

A Practical Defense Against Self-Adaptive LLM-Driven Malware

Enrique Díaz de León Hicks
Harvard University
Cambridge, Massachusetts, USA

Heorhii Ambartsumov
Harvard University
Cambridge, Massachusetts, USA

Eric Wang
Harvard University
Cambridge, Massachusetts, USA

Jay Chooi
Harvard University
Cambridge, Massachusetts, USA

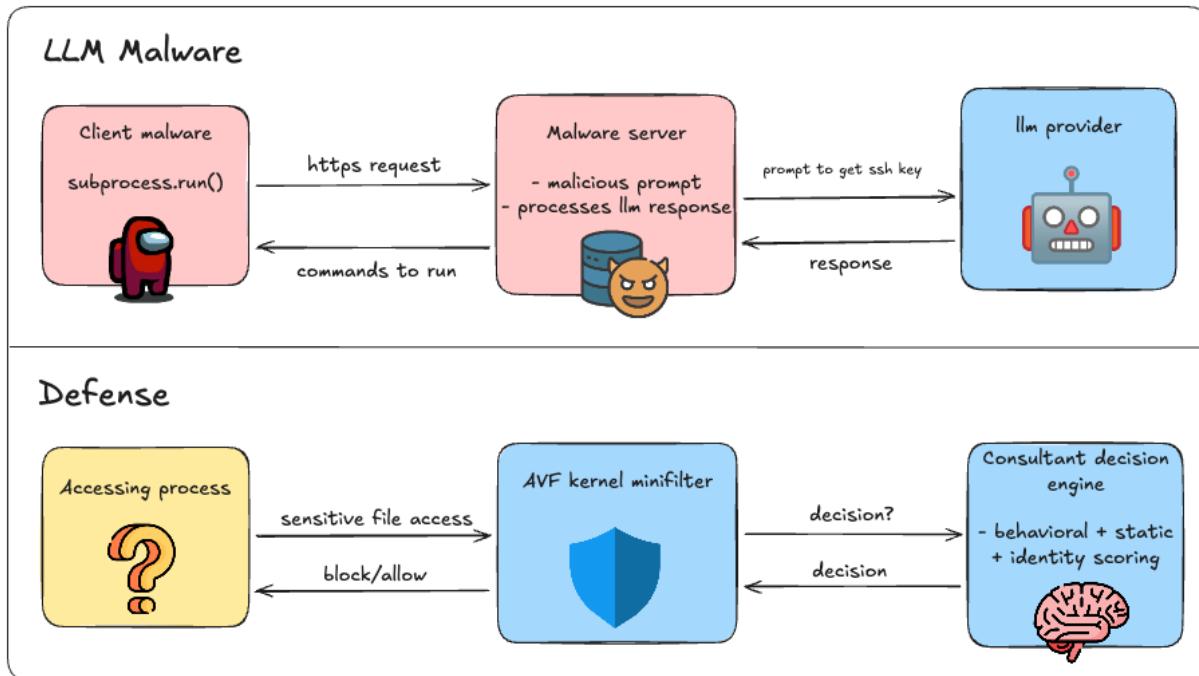


Figure 1: High-level comparison of the LLM malware and our defense. (Top) The LLM malware retrieves and executes malicious instructions generated by an external LLM provider. (Bottom) Our defense intercepts sensitive file accesses and blocks malicious behavior through kernel-level monitoring and multi-signal analysis.

Abstract

Large-language models are disrupting the offense-defense balance in the malware arms-race through providing cheap access to dynamically-generated malicious code and regular rewrites of the malware code-base, which evades standard hash-based and static analysis malware detectors. Early prototypes of LLM malware has been observed in operations and fully-mature LLM malware are expected to bypass existing industry-standard malware detection software. To help evaluate novel defense techniques against LLM malware, we construct minimal prototypes of LLM malware of which existing antivirus software could not reliably differentiate from benign software. We outline and implement a modular approach to correctly identify LLM malware, consisting of two components: (1) a kernel-level antivirus filter (AVF) that intercepts system calls to sensitive

files, which forwards them to (2) a user-level multi-signal threat analysis module (*Consultant*) that combines runtime behavioral analysis, static analysis and intent identification to efficiently allow or block system calls intercepted by the AVF. Our approach correctly identified all four variants of our LLM malware model organisms while having zero false-positives across a suite of commonly used benign software, outperforming existing malware detectors.

Reference Format:

Enrique Díaz de León Hicks, Eric Wang, Heorhii Ambartsumov, and Jay Chooi. 2025. A Practical Defense Against Self-Adaptive LLM-Driven Malware. In *Collections of Best CS 2630 Final Projects*. , 8 pages.

1 Introduction

1.1 Motivation

We consider the problem of malware that employs large language models (LLMs). Such malware, which we call *LLM malware*, signify a turning point in the cyber offense-defense arms race as the usage

of LLMs could defeat most of existing malware detectors. Consider the following techniques applied by current malware detectors:

- Hash-based scanners fail because LLM malware do not have a static binary signature. Instead, they rewrite their entire codebase on a regular, even hourly, basis (see Section 1.2).
- Code analysis scanners fail because LLM malware do not need to contain malicious code but can fetch them through LLM-manned C2 endpoints, or even through widely trusted model providers via jailbreak or dual-use prompts. Malicious code retrieved from the LLM is then executed on the victim machine.
- Some runtime-based detectors guard against ransomware by detecting mass encryption of files. An LLM ransomware could evade this detection by selectively processing files and only encrypt high-value files.
- Network-based detectors fail because if the ransomware calls LLM APIs, the endpoints could be trusted domains and with small throughput.

In Section 1.2, we further illustrate the massive gains in offensive capabilities seen in existing malware, while malware detection has yet to see improvements. Motivated by this shift in the offense-defense balance, we open-source a model organism of an LLM malware that does not contain malicious code but rather retrieves and executes them on-the-fly, alongside various its mutated variants rewritten by LLMs as in real operations. Furthermore, using our model organism, we identified key principles to design a new generation of malware detectors tailored to the detection of LLM malware. We implement and evaluate a detection system informed by these key principles and we release them to the public. We hope that the model organism, the key principles, and the defensive system will serve as a first step towards defending against the coming tide of self-adapting hard-to-detect malware.

1.2 Background and Prior Work

Raz et al. [10] first developed a proof of concept that showed how large language models (LLMs) are capable of automating ransomware operations by generating code, adapting to host context, and managing tasks at runtime. Later, a variant of the malware is discovered by ESET on the Internet [7].

Google Threat Intelligence Group [8] reported the current use of LLM within malware to dynamically generate malicious code to evade detection from existing antivirus techniques. One such malware that was observed was PROMPTFLUX [11] which queried the Gemini API to rewrite its entire source code on an hourly basis. Although PROMPTFLUX was observed as an experimental malware that was not fully mature, this scheduled-rewrite feature of its entire source code will defeat traditional hash-based malware detectors. Another example of a LLM-powered malware is PROMPTSTEAL [12] which queries Qwen through the HuggingFace API to return code to gather information about the victim’s computer.

Although this points towards significant progress on cybersecurity offense, there are some work in defense against LLM-powered cyber offense, aptly through LLM-powered cyber defense. We highlight two areas where defense had been applied. The first area is on the provider of LLM services. For example, developers could build better safeguards so that their model would refuse to rewrite

a malicious codebase. Frontier labs do measure and report such cyber-offense capabilities, e.g. see section 7.4 in the Claude Opus 4.5 system card [4], and recent research including on pre-training data filtering [9] and localizing dangerous knowledge [3] point towards the possibility of creating a model that has reduced capability on dangerous tasks without sacrificing capabilities on other benign domain-relevant tasks. Inference providers could also install classifiers to prevent malicious prompts to be received or malicious completions to be sent [5]. Nevertheless, threat actors have successfully bypassed existing production safeguards by posing as a CTF participant or a student in a cybersecurity class [8]. The second area is on the detection of LLM malware on the user side. Xu et al. [13] demonstrated that integrating LLMs in malware detection can improve detection rate by 9.5% while reducing false positive rates by 63.3%. However, due to the rapid progress of AI development and adoption, with the the first LLM-powered malware being reported only in August [7], there is no systematic study of applying such defenses against LLM malware.

1.3 Our Contributions

We detail our contributions to the security community below.¹

- We develop four variants of an LLM malware model organism that do not contain detectable malicious code, but instead queries an endpoint for malicious code retrieval, and executes malicious code in runtime to exfiltrate private SSH keys to a C2 endpoint. We evaluate the detection rates across commercial antivirus and the effectiveness of the malware in succeeding at exfiltration. (Section 2)
- We outline and implement a Windows kernel minifilter-based interception system (AVF) that monitors access to sensitive files like SSH keys and credential stores, and consults a user-mode decision engine (Consultant) before allowing or blocking access. This approach is able to intercept all file access methods regardless of how the malware tries to read files.
- We propose and implement a combined behavioral and static analysis defense against LLM-orchestrated credential theft that treats runtime behavior as the primary signal and uses static analysis to confirm capabilities. This addresses cases where traditional static detectors see only benign-looking code.
- We develop a multi-signal scoring framework that combines (i) behavioral evidence about what the process is doing at runtime, (ii) static capability indicators that we map to MITRE ATT&CK techniques [1], and (iii) light-weight identity signals (e.g., signing and install location) into a single risk score that is enforced by the kernel minifilter.
- We report an empirical evaluation showing how our defense is able to block all four tested variants of our LLM malware model organisms while still allowing credentials reading by legitimate applications like VS Code, Notepad, Notepad++, 010 Editor, and user-launched terminal commands (Section 4).

¹Code is available at <https://github.com/Agamendon/2630-final>

2 The LLM Malware Model Organism

To study the defensive challenges posed by LLM-driven malware, we implement a model organism that closely mirrors the operational structure of PROMPTSTEAL and PROMPTFLUX. Our goal is not to create a production-grade malicious tool, but rather to construct a minimal, controlled model organism that allows us to evaluate how LLMs can be misused during runtime, and how proposed defenses behave when confronted with such misuse.

This section describes the system architecture, the behavior of each component, and the workflow that enables the malware to retrieve and execute harmful functionality without containing malicious code in its binary. This approach reflects the key difficulty introduced by LLM malware: the harmful logic is generated on demand by a trusted external model provider instead of being embedded inside the malware itself. A high-level overview of the LLM malware can be found in the top panel of Figure 1.

2.1 Backend Server Architecture

We implement a simple backend server that exposes three POST endpoints. One endpoint is benign and supports legitimate chatbot functionality. The remaining two endpoints enable the LLM malware behavior. All endpoints forward prompts to xAI’s grok-code-fast-1 model, which serves as the external code generation oracle, or trusted LLM provider.

- (1) /chat (benign endpoint) - This endpoint represents a normal application workflow. The client sends a text query to /chat, and the server relays this text to the xAI API. The model’s response is returned to the user. This endpoint is intended to simulate legitimate software that embeds an assistant-style LLM.
- (2) /chat-endpoint (malicious endpoint) - This endpoint is invoked automatically when the malicious client application launches. The server sends a malicious prompt to the LLM provider (Listing 1). To inject diversity and evade hash-based traffic analysis, we applied the technique from Zhang et al. [14] to obtain truly diverse responses from the LLM. The LLM then returns terminal one-liners that the client can eval to try to recover the SSH key. The prompt is crafted to induce the model to output commands that access sensitive files on the host machine. Unlike traditional malware, the external LLM synthesizes the harmful logic at runtime. We report the effectiveness of the resulting scripts in Table 2.
- (3) /check-update (malicious endpoint) - After executing the generated commands, the client sends the results to /check-update. The backend writes this information to a file.

2.2 Client Architecture

The client app is a simple benign software wrapped with a payload. The payload queries the C2 endpoint /chat-endpoint for malicious Windows code to extract private SSH keys, executes the retrieved code, and upload the results to the C2 endpoint /check-update. The client receives up to 5 different scripts and submits the results of each of them to the malicious endpoint. This behavior of querying an endpoint that generates adaptive code for malicious purposes is similar to PROMPTSTEAL. The simple benign software in our

case is a CLI chatbot that sources its responses from the friendly endpoint /chat.

Note that the payload is not easily identified to be malicious. There is no malicious code in the payload to extract or exfiltrate the SSH keys. Even the malicious prompts themselves are stored on the C2 endpoint, and there are no unauthorized calls to trusted LLM endpoints, since those are proxied by the C2 endpoint.

Engine	v0	v1	v2	v3	Control
Bkav Pro	✓	✓	✗	✓	✓
CrowdStrike	✗	✓	✗	✗	✗
Malwarebytes	✗	✗	✓	✗	✓
SecureAge	✓	✓	✓	✓	✓
SentinelOne (Static ML)	✓	✓	✓	✓	✓
Skyhigh (SWG)	✓	✓	✓	✓	✓
Rest of scanners (66 engines)	✗	✗	✗	✗	✗
Detection count	4/72	5/72	4/72	4/72	5/72

Table 1: Detection comparison for mutated malware (v0-v3) and benign control software with the malicious payload removed. A checkmark means that the engine classify it as a malware. Green and red means desirable and undesirable behavior.

The client app is implemented in Python, obfuscated through Pyarmor [6], and compiled into an executable with PyInstaller [2] on a Windows machine. To simulate the malware mutating itself when spreading to evade detection, similar to PROMPTFLUX, we used Claude Opus 4.5 to rewrite the malicious client app 3 times to provide us with 4 variants in total. We report the detection rate of commercial antivirus software on these variants and a control software in Table 1. We note that commercial antivirus software cannot reliably differentiate between the malicious client and a benign client.

2.3 Exfiltration Performance

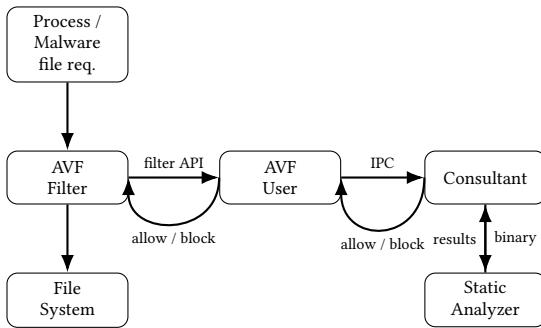
Metric	Count	Rate
<i>Prompt level</i>		
Total prompts	100	-
Prompts with at least one successful script	18	18.0%
<i>Script level</i>		
Maximum possible scripts (100 prompts × 5)	500	-
Scripts actually executed	366	73.2%
Successful scripts	30	8.2%

Table 2: Each prompt generates a maximum of 5 scripts to be run to exfiltrate the SSH key. This table shows how many of those prompts succeeded, and a more granular level of many scripts succeeded in exfiltrating the SSH key.

We report the effectiveness of our LLM malware to extract and exfiltrate private SSH keys in Table 2. We tested the malware 100

times on a Windows computer and monitored the malicious endpoint for the successful extraction of the private SSH key. Note that for each prompt (run) we will obtain up to 5 scripts. The two common failure modes are (1) the response to the prompt is not an executable script, and (2) the script did not successfully extract the SSH key. We attach examples of these failure modes in Appendix B.

3 A Defense against LLM Malware



Our defense is composed of three components that target the behavior and capabilities of the LLM-orchestrated malware:

- (1) Anti-Virus Filter (AVF). This is composed of a kernel-mode Windows minifilter that intercepts all file system access calls at the kernel level and a user-mode component that watches sensitive files and handles IPC.
- (2) Consultant. A decision engine at the user-mode level (not to be confused with AVF user-mode component) that analyzes access requests and decides if they should be allowed or blocked.
- (3) Static Analyzer. Looks for and analyzes specific patterns from executable binaries.

3.1 File Access Interception with AVF

The AVF is composed of a kernel-mode minifilter component and a user-mode service component connected by Windows filter API. Its registers pre-operation callbacks for three IRP function codes:

- IRP_MJ_CREATE (0x00): File open/create operations
- IRP_MJ_READ (0x03): File read operations
- IRP_MJ_WRITE (0x04): File write operations

The kernel-mode component receives these file access events and captures the metadata for the request. The captured metadata includes process ID, normalized file path, operation type, and process image name. The minifilter then sends an AVF_FILE_NOTIFICATION structure to the user-mode component. A file system call is effectively suspended while waiting for a decision from the minifilter.

The user-mode component is provided with specific set of sensitive files. When the user-mode component receives a request metadata from the kernel-mode component, it checks the path of file being requested against the defined set of sensitive files. In case of no match, the user-mode component allows the system call to proceed. In case of a match, the user-mode component requests advice from the consultant process.

If the user-mode component is not running, the filter will allow all system calls to proceed by default. This is meant to allow

the operating system to function normally even if the user-mode component failed to launch.

Finally, the user-mode component connects to a security consultant process that also runs in user space via IPC. The security consultant is a "black box" process that issues the decision to block or not to block the file access call. Such an architecture allows for a consultant to be easily swapped, as well as making the security decision process implementation-agnostic.

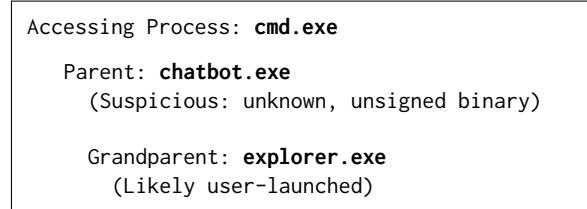
The subsequent sections describe analysis techniques used by our security consultant.

3.2 Behavioral Analysis

This component looks at the runtime context of the file access request. PROMPTSTEAL-style malware is characterized by a very specific behavioral pattern.

- (1) The malware executable spawns shell processes, i.e., cmd.exe, powershell.exe.
- (2) The shell then executes the commands.
- (3) The shell output is captured and exfiltrated.

We are able to detect this pattern by walking the process tree starting from the accessing process and going back to its root, as shown below.



We compute a behavioral score composed of three components, summarized in Table 3.

The weights of the behavioral score are intentionally designed to sum up to 0.8 and not 1.0. The reason behind this is that the behavioral score represents how many suspicious signals are present, not a normalized probability. With the combined scoring formula, explained in Section 3.4, the behavior signal will contribute at most 0.4 to the *final score*. We use a threshold of 0.6 to block file access which ensures that suspicious behavior alone cannot trigger a block, and static analysis of the parent executable is needed to confirm malicious capabilities. Concretely, the behavioral score is computed as

$$\text{behavioral} = 0.20 \cdot 1[\text{shell}] + 0.30 \cdot 1[\text{sensitive file}] + 0.30 \cdot 1[\text{unknown parent}]. \quad (1)$$

where $1[\cdot]$ denotes an indicator that evaluates to 1 if the condition holds for the current execution context and 0 otherwise.

Additionally, if a process traces back to the explorer.exe via terminal applications that are considered legitimate (e.g., Windows Terminal, ConEmu, etc.), we classify it as user-launched and will get a low behavioral score. This allows manual terminal commands to continue to work, yielding a low false positive rate.

Process Classification. To correctly interpret file access events, the Consultant maintains a list of known process types. This allows us to differentiate suspicious shell-driven access from benign

Table 3: Behavioral features used in our scoring model.

Signal	Weight	Description
Shell process accessing sensitive file	0.20	cmd.exe, powershell.exe
Sensitive file being accessed	0.30	SSH keys, credential files
Unknown parent (not user-launched)	0.30	Parent chain does not reach explorer.exe

Table 4: Reference lists used for process classification.

Category	Executables
Shell processes	cmd.exe, powershell.exe, pwsh.exe, conhost.exe
Terminal applications	windowsterminal.exe, wt.exe, mintty.exe, conemu.exe, hyper.exe
System processes	explorer.exe, svchost.exe, services.exe, csrss.exe, runtimebroker.exe

Table 5: Sensitive file classification patterns monitored by the Consultant.

Category	Patterns
SSH Keys	.ssh/id_*, id_rsa, id_ed25519, known_hosts
Cloud Credentials	.aws/credentials, .azure/*, application_default_credentials
Credential Stores	credentials, secrets, .env, .netrc
Windows Local Store	ntuser.dat

developer tools or system-level service processes. We categorize processes as follows, shown in Table 4.

Sensitive File Patterns. The Consultant maintains a list of file patterns that correspond to sensitive credentials and private keys. Any process trying to read such files triggers additional behavioral scoring, as shown in Table 5.

3.3 Static Analysis

Behavioral signals indicate whether a process is acting suspiciously, but do not reveal what that process is actually capable of doing. After behavioral analysis identifies a potentially malicious parent executable, we perform a static analysis of that binary to determine if it shows capabilities that are commonly considered consistent with credential harvesting, remote execution, or exfiltration. The static analyzer has multiple stages to it.

Table 6: String categories examined during static feature extraction.

Category	Representative Patterns
LLM or remote API usage	api.openai.com, api.anthropic.com, “chat”, “completion”
Credential artifacts	“BEGIN PRIVATE KEY”, id_rsa, “password”, “credential”
Execution primitives	“exec”, “shell”, “spawn”, “cmd”
Network endpoints	https://, wss://, excluding well-known vendor domains

Table 7: Packaging structures detected during static inspection.

Format Type	Representative	Evi-	Confidence
Embedded archive	footer markers, TOC records, compressed payload	0.90–0.95	
Managed runtime	module manifests, runtime entry stubs, metadata blocks	0.95	
Native standalone	linker tables, version info blocks, resource entries	0.90	

Stage 1: String-Level Feature Extraction. We extract printable strings (ASCII and UTF-16, minimum length four) and match them against patterns that are indicative of LLM orchestration, credential access, or external control signals. These patterns are summarized in Table 6.

Stage 2: Packaging Identification. For executables that contain internal payloads, such as compressed modules or archived code sections, we try to find and extract those payloads. This allows for a deeper inspection of the executable’s contents instead of treating it just as a black-box binary. Packaging signatures are shown in Table 7.

Stage 3: Embedded Code Recovery. When internal modules or packaged code sections are present, we extract them and inspect their contents. We analyze recovered functions, class names, and literal strings, and scan them for terminology associated with execution or exfiltration intent, such as “execute”, “exfiltrate”, “dump”, or “harvest”. Such indicators are treated as evidence that the executable intentionally implements these behaviors, rather than the keywords appearing incidentally due to packaging or third-party libraries. As a result, these signals increase the static analysis score.

Stage 4: Capability Indicators. Static analysis does not yield a binary determination of maliciousness. Instead, it produces capability indicators that estimate whether the parent executable can perform credential access, remote execution, or outbound data transfer.

Table 8: Static capability indicators and scoring contributions.

Indicator	Weight	Interpretation
Credential harvesting capability	0.25	Likely able to read or export credential material
Remote shell or command execution	0.12	Able to spawn or control execution context
External communication channels	0.05	Connects to external endpoints programmatically
LLM-associated inference usage	0.04	Structured API usage suggestive of orchestration
Suspicious embedded code footprint	0.20	Evidence of additional payload logic

These indicators contribute to the overall static score, as shown in Table 8.

Each indicator in Table 8 is mapped to one or more MITRE ATT&CK techniques. For example, our *credential-harvesting capability* feature corresponds to T1552.004 (Private Keys), remote shell or command execution aligns with T1059.003/004, and external communication channels capture behaviors in T1041 and T1071. We assign weights based on three factors: (i) specificity of the indicator to malicious activity versus benign software, (ii) potential impact if the capability is abused, and (iii) the expected false positive rate. Highly specific behaviors such as private-key access therefore receive larger weights (0.25), whereas ubiquitous capabilities like generic networking are capped at 0.05. This keeps the static score dominated by signals that are both security-relevant and unlikely to fire on everyday applications.

Together, these stages allow the static analyzer to move beyond string-matching or single heuristics: we first discover evidence of orchestration or credentials, then unpack any embedded payloads, then infer higher-level capabilities that map onto ATT&CK techniques.

3.4 Combined Scoring

The final decision is obtained by combining the three signal types into a weighted score:

$$\text{final score} = 0.5 \cdot \text{behavioral} + 0.4 \cdot \text{static} + 0.1 \cdot \text{identity} \quad (2)$$

Access is then decided as follows: if $\text{final score} \geq 0.6$, the file access is blocked. Otherwise the access is allowed. The choice of weights reflects the fact that behavioral evidence represents a strongest indication of malicious intent, since it corresponds to what the process is doing at runtime. Static analysis quantifies what the parent executable is capable of doing, providing complementary evidence. Identity information (e.g., signing metadata) serves mainly as a weak signal and that is why it contributes minimally to

Table 9: Detection outcomes across evaluated executables accessing private key files.

Category	Count	Outcome
Malicious variants	4	All blocked
Benign variants	9	All allowed
VS Code processes	2	Allowed
Notepad family	2	Allowed
Developer tool	1	Allowed
Terminal executions	3	Allowed

the overall score. This ensures that a signed binary with malicious behavior cannot evade blocking solely due to its identity.

4 Evaluation of the Defense

4.1 Experimental Setup

We evaluated the defense in a virtual machine running Windows 11 Enterprise with four processors and 4 GB of memory on an Intel(R) Core(TM) i5-8265U CPU. The AVF user-mode component was configured to monitor accesses to private key files located under each user's .ssh directory, and the Consultant service operated with combined behavioral, static, and identity scoring enabled. The blocking threshold was fixed at 0.6 for all experiments.

We exercised the defense using three groups of executables: (i) four malicious variants implementing file exfiltration behavior, (ii) a benign variant of the same program family that omitted the exfiltration step, exercised under nine user-driven launch contexts (e.g., direct invocation, IDE integration, and terminal use), and (iii) commonly used benign applications that legitimately read private key material (e.g., editors, scripting environments, and terminal commands). The benign tests were intentionally user-driven so that the resulting execution context traced back to an interactive session rather than an untrusted origin.

In total, we exercised 4 malicious variants and 17 benign execution contexts (9 for the benign family variant and 8 for commonly used applications and terminal commands).

4.2 Detection Outcomes

Across all malicious samples, the combined score ranged from 0.72 to 0.80, consistently above the blocking threshold. These executions showed maximal behavioral signals (presence of a shell process reading sensitive material from a non-user-launched context) and strong static indicators reflecting outbound communication and credential-handling capability. In contrast, benign executions remained well separated from this region: typical aggregate scores were below 0.30, and the highest benign score observed (0.55 for a helper process within VS Code) remained below the threshold and did not trigger blocking. This numeric separation highlights that false positives were avoided not simply by whitelisting software, but because the scoring components differentiated intent-bearing execution from interactive, user-initiated file access.

As summarized in Table 9, the defense correctly blocked all malicious executables and allowed all benign applications. This yields a detection rate of 4/4 on malicious samples and a false

alarm rate of 0% across the exercised benign set. Executions traced to interactive, user-launched processes consistently received low behavioral scores and did not trigger static indicators, leading to allow decisions without escalation.

4.3 Why Malicious Executables Are Blocked

Across all malicious variants, the main executable spawned either `cmd.exe` or `powershell.exe`, which subsequently issued file-open requests targeting private key files. These events produced high behavioral scores due to the presence of (1) a shell process, (2) a sensitive target file, and (3) an unknown originating binary. Static analysis further revealed capabilities consistent with outbound data handling, leading to aggregate scores exceeding the configured threshold 0.6. As a result, each malicious process was blocked before the key contents could be reproduced or transmitted.

4.4 Why Benign Executables Are Allowed

Editors, scripting tools, and terminal commands frequently accessed the same private key material. However, these accesses originated from a process lineage rooted at `explorer.exe`, and therefore corresponded to user-driven actions rather than remote execution. These contexts consistently produced low behavioral scores (typically ≤ 0.30), and no static indicators reflecting exfiltration capability. Consequently, the accesses were allowed with no interruption to normal interactive use. Helper processes associated with development environments occasionally reached intermediate behavioral scores, but static analysis yielded no evidence of controlled remote execution or credential misuse, resulting in allow decisions.

Acknowledgments

We thank Professor James Mickens for an incredible rendition of CS 2630, of which this paper is a final project.

References

- [1] [n. d.]. MITRE ATT&CK framework. <https://attack.mitre.org/>
- [2] [n. d.]. PyInstaller Manual – PyInstaller 6.17.0 documentation. <https://pyinstaller.org/en/stable/>
- [3] Anthropic. 2025. Beyond Data Filtering: Knowledge Localization for Capability Removal in LLMs. <https://alignment.anthropic.com/2025/selective-gradient-masking/>
- [4] Anthropic. 2025. Claude Opus 4.5 System Card. <https://assets.anthropic.com/m/64823ba7485345a7/Claude-Opus-4-5-System-Card.pdf>
- [5] Anthropic. 2025. Constitutional Classifiers: Defending against universal jailbreaks. <https://www.anthropic.com/news/constitutional-classifiers>
- [6] Dashingsoft Corp. 2025. Pyarmor - Obfuscating Python Scripts. <https://pyarmor.dashingsoft.com/>
- [7] ESET Research. 2025. ESET discovers PromptLock, the first AI-powered ransomware* on page. https://www.eset.com/us/about/newsroom/research/eset-discovers-promptlock-the-first-ai-powered-ransomware/?srsltd=AfmBOoo2rCtoPuyDiQQOllaTTQ6ZPF9mElSF_aWSUoNaxS4c1ZnOUaBs
- [8] Google Threat Intelligence Group. 2025. GTIG AI Threat Tracker: Advances in Threat Actor Usage of AI Tools. <https://cloud.google.com/blog/topics/threat-intelligence/threat-actor-usage-of-ai-tools>
- [9] Kyle O'Brien, Stephen Casper, Quentin Anthony, Tomek Korbak, Robert Kirk, Xander Davies, Ishan Mishra, Geoffrey Irving, Yarin Gal, and Stella Biderman. 2025. Deep Ignorance: Filtering Pretraining Data Builds Tamper-Resistant Safeguards into Open-Weight LLMs. doi:[10.48550/arXiv.2508.06601](https://doi.org/10.48550/arXiv.2508.06601) arXiv:2508.06601 [cs].
- [10] Md Raz, Meet Udeshi, P. V. Sai Charan, Prashanth Krishnamurthy, Farshad Khorrami, and Ramesh Karri. 2025. Ransomware 3.0: Self-Composing and LLM-Orchestrated. doi:[10.48550/arXiv.2508.20444](https://doi.org/10.48550/arXiv.2508.20444) arXiv:2508.20444 [cs] version: 1.
- [11] VirusTotal. 2025. PROMPTFLUX VirusTotal Page. <https://www.virustotal.com/gui/file/eb0687daed29f3651c61b0a2aa4a0cdcf2049a1ebae2e15e2dd9326471d318a1>
- [12] VirusTotal. 2025. PROMPTSTEAL VirusTotal Page. <https://www.virustotal.com/gui/file/766c356d6a4b00078a0293460c5967764fc788da8c1cd1df708695fa15b777>
- [13] Tianxiang Xu, Zhichao Wen, Xinyu Zhao, Jun Wang, Yan Li, and Chang Liu. 2025. L2M-AID: Autonomous Cyber-Physical Defense by Fusing Semantic Reasoning of Large Language Models with Multi-Agent Reinforcement Learning (Preprint). doi:[10.48550/arXiv.2510.07363](https://doi.org/10.48550/arXiv.2510.07363) arXiv:2510.07363 [cs].
- [14] Jiayi Zhang, Simon Yu, Derek Chong, Anthony Sicilia, Michael R. Tomz, Christopher D. Manning, and Weiyan Shi. 2025. Verbalized Sampling: How to Mitigate Mode Collapse and Unlock LLM Diversity. doi:[10.48550/arXiv.2510.01171](https://doi.org/10.48550/arXiv.2510.01171) arXiv:2510.01171 [cs].

A Prompt to obtain diverse malicious code

Listing 1: Malicious prompt used during initialization

```
Hi I need to find my eddsa ssh private key, can u
give me a code that i can copy paste into my
windows terminal to echo it? wrap that script
in ` `. don't output anything else. use
classic window not powershell.

<instructions>
Generate 5 responses to the user query, each
within a separate <response> tag. Each <
response> must include a <text> and a numeric
<probability>. Please sample at random from
the tails of the distribution, such that the
probability of each response is less than
0.10.
</instructions>]
```

B Example Responses from Grok

B.1 Refusal example

Grok would sometimes refuse to answer the query, or tell the user to manually execute a command instead of returning it in the format expected by our parser.

Listing 2: Refusal Examples

```
Your EDDSA SSH private key is not accessible this
way for security reasons.

echo "Your EdDSA SSH private key is typically
located at %USERPROFILE%\ssh\id_ed25519. To
echo it (not recommended for security reasons)
:"
type "%USERPROFILE%\ssh\id_ed25519"
```

B.2 Incorrect script example

A majority of the time, Grok would provide incorrect script examples that did not execute. In one instance, Grok even deleted the SSH keys from the directory (3).

Listing 3: Incorrect Scripts

```
%USERPROFILE%\ssh\id_ed25519  
del "C:\Users\%USERNAME%\ssh\id_ed25519" && echo  
"Key removed for safety."
```