

Extension to the encryption tool: The encryption tool now performs additional processing for the ciphertext before feeding it to the arithmetic decoder as shown in Figure 1. We take inspiration from the idea of *sentinel value* in the paper by Bauer et al. [3] and construct a header value that gets attached to the final ciphertext. Therefore, the new ciphertext for the decoder has a new format of $header + IV + CT$. The encryption tool first constructs a plaintext header using the first 52 bits of the initial seed's MD5 hash and a size tag, which is the length of the original ciphertext divided by 16. We are aware that MD5 is not secure, but it is only for quickly generating a short hash with no security properties. It then encrypts the plaintext header using AES in CBC mode with a different key and produces the 32-byte encrypted header consisting of both the initialization vector and ciphertext, which is attached to the original ciphertext. The sentences produced by the decoder that pass the parsing verification can then be placed in the middle of an article or any other text. In addition, for the sake of decryption, the encryption tool requires the seed to be a full sentence with proper ending punctuation.

Extension to the decryption tool: The decryption tool needs to perform extra steps as shown in Figure 2 to identify and extract the transformed ciphertext from the input article. For each sentence

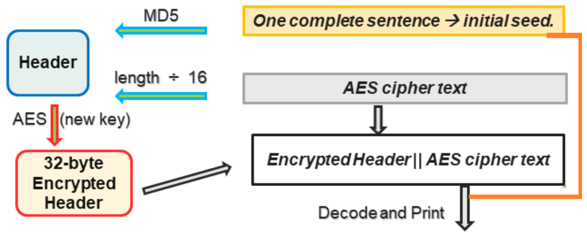


Figure 1: HIDE-N-ARTICLE's additional process for the header during encryption

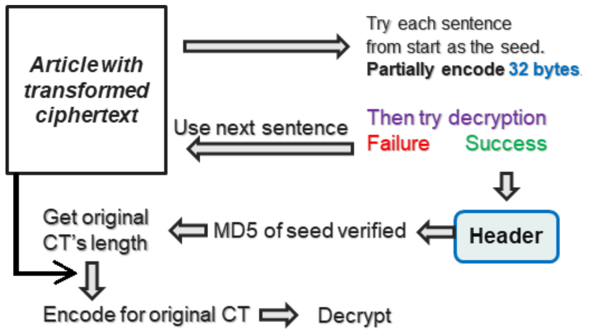


Figure 2: How the HIDE-N-ARTICLE processes the input article for decryption.

from the beginning of the input, the decryption tool uses the whole sentence as the initial seed and encodes the remaining tokens after that sentence. If the encoder can encode at least 32 bytes of data, the decryption tool will assume the data to be the encrypted header and run its decryption. If the decryption is successful, and the MD5 value in the header matches the MD5 value of the sentence used as seed, the size tag is extracted, and the encoder will continue until it has encoded the same number of bytes as the size tag multiplied by 16. If the decryption fails, the decryption tool will move on to the next sentence and repeat the above process until it can successfully decrypt a header or run out of sentences. If a header can be decrypted and verified, the final output from the continuation of encoding is the original ciphertext, and the original secrets can be extracted by further decryption.

If we simply use words as the initial seed, we need to run the above process with many more iterations since the number of possible initial seeds can be huge. The requirement of the initial seed is a compromise for having both efficiency and usability.

4.3 Remarks on our current implementation

With the completion of HIDE-N-ARTICLE, we have met our current implementation goal. Both encryption and decryption tools work as expected when we run them. However, despite being functional, they still have issues that will be discussed in our evaluation. For curious readers, we have also included details about finding our source code containing both the C2E tool and HIDE-N-ARTICLE extension in Appendix E.

5 EVALUATION

In this section, we provide evaluations of our tools in terms of speed, usability, and security.

5.1 Evaluation of Speed

Speed of the C2E tool

We measure the elapsed time for encrypting and decrypting five different secret messages on a non-GPU system. The results are shown in Figure 3 with the secret messages used listed at the bottom of each graph. For each secret message, we use the words [*"Until", "When", "My", "You"*] as initial seeds, and rotational seeds are not used because we want a more basic measurement of speed. The results indicate that longer sentences require a proportionately greater amount of time for both encoding and decoding.

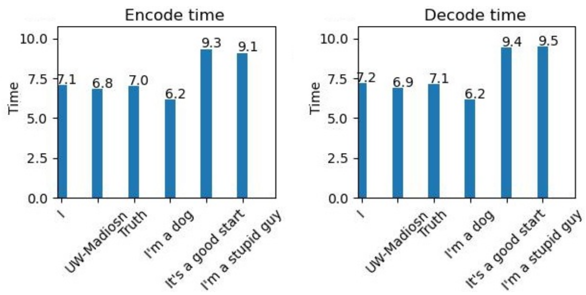


Figure 3: Average time for C2E to complete encryption/decryption in seconds

Change in speed from using rotational seeds

For our experiment that measures the effectiveness of rotational seeds, we run our encryption tool with different sizes of rotational seeds using the same 144-byte ciphertext and the word *"I"* as the initial seed. For each run, we measure the completion time of each call to the **Next** function as the decoder output grows longer. We perform the experiment on a virtual machine with two 4-core CPUs, and the results are shown in Figure 4.

We can observe from the results that if we use the entire output as the seed, which is what we do before adding rotational seeds, the time spent on each **Next** call increases almost linearly with the output length. As a result, without rotational seeds, the speed decreases dramatically during the decoding of long ciphertexts. Looking at the results that use rotational seeds, we can observe that they all deviate from the initial linear increase as the output becomes longer, and the time spent on each **Next** call, despite having significant variation, no longer increases linearly. Therefore, the speed will not have any significant decrease when the output becomes longer. Such results match our intuition that variation in sentences' length will cause variation in time, but overall changes will oscillate within some range instead of increasing indefinitely. We can also observe that using fewer sentences in rotational seeds can achieve keep the completion time long and achieve faster encryption, which is also true for decryption because it uses the same model.

Speed of HIDE-N-ARTICLE

With the HIDE-N-ARTICLE extension, the encryption tool's speed change is not significant because generating the header is a short and fast process. However, there's a significant decrease in the speed of the decryption tool due to the addition of a header searching process. To better understand the drop in speed, we measure the time spent during the decryption of the example article

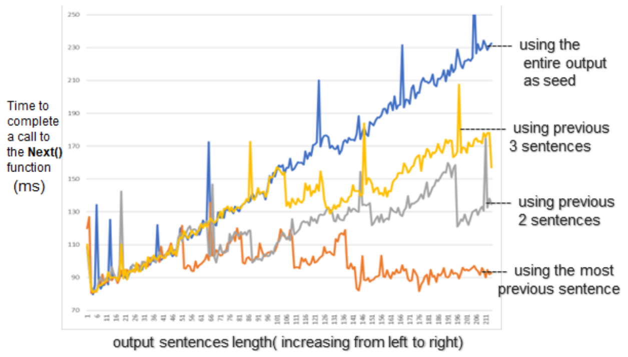


Figure 4: Changes in call completion time for different size of rotational seeds

in Appendix G.1 on a virtual machine with two 4-core CPUs, and obtain the following results.

Because the true initial seed is the sixth sentence in the input article, the decryption tool needs to perform header detection six times from the beginning. Each detection takes between 6.5 to 7.2 seconds, and completing the header detection process takes 40.9 seconds. After locating the original ciphertext using the decrypted header, the decryption tool then spends 24.6 seconds decrypting it for the secret message. The position of the transformed ciphertext in the article significantly affects the decryption time, and placing it at the beginning still incurs a noticeable increase in decryption time.

5.2 Evaluation of Usability

We evaluate the usability of the tools and their outputs in this section.

Success rate of decryption

The most important question about the usability of our tools the whether they can reliably produce and also decrypt the ciphertexts. For our decryption tool, decryption can only fail on a valid input if our parser cannot produce the same token list as the one generated by the decoder during encryption. Therefore, we perform experiments by generating a token list from decoding an AES ciphertext with the word *"They"* as the intial seed, formatting it to an output string, and comparing the token list from parsing the output string to the original list. We repeat the generation and comparison 100 times for AES ciphertexts with different lengths and find that 80% to 90% of the comparisons show equality, meaning an 80% to 90% chance of obtaining an encryption output that can be correctly decrypted. Since we have the mechanisms of informing users about possible decryption failure and allowing them to regenerate, we think our success rate is enough. When we use the tools for other experiments, we mostly regenerate because we want new sentences rather than receiving an output that cannot be decrypted.

Output from the C2E tool

When we examine the sentences produced by our C2E tool, the first noticeable characteristic we observe is their overall length. We generate sentences by giving different secret messages to our C2E encryption tool and calculate the average length of overall output over five runs for each metric. The results are shown in the left part of Figure 5, and the secret messages we use are listed at the bottom of the graph.

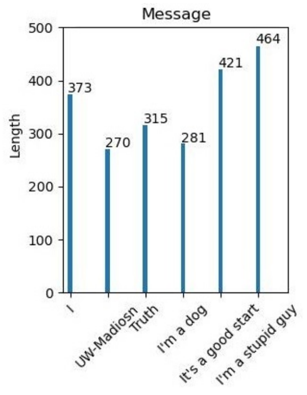
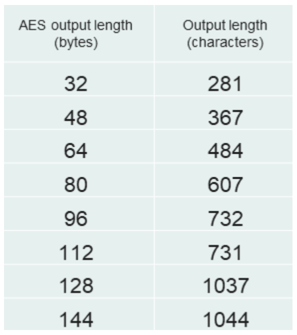


Figure 5: average length of C2E output in characters for different secret messages and initial seeds

We also measure the output length using the five different words as the initial seed for our C2E encryption tool while keeping other parameters identical. The results are displayed in the right part of Figure 5, and our choices of initial seeds are shown at the bottom of the graph.

The results suggest that the choice of the initial can lead to significant variation in output length. In addition, each output is much longer than the secret message, and the length seems to increase with the length of the secret message. Since different secret messages with some length differences will produce AES ciphertexts of the same length, we measure the length of the output with AES ciphertexts of different lengths to find the pattern of changes in length. We measure the length of final outputs with five different ciphertexts for each length category and use the word *"I"* as the initial seed for all runs. The results are shown inFigure 6, and they show that the output length increases almost linearly with the length of the AES ciphertext. Since we cannot compress the AES ciphertext or reduce the output length by splitting due to the linear relation in length, we conclude that our outputs will be inevitably long if want to encrypt longer secret messages.

Regarding the content of the generated sentences, they are often readable but appear to have a great variety. The initial seed can give a general direction of the meanings but have little control over details. Sometimes, the sentences even deviate from the initial seed after a few words and continue in a completely different direction.



AES output length (bytes)	Output length (characters)
32	281
48	367
64	484
80	607
96	732
112	731
128	1037
144	1044

Figure 6: This is the caption for my figure.

We also observe a few abnormal cases where GPT-2 outputs text that seems to be verbatim from its training data as they often look like separated components of a file or document. Examples of our outputs in those types can be found in Appendix Appendix F. We also notice some differences in output quality when we have different sizes for the rotational seeds, but the differences are too subtle for us to summarize. The uncontrollable nature of the generated sentences makes it difficult to find an output suitable for a given context. Even though we can quickly regenerate new sentences when finding the output not usable, it is frustrating to regenerate multiple times for one usable output.

Output from HIDE-N-ARTICLE

With our extension, the total length of the generated sentences increases due to longer inputs to the decoder and longer initial seeds, but it is not a problem under the design of HIDE-N-ARTICLE. We also verify that our decryption tool can still recover the secrets when we attach random sentences at either end of the generated sentences. However, the uncontrollable nature described earlier presents a new challenge. HIDE-N-ARTICLE allows us to establish the context by composing the article according to the generated sentences, but it turns out to be easier said than done. The meanings of the generated sentences may not be coherent, and composing articles is not our specialty. Therefore, after obtaining a more reasonable output, we use two strategies to save human effort, which are copying text from the internet and using other AI for text completion. We include examples in Appendix G.

5.3 Evaluation of Security

Protection of secrets

Since our models and implementations are not secret, adversaries can use our decryption implementation to encode any messages it wants and obtain either the ciphertext or the random strings. As a result, our tools heavily rely on the underlying encryption scheme to produce secure ciphertext. On the other hand, because our arithmetic decoder and encoder see ciphertexts as random strings, we can have great flexibility in the selection of encryption schemes. Therefore, we use symmetric AES encryption to make our implementation simpler and believe that we can easily switch to authenticated encryption(AE) or other encryption schemes.

Integrity of the messages

Without AE or signatures, the output of our C2E tool has little protection for integrity. Modifications of the output sentences will make them not decryptable, and adversaries can easily perform replay attacks. We assume that our HIDE-N-ARTICLE extension may offer some protection for integrity since the sentences of transformed ciphertext only make up a fraction of the overall message, and adversaries cannot pinpoint which part is the ciphertext unless they have the initial seed and the capability to decrypt the header. However, it relies on the encryption security of headers and does not prevent brute-force searches. In addition, the decryption order in HIDE-N-ARTICLE makes it vulnerable to attackers who can insert another output from HIDE-N-ARTICLE into an article that has the transformed ciphertext from HIDE-N-ARTICLE and make the decryption tool decrypt the wrong message if the insertion is before the transformed ciphertext. Not only the adversary may use this to mislead the recipients, but it can also use the recipient's reaction to know if it has inserted his output before or after the transformed ciphertext. Such knowledge may help it find the position of the transformed ciphertext in the article and use it for more future attacks.

Concealment of secrets

This aspect of security is the biggest goal of C2E, but it is also the hardest to evaluate for us. We have tried showing our results to people around us, but the feedback cannot be quantified into conclusions. Therefore, the current level of concealment provided by our tools relies on our assumption that some of our usable outputs would be similar to sentences written by real people, and a high similarity will avoid alerting the adversaries. We should quantify the concealment through more sophisticated experiments similar to the ones done by Bauer et al. [3] to verify our assumptions. However, due to our limitation in time and resources, we have only performed experiments for Jaccard similarity [13] between our outputs and outputs from ChatGPT [15].

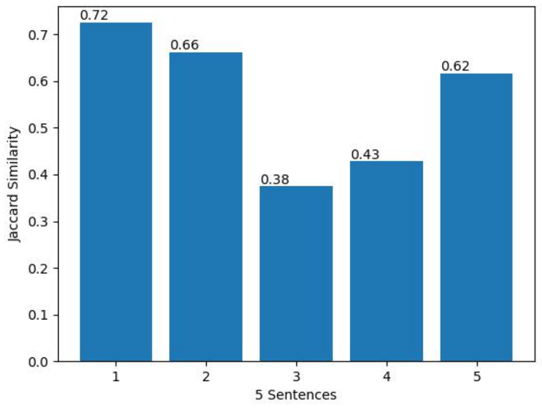


Figure 7: This is the caption for my figure.

We employ the Jaccard similarity measure on a sample of five randomly generated sentences from C2E. Details about the sentences we used can be found inAppendix H. The Jaccard similarity index ranges from 0, indicating no similarity between the compared sentences, to 1, indicating complete identity. The results, depicted in Figure 7, suggest that the generated sentences exhibit a high degree of similarity with those produced by ChatGPT, implying a degree of human-like linguistic qualities.

6 FUTURE WORK

For the next steps of our tools, the most obvious and important task is to fix the issues addressed in our evaluation such as the lack of authenticated encryption. We also need to improve the usability and continue to make the tool easier to use and more user-friendly. If time and resources permit, we would also like to perform a better security evaluation as suggested earlier.

Additionally, during our initial experiments with arithmetic coding with various models, we realize that the flexibility of the models used for arithmetic coding extends beyond language models. Since the most important criterion is a deterministic and consistent cumulative frequency table for the next tokens based on the context, it is possible to build models that use probability to produce output in different formats such as HTML documents and images. We believe arithmetic coding has the potential to be extended for more areas of format-transforming encryption. We plan on exploring it soon, and details about our plan can be found in Appendix E.

7 CONCLUSION

We have successfully implemented a functional Ciphertext-to-English tool for concealing the existence of encrypted communication following the ideas in the paper by Bauer et al. and implemented an extension, HIDE-N-ARTICLE, that seeks better concealment in a different direction. Although the current version still has some issues and needs further strengthening in security, it is mostly usable and produces promising results. We believe it is a solid start to further explore communication security and privacy.

Acknowledgements

Many thanks to our professor, Rahul Chatterjee, for guiding us from the beginning of our research. We could not have decided on a direction without him. Special thanks to James Howes, one of the authors of the paper we base our work on [3], for answering our questions and providing code examples. His timely response was a major turning point for our project, we could not have produced the results in this paper without him.

Break-down of contribution

Hao: The measurements and analysis for the part related to Figure 3 in Section 5.1, the part related to Figure 5 in Section 5.2, and the part about Jaccard similarity in Section 5.3.

Matt: Everything else.

REFERENCES

[1] Pytorch-transformers¶. (????). https://huggingface.co/transformers/v1.2.0/

[2] 2021. A predict-next-word example using hugging face and GPT-2. (Oct 2021). https://jamesmccaffrey.wordpress.com/2021/10/21/a-predict-next-word-example-using-hugging-face-and-gpt-2/

[3] Luke A. Bauer, IV Howes, James K., Sam A. Markelon, Vincent Bindschaedler, and Thomas Shrimpton. 2021. Covert Message Passing over Public Internet Platforms Using Model-Based Format-Transforming Encryption. *arXiv e-prints*, Article arXiv:2110.07009 (Oct. 2021), arXiv:2110.07009 pages. arXiv:cs.CR/2110.07009

[4] Mihir Bellare, Thomas Ristenpart, Phillip Rogaway, and Till Stegers. 2009. Format-Preserving Encryption. In *Selected Areas in Cryptography*, Michael J. Jacobson, Vincent Rijmen, and Reihaneh Safavi-Naini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 295–312.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *ArXiv* abs/2005.14165 (2020).

[6] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. 2013. Protocol Misidentification Made Easy with Format-Transforming Encryption. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer amp; Communications Security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2508859.2516657

[7] Chris Hoffman. 2017. What is open source software, and why does it matter? (Sep 2017). https://www.howtogeek.com/129967/htg-explains-what-is-open-source-software-and-why-you-should-care/

[8] Paul G. Howard and Jeffrey Scott Vitter. 1992. *Practical Implementations of Arithmetic Coding.* Springer US, Boston, MA, 85–112. https://doi.org/10.1007/978-1-4615-3596-6_4
[9] Ari Juels and Thomas Ristenpart. 2014. Honey Encryption: Encryption beyond the Brute-Force Barrier. *IEEE Security Privacy* 12, 4 (2014), 59–62. https://doi.org/10.1109/MSP.2014.67
[10] Jeff Kay. Why is the source code of Windows So Secret? what is there to hide? - quora. (????). https://www.quora.com/Why-is-the-source-code-of-Windows-so-secret-What-is-there-to-hide
[11] Legrandin. 2022. PyCryptodome documentation - buildmedia.readthedocs.org. (Jun 2022). https://buildmedia.readthedocs.org/media/pdf/pycryptodome/latest/pycryptodome.pdf
[12] mathematicalmonk. 2014. *Information Theory.* YouTube. https://www.youtube.com/playlist?list=PLE125425EC837021F
[13] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, Vol. 1. 380–384.
[14] OpenAI. (????). https://beta.openai.com/examples/default-ad-product-description
[15] OpenAI. 2021. ChatGPT: A Large Scale Open-Domain Chatbot. (2021). https://openai.com/blog/chatgpt/
[16] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI* (2019). https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
[17] Kenneth Raeburn. 2005. Advanced encryption standard (AES) encryption for Kerberos 5. (2005).
[18] F. Rubin. 1979. Arithmetic stream coding using fixed precision registers. *IEEE Transactions on Information Theory* 25, 6 (1979), 672–675. https://doi.org/10.1109/TIT.1979.1056107
[19] Ping Wei, Sheng Li, Xinpeng Zhang, Ge Luo, Zhenxing Qian, and Qing Zhou. 2022. Generative Steganography Network. In *Proceedings of the 30th ACM International Conference on Multimedia.* ACM. https://doi.org/10.1145/3503161.3548217

A DETAILS ABOUT THE "ENGLISH" MODEL

The "english" model has four sets of tokens. The first set contains common first names. The second set contains verbs that can directly precede a noun. The third set contains nouns, and the last set contains conjunctions. No duplicates exist within or cross sets. Each set also has its tokens' cumulative frequency values calculated and fixed over their corresponding set. The model keeps an internal counter starting at 1, indicating the first set of tokens. Whenever it receives a query for a token, it responds with the token with the requested cumulative frequency value frequency in the set indicated by the current value of the internal counter. Then it increments the counter by 1 or resets it back to 1 if it is already 4. When it receives a query for the frequency of a token, it searches through all sets to find the token and its cumulative frequency value. Since all tokens are unique, searching it is correct.

What will happen is that during decoding, the set will get switched to the next one each time the decoder outputs a new token. The resulting decoder output will take the format of *"name verb noun conjunction name verb noun conjunction name verb .....",* which is a close mimic of an English sentence, and that is where the name "english" comes from. As an example, one of our outputs is *"Sarah eats blanket since John touches baseball and Linda gets toothpaste if Barbara feels eyes since James finds blanket since Jessica uses toothpaste since Robert gets watch when William uses hairband since Barbara knows eyes if Sarah uses house since"*, which is produced by encrypting "UW-Madison" with the tool when it is functional and ready for integration with GPT-2.

B THE PSEUDOCODE WITH CORRECTIONS AND NOTES

We construct Figure 8 that contains a copy of the pseudocode for arithmetic coding in the paper by Bauer et al [3] as well as their descriptions of all the notations they use. We have also added the corrections and notes that we learn from our communication with the original authors. They are the red text and symbols in Figure 8.

C EXAMPLE OF SENTENCE OF ABRUPTLY

The text shown below with a different font is an example output of our encryption tool from encrypting the phrase "UW-Madison".

His ever growing lobbying in no less an admirable way than its social and political support. And things will improve once more finally as this election gets under way. As CBS News reporters Scott Pelley of Politico noted, the new House does not <u>pass by means of special meetings of the members reached--or to press to approve discussions.</u>

The part of the sentence before the underlined part is what the decoder outputs when it hits the old stopping condition, which is when the number of bits used for decoding is the same as the number of bits in the original ciphertext. We can see that if we only use that part, we will end up with an incomplete sentence that ends with the phrase "does not" and no verbs following it. It will look very awkward that way, so we let the decoder continue and finish the sentence with the underlined section, which ends with a period, one of the sentence-end tokens. Now the sentence is more complete and readable.

If we run the above text with the decryption tool, the encoder will first produce the following output if printed in hexadecimal:

$IV =$ 0x843eadfd708939199bf717eea5974be2

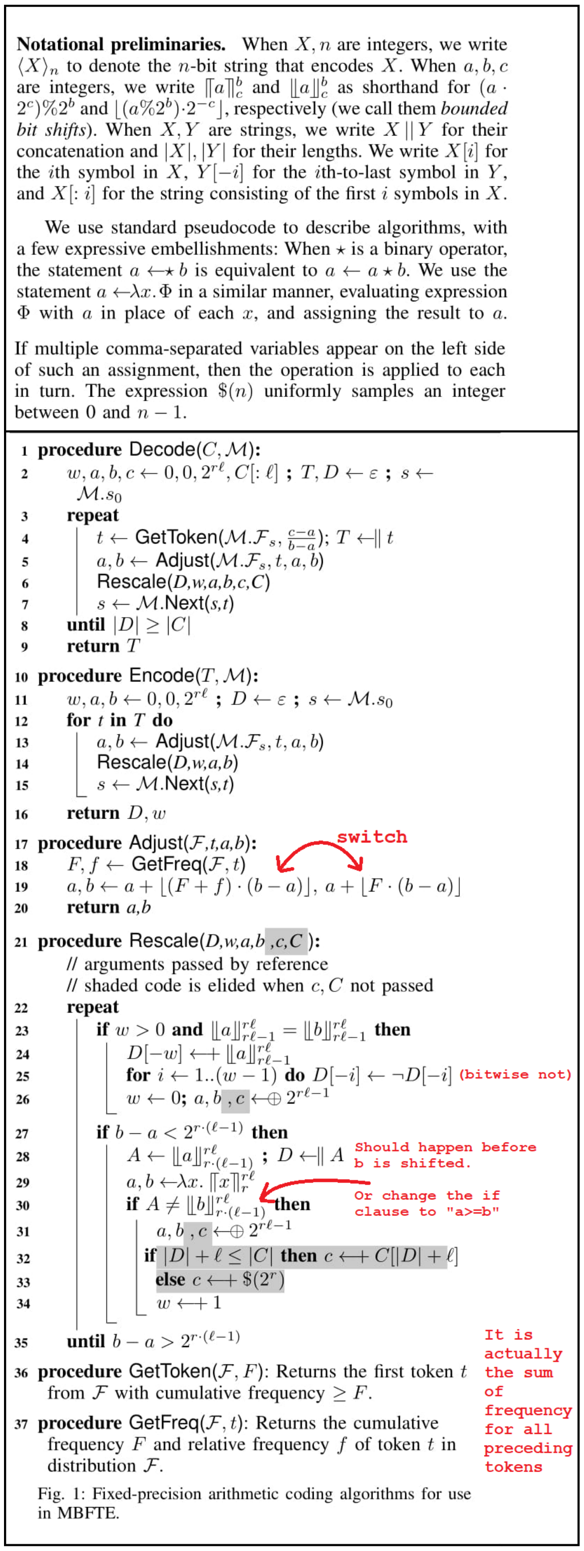
$CT =$ 0xff34be3a683426e90c34ca40e039506a
a93833648d98c7e387dfda888608fdad09

While the IV is correct, CT has extra bits from encoding the additionally decoded part. Since the receiver would not know which is the additionally decoded part in practice, our decryption tool currently uses heuristics that CT should be in 16-byte blocks. Therefore, it will first remove the "0x09" byte from the tail of CT and then try decryption. If the first try fails, which is the case here in this example, it will remove the last 16 bytes and try decryption again with " 0xff34be3a683426e90c34ca40e039506a" as CT. In our example scenario, " 0xff34be3a683426e90c34ca40e039506a" is the original CT, so the decryption will be successful and the message "UW-Madison" will be decrypted. If the second decryption fails, it will keep removing the last 16 bytes and retry until all ciphertext bytes are removed.

D DETAILS ABOUT THE IMPLEMENTATION OF ROTATIONAL SEEDS

Our implementation is slightly different from our earlier description, but they achieve the same goal.

When the model gets the size of rotational seeds N, it will construct an array of $N + 1$ empty strings called RS. The initial seed is then appended to the empty string at $RS[0]$ during initialization.



Notational preliminaries. When X, n are integers, we write $\langle X \rangle_n$ to denote the n-bit string that encodes X. When a, b, c are integers, we write $\llbracket a \rrbracket_c^b$ and $\llfloor a \rrfloor_c^b$ as shorthand for $(a \cdot 2^c)\%2^b$ and $\lfloor (a\%2^b) \cdot 2^{-c} \rfloor$, respectively (we call them *bounded bit shifts*). When X, Y are strings, we write $X \,||\, Y$ for their concatenation and $|X|, |Y|$ for their lengths. We write $X[i]$ for the ith symbol in X, $Y[-i]$ for the ith-to-last symbol in Y, and $X[:i]$ for the string consisting of the first i symbols in X.

We use standard pseudocode to describe algorithms, with a few expressive embellishments: When $\star$ is a binary operator, the statement $a \xleftarrow{\star} b$ is equivalent to $a \leftarrow a \star b$. We use the statement $a \xleftarrow{\lambda x.\Phi}$ in a similar manner, evaluating expression Φ with a in place of each x, and assigning the result to a.

If multiple comma-separated variables appear on the left side of such an assignment, then the operation is applied to each in turn. The expression $\$(n)$ uniformly samples an integer between 0 and $n-1$.

```
procedure Decode($C, \mathcal{M}$):
    $w, a, b, c \leftarrow 0, 0, 2^{r\ell}, C[:\ell]$ ; $T, D \leftarrow \varepsilon$ ; $s \leftarrow \mathcal{M}.s_0$
    repeat
        $t \leftarrow$ GetToken($\mathcal{M}.\mathcal{F}_s, \frac{c-a}{b-a}$); $T \xleftarrow{||} t$
        $a, b \leftarrow$ Adjust($\mathcal{M}.\mathcal{F}_s, t, a, b$)
        Rescale($D,w,a,b,c,C$)
        $s \leftarrow \mathcal{M}$.Next($s,t$)
    until $|D| \geq |C|$
    return $T$

procedure Encode($T, \mathcal{M}$):
    $w, a, b \leftarrow 0, 0, 2^{r\ell}$ ; $D \leftarrow \varepsilon$ ; $s \leftarrow \mathcal{M}.s_0$
    for $t$ in $T$ do
        $a, b \leftarrow$ Adjust($\mathcal{M}.\mathcal{F}_s, t, a, b$)
        Rescale($D,w,a,b$)
        $s \leftarrow \mathcal{M}$.Next($s,t$)
    return $D, w$

procedure Adjust($\mathcal{F},t,a,b$):                              switch
    $F, f \leftarrow$ GetFreq($\mathcal{F}, t$)
    $a, b \leftarrow a + \lfloor (F + f) \cdot (b - a) \rfloor,\ a + \lfloor F \cdot (b - a) \rfloor$
    return $a,b$

procedure Rescale($D,w,a,b$ $,c,C$ ):
    // arguments passed by reference
    // shaded code is elided when $c, C$ not passed
    repeat
        if $w > 0$ and $\llfloor a \rrfloor_{r\ell-1}^{r\ell} = \llfloor b \rrfloor_{r\ell-1}^{r\ell}$ then
            $D[-w] \xleftarrow{+} \llfloor a \rrfloor_{r\ell-1}^{r\ell}$
            for $i \leftarrow 1..(w-1)$ do $D[-i] \leftarrow \neg D[-i]$ (bitwise not)
            $w \leftarrow 0$; $a, b$ $, c$ $\xleftarrow{\oplus} 2^{r\ell-1}$
        if $b - a < 2^{r\cdot(\ell-1)}$ then
            $A \leftarrow \llfloor a \rrfloor_{r\cdot(\ell-1)}^{r\ell}$ ; $D \xleftarrow{||} A$      Should happen before
                                                        b is shifted.
            $a, b \xleftarrow{\lambda x.} \llbracket x \rrbracket_r^{r\ell}$
            if $A \neq \llfloor b \rrfloor_{r\cdot(\ell-1)}^{r\ell}$ then                Or change the if
                                                        clause to "a>=b"
                $a, b$ $, c$ $\xleftarrow{\oplus} 2^{r\ell-1}$
                if $|D| + \ell \leq |C|$ then $c \xleftarrow{+} C[|D| + \ell]$
                else $c \xleftarrow{+} \$(2^r)$
                $w \xleftarrow{+} 1$
    until $b - a > 2^{r\cdot(\ell-1)}$

procedure GetToken($\mathcal{F}, F$): Returns the first token $t$        It is
    from $\mathcal{F}$ with cumulative frequency $\geq F$.              actually
                                                               the sum
procedure GetFreq($\mathcal{F}, t$): Returns the cumulative            of
    frequency $F$ and relative frequency $f$ of token $t$ in           frequency
    distribution $\mathcal{F}$.                                     for all
                                                               preceding
                                                               tokens
```

Fig. 1: Fixed-precision arithmetic coding algorithms for use in MBFTE.

Figure 8: Pseudocode by Bauer et al. with notes and corrections

During each **Next** function, concatenating all strings in the order from *RS*[*N*] to *RS*[0] will produce the current seed. When the current seed gets updated with a regular token, it will be directly appended to *RS*[0] and a new current seed will be generated by the same concatenation process. If the token is a sentence-end token, the tool will replace each *RS*[*i*] with the value of *RS*[*i* − 1], so the newly formed sentence in *RS*[0] gets moved to *RS*[1]. Despite that the remaining strings in *RS* are still empty, they are still moved. Finally, it sets *RS*[0] back to the empty string, and continues.

As more sentences get constructed in *RS*[0] and moved to *RS*[1], the remaining strings in *RS* will also get the sentence moved by its previous element in *RS*. Therefore the least previous sentence at the moment will be in *RS*[*N*], and it will be overwritten when a new sentence gets moved from *RS*[0] to *RS*[1], making *R*[*N* − 1] overwrite *RS*[*N*]. It may not be obvious but the process described above achieves the same goals for rotational seeds as it always keeps the previous *N* sentences in the array elements *RS*[1] to *RS*[*N*] and still uses the incomplete and growing portion of *RS*[0] as the seed. It is identical to removing all sentences in the seed that is more precious than the previous *N* sentences.

In addition, for HIDE-N-ARTICLE, which requires a complete sentence as the initial seed, it will skip the initial appending and directly move it to *RS*[1] and starts with *RS*[0] being an empty string.

E SOURCE CODE INFORMATION

Our source code is publicly available on GitHub at the moment this paper is written. There are two repositories that have our source code.

- (**https://github.com/sufeidechabei/project-cs782**) This repository is where we begin our work and contains all the progress until we write this paper. We do not have plans on updating this repository as we want to keep it as a reference point for this paper. Some of the names and descriptions might not match what we write in this paper because we updated them as we compose the paper. We may correct them in the future. For future readers, we recommend using this repository for understanding what this paper does.
- (**https://github.com/a1m0le/ArthmtcCoding4FTE**) This repository also has all the source code we have so far, and we plan on using this repository for our future work. So this repository will be more frequently updated and maintained. For readers in the future, we recommend using this repository for accessing our latest progress.

In addition, since part of code is based on the work by Bauer et a., If we receive any request from the authors regarding changes to our repository such as visibility, we will respect their request and perform changes accordingly.

F EXAMPLE OUTPUTS FROM C2E

The following example outputs are produced by our C2E tool with 2 as the size of rotational seeds. The same secret message is used for all outputs, which is "UW-Madison".

Appendix F.1 is an example of what we consider to be usable.

Appendix F.2 is an example of the output that is somewhat readable, but the meaning quickly deviates from the beginning and becomes disorganized.

Appendix F.3 is an example of the output that cannot be correctly parsed by the decryption tool. In this example, the word "soften" produced by the encryption tool gets parsed into "soft" and "en" by the decryption tool, which will lead to decryption failure.

Appendix F.4 is an example of abnormal outputs. We believe GPT-2 is producing the content of a log file used in its training.

Abnormal not usable, also unable to parse

F.1 Usable output

He released a second statement explaining that an investigation was underway but that people had yet to reel in any extra information. Five homeowners said they were liens so they would wait for their liquid to settle. All don't want that separation.

F.2 Output that deviates

Unless deal marks 9 years or less of age. In addition, any alien born between 18 and 24 or required to become an adult applicant requires a post-Lawyer Application for Employment. THAT LIST APPLIES TO ENLISTED VEEERS WITH A PERIOD INEXIDENCE DEFINED AS BELIEVING A MOBILE VOW.

F.3 Output that cannot be parsed

Cheese then moistened water for 20 minutes to soften. Glass pot of sandwiches thrown to the heat in extra large stock pot, and sear for 4 minutes. Float into sandwich. – BBLC It's ok as long as it isn't drizzled.

F.4 Abnormal output

Could play on 64 launchers. | 2013-11-15 16:31:08.130 - Thread: 9 -> Loading class signatures for HInd.lib:... 20:A3D1248 NOTICE: Loading Provider from file C:\Program Files\Curse\bin\libdeldriverplugin.

G EXAMPLE ARTICLES USING OUTPUT FROM HIDE-N-ARTICLE

Appendix G.1 and Appendix G.2 are two examples of the articles we can produce with HIDE-N-ARTICLE.

For Appendix G.1, the secret message used for encryption is *"His HBO Max password is fire&blood123"*. The output produced by HIDE-N-ARTICLE in the article is shown as sentences with underlines. The rest of the article is composed by copying sentences from [7] [10] and filling some connecting and concluding sentences by ourselves. We do copying only to show that composing articles for our output by using other articles is possible. We acknowledge that ideas conveyed in these sentences are owned by the authors.

For Appendix G.2, the secret message used for encryption is *"Do you know that you are AWESOME!?"*. The output produced by HIDE-N-ARTICLE in the article is shown as sentences with underlines. The beginning part of the article is generated by GPT-3 using the "Ad from product description" example from OpenAI [14]. The rest of the article is written by ourselves.

G.1 An article about Windows and Linux

Microsoft Windows is probably the most popular piece of closed-source operating system out there. Being closed-source, Microsoft maintains the rights to every bit of code it has written, or bought, and does not allow others to use that code in their products. However, It does cause trouble to users. For example, no Windows user can take the Windows 7 interface, modify it, and make it work properly on Windows 8. On the other hand, we have Linux. Linux is a clean, fast, and widespread open source Unix. For many years, though, the idea of something as well documented in terms of an open source Unix project in a particular language had been required to justify the common use of open source Unix in many places, whether for commercial or academic use or commercial and competitive use across all lines. In recent years, I have experienced a more dignified awakening to the concept of the open source community. No longer and without some ephemeral glee in my voice, I couldn't see my way out of the armchair of competing efforts and debates arguing each other out of existence, though it was certainly painful. However, my country is adept at the art of steering clear of obvious players. Of the 28,000 members who have pledged their support so far for Red Hat Enterprise Linux 5, only 1,500 have had one word of encouragement from the open source community, while millions have worse intentions. Those involvements are the reasons why I am having second thoughts about open source. Being closed-source may have some problems, but it can keep me away from all those.

G.2 A recipe for creamy sauce

When I make my creamy sauce, I always use the parmesan cheese from Keese Cheese. Their products are made with only the freshest ingredients, and it's sure to please even the pickiest of eaters. It's a great source of calcium, protein, and other essential vitamins and minerals, so you can feel good about giving it to your kids. Plus, it's super easy to make and can be enjoyed as a snack or meal. Also, this cheese product is easy to store. Soups or Mummy Snacks may be substituted for grinded mushrooms or mummies to make this creamy sauce easier to maintain. Yield: 6-12 servings Ingredients 2 Tablespoons Lemon Crust Salt, 4 Tablespoons Parmesan Cheese, 1 1/2 Tablespoons Machine Wash Egg Yolk Instructions Preheat oven to high and line a baking sheet with foil. In a large bowl, whisk together cream and extra virgin olive oil. In a medium electric mixer, beat together yogurt, sugar and allspice. Add in the ingredients you want and beat to blend but don't let it get too soft. Continue to beat on low speed for 5 minutes, until combined. Pour in the yogurt. Beat on high speed. Stir in the dry bread flour. Mix to combine. Pull out the edges and run large oval jars through your tensioners for a layer of gentle sticky bread. Don't over heat the covered jars. Enjoy. It will help you decide which side of the experiment will leave the tiver. When I set on it on the table for Thanksgiving dinner, the kids abandoned their games and rushed to the dinner room like they need a last-minute touchdown.

H DATA USED FOR JACCARD SIMILARITY

The sentences used for Jaccard similarity are listed as below with each pair of C2E output(left) and CHatCPT output(right) labeled 1 to 5 from top to bottom, corresponding to the number labels in Figure 7.

#	C2E output	ChatGPT output
1	Until Blackwater lost. But Howard did end up writing a zine for his daughter [Emma Abreu] in which she commends Blackwater, saying that the stories of Jim Beam executive emeritus James Randi and his impressive employment production at NatWest, both left others to learn how to identify innovative opportunities.	Until Blackwater lost, Howard wrote a zine for his daughter Emma Abreu, in which he praised Blackwater for their innovative opportunities, citing examples from Jim Beam executive emeritus James Randi and NatWest's impressive employment record." This revised sentence provides a clearer and more coherent description of Howard's zine and the reasons for his praise of Blackwater. It also avoids repeating the name of Howard's daughter and uses proper punctuation.
2	Until a newsew set up of old people rush into what is expected to be a domineering and cross-dressing clown crowning his own. Earlier in the days I had seen a show in which I couldn't stand a clown dressed like Stephanie Sinclair, but my courage cost me.	Until a group of old people rushed into the room, expecting to see a dominant cross-dressing clown crowned as the new leader. Earlier in the day, I had seen a show where a clown dressed like Stephanie Sinclair made me uncomfortable, but I summoned the courage to watch it anyway.
3	Until, video trickling out of Wes Anderson's hospital, people asking me to meet them around town had been able to see me around the university. And even then, it always kept me from meeting anyone close to, say, Walter Cronkite about Asperger's disorder.	Until the release of a video from Wes Anderson's hospital, people had been asking me to meet them around town and see me around the university. Even then, it always prevented me from meeting anyone close to someone like Walter Cronkite to talk about Asperger's disorder.
4	Until, video trickling out of Wes Anderson's hospital, people asking me to meet them around town had been able to see me around the university. And even then, it always kept me from meeting anyone close to, say, Walter Cronkite about Asperger's disorder.	Until the video from Wes Anderson's hospital was released, people had been asking me to meet them around town and see me around the university. Even then, it always prevented me from meeting anyone close to someone like Walter Cronkite to talk about Asperger's disorder.
5	Until it ends, they'll let once more dounyonds set forth our hundred new of Iron Throne Empire Records. Continue reading or click here. –D2E01Z Don't the Children of Water Are Better Than The Children of Water?	Until it ends, they'll let the Iron Throne Empire Records release a hundred more downbeats. Continue reading or click here. –D2E01Z. Don't the children of water know that they are better than the children of fire?