USENIX Security '25 Artifact Appendix: A Framework for Designing Provably Secure Steganography

Guorui Liao, Jinshuai Yang, Weizhi Shao and Yongfeng Huang

Tsinghua University

A Artifact Appendix

A.1 Abstract

A.1.1 Artifact Description

The artifact includes the following main components:

1. **Python implementations of steganography schemes:**
   - Two uniform steganography schemes mentioned in the paper.
   - Three provably secure symmetric steganography schemes proposed within the framework introduced in the paper.
2. **Testing scripts:**
   - Enable readers to verify the correctness and security of the steganography schemes.
   - Validate other theoretical properties discussed in the paper, such as stability and the theoretical lower bound of steganographic capacity.
3. **Steganography scripts for text channels:**
   - Steganography scripts that operate on text channels, enabling the conversion of secret information into normal text using the provided schemes, ensuring that the generated text is indistinguishable from text naturally produced by large language models.
   - Verify the encoding and decoding correctness of text-based stegotext.
   - Provide related statistical metrics mentioned in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact implements steganography schemes and generally does not pose security or privacy risks. It is developed for academic research and is not intended for unauthorized or malicious applications, such as covert communication for illegal purposes. Users must adhere to ethical guidelines and comply with applicable laws. The text-based steganography implementation involves large language models, which may generate varied outputs based on their training data. While this artifact does not intentionally influence model outputs, users should be aware of potential biases or unintended behaviors in the generated text.

A.2.2 How to access

https://doi.org/10.5281/zenodo.14737116

A.2.3 Hardware dependencies

This artifact is designed to run on a CUDA-supported GPU for efficient execution by default. Specifically, a CUDA-compatible GPU with at least 32GB VRAM is required for running text-based steganography schemes using large language models (e.g., **Llama3-8B-Instruct**). However, smaller models can be run on GPUs with less VRAM, depending on the model size being used.

A.2.4 Software dependencies

Please ensure the following tools are available on your device:

- **Miniconda / Anaconda**: Used for managing dependencies and creating an isolated Python environment.
- **Jupyter Notebook**: Required for running verification tests of the steganography schemes. Ensure Jupyter Notebook is installed in your environment.
- **CUDA-supported GPU environment**: Necessary for executing steganography schemes efficiently. All provided scripts and code are designed to run on a CUDA-supported GPU.
- **Large Language Model**: Required for text-based steganography. Download from **Hugging Face**. We recommend **Llama3-8B-Instruct**, but you may use other models like **Qwen2-7B-Instruct**, etc.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

The following steps guide the installation of dependencies and the main artifact:

1. **Download the artifact:** The artifact is available as a ZIP archive on **Zenodo**. Download and extract it.
2. **Set up a Python environment:** It is recommended to use **Miniconda** or **Anaconda** for environment management. Create and activate a new environment with **Python 3.12**:

```
conda create -n fdpss python=3.12
conda activate fdpss
```

3. **Install required dependencies:** Install all necessary Python packages:

```
python -m pip install -r requirements.txt
```

4. **Download the large language model:** We recommend downloading the large language model used for text steganography (e.g., **Llama3-8B-Instruct**) to your local machine first and then using the local path to load the model.

After completing these steps, the artifact should be ready for evaluation.

A.3.2 Basic Test

To verify that the artifact is installed correctly, run the following command:

```
python TextStego.py --model <path_to_model>
```

When using a downloaded language model, replace <path_to_model> with its local path. If no errors are reported, the installation is successful.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The two uniform steganography schemes mentioned in the paper ensure **Security** and **Correctness**. Additionally, the Cyclic-shift Uniform Steganography scheme achieves near-optimal **Capacity**, satisfying the theoretical bound:

$$H(U(n)) \geq C(U(n)) > H(U(n)) - 0.0861.$$

These are proven by the experiment (E1) and described in Section 4.3 (**Security**, **Correctness**) and Section 5.1.1 (**Capacity**).

(C2): The three provably secure symmetric steganography schemes (Differential-based, Binary-based, and Stability-based) proposed in the paper ensure **Security** and **Correctness**. Additionally, the Stability-based scheme guarantees the ability to embed at least one bit given any distribution with a minimum entropy of at least 1 (**Stability**), and the Differential-based scheme achieves an **Capacity** that satisfies the theoretical bound:

$$H - \log_2(1 + H \cdot \ln 2) - 0.0861 \leq C \leq H.$$

These claims are validated through experiment (E2) and further detailed in Section 3.4 (**Correctness**), Section 3.5 (**Security**), Section 5.1 (**Capacity**), and Section 5.2 (**Stability**).

(C3): For implementing steganography on text channels, we provide a script (`TextStego.py`) that performs textual steganography, verifies encoding and decoding **Correctness**, and provides **statistical metrics** for a single sample. These experiments correspond to scenarios (A and B) described in Table 1 of Section 5 and demonstrate the relevant metrics for an individual sample. These claims are validated through experiment (E3).

See `Schemes_Verification(English).ipynb` for more details on the validation of C1 and C2.

A.4.2 Experiments

(E1): [less than 1 min]: To verify the uniform steganography schemes, open `Schemes_Verification(English).ipynb` and follow the steps in the section **Verification of Uniform Steganography**. Run the Jupyter Notebook blocks corresponding to:

- **Test 1.1: Security** validation – Ensures the sampled frequencies align with the given n-uniform distribution.
- **Test 1.2: Correctness** validation – Verifies that encoding and decoding produce consistent results.
- **Test 1.3: Capacity** validation – Tests whether the Cyclic-shift Uniform Steganography scheme satisfies the theoretical bound:

$$H(U(n)) \geq C(U(n)) > H(U(n)) - 0.0861.$$

The execution results, corresponding explanations, and expected outcomes are provided within the Jupyter Notebook itself.

(E2): [less than 30 mins]: To verify the provably secure symmetric steganography schemes, open `Schemes_Verification(English).ipynb` and follow the steps in the section **Verification of Provably**

Secure Symmetric Steganography. Run the Jupyter Notebook blocks corresponding to:

- **Test 2.1: Security** validation – Ensures that the frequencies of multiple steganographic sampling results align with the given distribution.
- **Test 2.2: Correctness** validation – Verifies that encoding and decoding produce consistent results.
- **Test 2.3: Stability** validation – Tests whether the Stability-based scheme always allows embedding at least one bit when the distribution has a minimum entropy of at least 1.
- **Test 2.4: Capacity** validation – Verifies whether the Differential-based scheme satisfies the theoretical bound:

$$H - \log_2(1 + H \cdot \ln 2) - 0.0861 \leq C \leq H.$$

The execution results, corresponding explanations, and expected outcomes are provided within the Jupyter Notebook itself.

(E3): [About 1 minute to run once under the default setting]:
Execution: Execute the following command:

```
python TextStego.py --model <path_to_model>
```

Results: The script performs encoding and decoding for text steganography and outputs relevant information. Specifically, it:

- Generates a secret key and initializes `PRG_Encode`.
- Uses the `Encode` algorithm with secret information from `bit_stream.txt` to produce stegotext.
- Displays the generated stegotext along with the embedded secret bits.
- Uses the same key to initialize `PRG_Decode` and apply the `Decode` algorithm to extract the embedded bits from the stegotext.
- Outputs verification results to check whether the extracted bits match the original secret bits.
- Provides statistical metrics per token, such as embedding capacity (`C`), entropy (`H`), `C/H`, the frequency of tokens with zero embedded bits (`NR`), and sampling time per token (`T`).

For more details, refer to the README under **Implementing Steganography on Text Channels → Execution and Output**.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.