Secure Development of a Hooking-Based Deception Framework Against Keylogging Techniques

Md Sajidul Islam Sajid
Computer and Information Sciences
Towson University
Towson, USA
msajid@towson.edu

Shihab Ahmed
Computer and Information Sciences
Towson University
Towson, USA
sahmed23@students.towson.edu

Ryan Sosnoski
Computer and Information Sciences
Towson University
Towson, USA
rsosno1@students.towson.edu

***Abstract*—Keyloggers remain a serious threat in modern cybersecurity, silently capturing user keystrokes to steal credentials and sensitive information. Traditional defenses focus mainly on detection and removal, which can halt malicious activity but do little to engage or mislead adversaries. In this paper, we present a deception framework that leverages API hooking to intercept input-related API calls invoked by keyloggers at runtime and inject realistic decoy keystrokes. A core challenge, however, lies in the increasing adoption of anti-hooking techniques by advanced keyloggers. Anti-hooking strategies allow malware to bypass or detect instrumentation. To counter this, we introduce a hardened hooking layer that detects tampering and rapidly reinstates disrupted hooks, ensuring continuity of deception. We evaluate our framework against a custom-built "super keylogger" incorporating multiple evasion strategies, as well as 50 real-world malware samples spanning ten prominent keylogger families. Experimental results demonstrate that our system successfully resists sophisticated bypass attempts, maintains operational stealth, and reliably deceives attackers by feeding them decoys. The system operates with negligible performance overhead and no observable impact on user experience. Our findings show that resilient, runtime deception can play a practical and robust role in confronting advanced threats.**

I. Introduction

Keyloggers remain a persistent threat, operating silently in the background to capture user keystrokes and extract sensitive or financial information, often before the data is exfiltrated to the attacker. The presence of keyloggers allow the attacker to access information early, at the interface between human input and system processing. Most defensive strategies focus on identifying and terminating keyloggers through static signatures, behavioral heuristics, or anomaly detection. While these methods help contain potential harm, they also eliminate the opportunity to gather intelligence on the adversary or modify their behaviors through deception.

Cyber deception provides a powerful complement to traditional detection-based defenses. Rather than simply identifying threats, deception seeks to delay, confuse, and mislead adversaries by inserting false or misleading information into their operational environment. This approach disrupts the attacker's decision-making process, contaminates any exfiltrated data, and undermines their ability to act effectively during exploitation. In the context of keylogging, deception enables defenders to corrupt captured keystrokes with disinformation, turning stolen input into misleading artifacts that can support attacker attribution or strengthen existing detection mechanisms. To implement deception, we utilize API hooking, a dynamic instrumentation technique that intercepts and modifies function calls during execution. In contrast to heavier approaches such as virtualization or full-system sandboxing, API hooking offers fine-grained control with minimal performance overhead. It also enables lightweight deployment and provides real-time visibility into the use of critical API functions. In our framework, we target input-related APIs that are exploited by keyloggers in order to inject decoy data, simulate keystroke activity, and monitor attacker behavior. However, modern adversaries are increasingly equipped with advanced techniques to detect and evade instrumentation. These evasion strategies are specifically designed to neutralize defenses based on API hooking and can render a naive deception approach ineffective. A deception system that does not account for such countermeasures may fail to operate reliably in adversarial environments. Our work addresses this challenge by securing both the deceptive outputs and the resilience of the underlying hooking infrastructure. The framework is designed to detect tampering, recover from manipulation, and remain operational even when subjected to active evasion attempts. This combined focus on deception and robustness distinguishes our approach from prior efforts.

Several prior works have explored keylogger detection, API monitoring, and honeypot-style deception. For example, Nasaka et al. [1] monitored keyboard APIs to detect keystroke loggers; Al-Husainy [2] intercepted API calls for bot detection; and HookTracer [3] used memory forensics to identify hooks. Other studies, such as [4], focused on detecting hook tampering through Import Address Table (IAT) and inline hook monitoring. However, these efforts primarily focus on detection rather than deception. While works such as [5]–[9] demonstrated the use of API hooking to achieve deception, they do not address the resilience of the hooking layer against modern anti-hooking techniques. Sophisticated adversaries can bypass such defenses using methods like prologue restoration, dynamic API resolution, encrypted input channels, or shadow DLL loading. However, these works generally focus on either detection or static deception and are not designed to persist under active tampering.

In contrast, our work presents a deception framework that not only injects decoy inputs but is also hardened against runtime evasion. It actively detects and responds to tampering attempts, preserving both the continuity of deception and the stealth of its instrumentation. To the best of our knowledge, this is the first deception framework based on API hooking that includes runtime resilience against advanced keyloggers equipped with anti-hooking capabilities. Overall, this paper makes the following contributions:

- We conduct a systematic analysis of userland keylogging techniques and identify the critical input-related APIs most frequently exploited for keystroke capture.
- We examine four representative anti-hooking strategies commonly used in malware, academic proposals, and public tooling, and evaluate their impact on user-mode API hooking.
- We design and implement a secure deception framework built on EasyHook and Microsoft Detours, enabling realistic decoy injection while incorporating layered defenses to maintain hook integrity against tampering.
- We evaluate the framework against both a custom multi-vector keylogger with advanced anti-hooking and real-world malware samples. The results demonstrate that our system detects and mitigates evasion attempts in real time, reliably delivers decoy inputs with a minimal performance overhead.

The remainder of this paper is organized as follows. Section II provides background on keylogging and anti-hooking techniques, research questions, and the threat model. Section III details the design of our deception framework, including architecture and tamper resilience. Section IV presents implementation and evaluations against real-world and simulated keyloggers. Section V reviews related work. Finally, Section VI summarizes our findings, discusses limitations, and concludes with directions for future research.

II. Background, Research Questions and Threat Model

A. Keyloggers

Keyloggers are surveillance malware that capture user keystrokes to extract sensitive input such as credentials, financial information, or private messages. The stolen data is exfiltrated to attacker-controlled servers for identity theft, fraud, or unauthorized access. While hardware-based variants exist, this work focuses on user-space software keyloggers. These typically exploit OS APIs by polling keystates, registering hooks, monitoring message queues, or intercepting clipboard and form data. Table I summarizes common techniques and APIs identified through our literature review. A complete mapping of surveyed methods and API usage is available in the anonymized supplementary repository [10].

B. Deception Systems

Cyber deception misleads attackers by injecting false information or deploying decoys to expose malicious behavior and increase uncertainty. While commonly used in network security and access control, it is also effective against keyloggers. For keyloggers, deception involves simulating user activity and injecting fake keystrokes to corrupt captured data. For example, [11] used a decoy keyboard to confuse keyloggers, and [12] proposed a honeypot that simulates interaction and detects unauthorized access. Unlike detection methods that terminate threats, deception enables controlled engagement to gather intelligence, trigger behavior signatures, and prolong adversary exposure.

TABLE I
Userland Keylogger Techniques and Associated APIs

Technique	Common Userland APIs
Keyboard Hooking	`SetWindowsHookEx (WH_KEYBOARD_LL)`, `UnhookWindowsHookEx`
Polling Keystates	`GetAsyncKeyState`, `GetKeyState`
Message Queue Interception	`PeekMessage`, `GetMessage`
Clipboard Monitoring	`OpenClipboard`, `GetClipboardData`
Form Grabbing	`HttpSendRequest`, `InternetWriteFile`, `WSASend`
Screen Logging	`BitBlt`, `PrintWindow`, `GetDC`
Mouse Tracking	`SetWindowsHookEx (WH_MOUSE_LL)`, `GetCursorPos`
Browser Script Injection	`onKeyDown`, `onKeyPress`, `onKeyUp` (JavaScript DOM APIs)
Raw Input Capture	`RegisterRawInputDevices`, `GetRawInputData`
Process Injection	`WriteProcessMemory`, `CreateRemoteThread`

C. API Hooking

API hooking is a widely used technique in system security and malware analysis for monitoring and modifying application behavior at runtime. It intercepts system or library calls to provide visibility and control, enabling sandboxes, antivirus tools, and debuggers to detect suspicious activity and enforce policies. Its dynamic, non-invasive nature supports deployment on live systems without kernel access. In cyber deception, API hooking enables real-time intervention as malware attempts to collect sensitive data. It is particularly effective against userland keyloggers that rely on APIs like `GetAsyncKeyState`, `SetWindowsHookEx`, and `ReadConsoleInput`. By intercepting these calls, defenders can inject false inputs, corrupt logs, or trigger early exfiltration. Unlike static deception, API-level deception enables dynamic, policy-driven interaction. However, advanced malware may use anti-hooking tactics to detect or disable hooks, requiring defensive deployments to emphasize stealth, integrity, and tamper resilience.

D. Motivation, Anti-Hooking, and Research Questions

A core challenge for hooking-based deception is resilience against anti-hooking techniques employed by advanced keyloggers. Modern malware can detect, bypass, or disable userland API hooks, threatening defensive frameworks built using prominent hooking libraries such as **EasyHook** [13] and **MS Detours** [14]. To ensure robust deception, our framework explicitly counters four widely observed anti-hooking strategies [15]–[18]: (i) .text section restoration to remove inline hooks, (ii) IAT rebinding to clean DLL copies, (iii) alternate DLL loading to evade instrumentation, and (iv) memory scanning to identify hook signatures. In these techniques, adversaries overwrite trampoline-modified code regions with clean instruction bytes, rebind imported function pointers in the IAT, load alternate DLLs (e.g., ntdll.dll), or scan memory for hook patterns to avoid instrumented functions. Such actions undermine any defense that assumes persistent control at the userland API boundary. Without effective tamper detection and mitigation, hooking-based deception may easily fail. Therefore, addressing these evasion strategies is essential for building an effective and resilient deception system. To guide the design and evaluation of our framework, we pose the following research questions:

- **RQ1:** Which API surfaces do userland keyloggers most frequently use, and what opportunities do they offer for deploying deception?
- **RQ2:** What anti-hooking techniques are employed by keyloggers to bypass userland monitoring, and how can a deception framework be designed to preserve its operational integrity in adversarial environments?

E. Threat Model

We consider user-mode keyloggers that operate with standard user privileges on a compromised system to capture sensitive inputs such as passwords or form data. These keyloggers may rely on common techniques including keyboard polling, input hooks, GUI message monitoring, clipboard access, or form grabbing. The attacker is assumed to be sophisticated, using anti-hooking tactics such as unhooking prologues, DLL shadow loading, dynamic API resolution, and memory scanning. Kernel-level and hardware keyloggers are out of scope. Our framework is not intended for malware detection. Instead, it becomes active after an external detection mechanism identifies a process as malicious. Such mechanisms may include an Endpoint Detection and Response (EDR) system such as [19], intervention by a human analyst or prior detection systems [20]–[22]. Once flagged, the deception module is injected into the process and operates at the same privilege level. The defender hooks input-related APIs, assuming OS and library integrity (e.g., EasyHook, Detours), and no interference from other endpoint security tools. The framework aims to **(1) prevent data leakage** by feeding keyloggers false inputs, **(2) preserve stealth** so attackers remain unaware of deception, and **(3) resist evasion and tampering** by detecting and recovering from common anti-hooking techniques to maintain the integrity of its instrumentation.

III. Framework Design

Our deception framework intercepts keylogger activity in real time, injects realistic but misleading inputs, and maintains its own integrity under attacker tampering. We build on EasyHook and Microsoft Detours—two widely used API hooking libraries—to achieve lightweight, runtime interception of relevant APIs. The architecture consists of three core components: a **Hooking Layer** that implants inline hooks at chosen input APIs, a **Deception Engine** that decides how to modify or fabricate data returned to the keylogger, and a **Hook Integrity Manager** that monitors and preserves the hooks against removal attempts. In essence, the Hooking Layer provides interception, the Deception Engine injects decoys, and the Integrity Manager ensures continued operation under adversarial conditions. Figure 1 provides the system architecture of the deception framework.

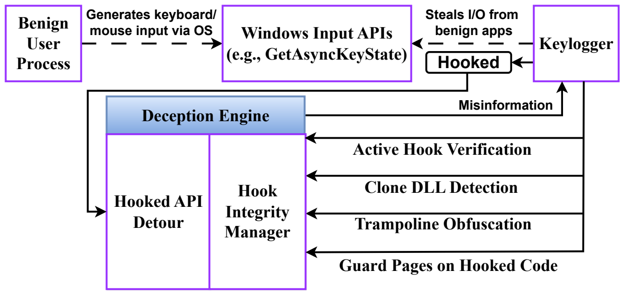

Fig. 1. System Architecture of the Deception Framework.

A. Target API Coverage

We focused on the Windows platform, as more than ninety percent of new malware samples target Windows systems [23]. Its mature and well-documented input APIs make it especially attractive to attackers. Our implementation uses both EasyHook and Microsoft Detours. While Microsoft Detours is tailored specifically for Windows, EasyHook is platform-agnostic. Our system can be adapted to other platforms with equivalent interception capabilities, once the relevant input APIs are identified for those environments.

A critical design consideration in building our deception framework was identifying which API calls to intercept to effectively undermine keyloggers. To identify these crucial APIs, we conducted a threat-driven study of fifty (50) keylogger samples drawn from ten widely observed keylogger families: Agent Tesla, AppleSeed, RokRAT, gh0st RAT, WarzoneRAT, Astaroth, Micropsia, NightClub, Azorult, and Lokibot. The samples were collected from publicly available malware repositories, including AnyRun and VirusTotal. For each family, we selected five representative binaries based on observed behavioral diversity and prevalence in recent campaigns. Each sample was dynamically analyzed using Cuckoo Sandbox, Rohitab API Monitor, and OllyDBG. We focused our efforts on the keylogging collection phase, as this stage reveals attacker intent most directly and offers a more tractable point for defensive intervention. Later stages such as file manipulation or network exfiltration often involve encryption or protocol obfuscation that makes deception less practical.

To identify the most relevant API calls used for gathering user input, we manually examined execution traces. Rather than relying on raw API call frequencies, we applied expert judgment to reconstruct the control flow and analyze the logic used by each malware sample to capture input. This allowed us to filter noise and highlight only those APIs that were functionally essential for keylogging. These categories, along with representative examples and associated keylogging strategies, are summarized in Table I. In particular, we concentrated on

APIs that (i) directly expose user input, (ii) allow indirect inference of keystrokes through GUI behavior, or (iii) facilitate passive capture of input data before exfiltration. We identified four key categories of target APIs:

Direct Input Acquisition APIs: Functions that immediately return keyboard state, which are popular for straightforward keylogging. For example, many malware use polling calls like `GetAsyncKeyState` or `GetKeyState` within loops to detect keystrokes with minimal latency. Our Hooking Layer intercepts these calls and returns manipulated key states or injected noise. For instance, when a keylogger polls these APIs, our system may respond with a stream of plausible but fake keystrokes, effectively corrupting the logged data at its source and rendering the captured information unreliable..

Window Message Inspection: APIs such as `GetMessage` and `PeekMessage` enable malware to monitor the GUI message queue and infer user keystrokes based on intercepted window messages. Our analysis shows that keyloggers targeting graphical applications, particularly browsers and email clients, frequently uses this technique to extract typed input. To counter this behavior, our framework intercepts message-level activity and introduces deception. When keylogging is detected within sensitive applications, our hooks can selectively suppress or manipulate `WM_KEYDOWN` and `WM_CHAR` messages. For example, the framework may inject fake key events or alter character messages to produce misleading input traces in the keylogger's log file.

Hook-Based Capture APIs: Some keyloggers rely on legitimate Windows APIs such as `SetWindowsHookEx` with the `WH_KEYBOARD_LL` flag to register global keyboard hooks and intercept keystrokes system-wide. Our framework counters this technique through two mechanisms. First, it can block untrusted processes from registering keyboard hooks by failing the hook installation or actively removing it. Second, if a hook is successfully registered, our system intercepts and overrides its callback logic. This allows us to prevent the keylogger from accessing real keystroke data and instead deliver fabricated input. These defenses are critical for mitigating threats that abuse legitimate OS-provided hooking capabilities.

Auxiliary Data Collection: In addition to capturing keystrokes directly, keyloggers often exploit supporting APIs to collect sensitive data or exfiltrate user input. Common examples include clipboard access via `OpenClipboard` and `GetClipboardData`, or intercepting form submissions using network functions like `HttpSendRequest` and `WSASend`. Our Hooking Layer monitors these APIs as well, allowing the Deception Engine to inject plausible decoy content. For instance, fake clipboard data can be returned to mislead malware accessing `GetClipboardData`, and fabricated credentials can be inserted into outbound transmissions. Since these APIs are typically more context-sensitive, hooks are selectively applied to preserve application stability while disrupting form-grabbing behavior. This categorization guided our API selection and prioritization: lower-level functions such as polling or clipboard access allow simple, low-overhead interception; GUI and hook-based methods provide richer context but require careful handling; and network-layer APIs, while demanding deeper inspection, offer strategic points to corrupt attacker objectives. Our framework balances these trade-offs, achieving broad coverage and effective deception with minimal performance overhead, as demonstrated in our evaluations.

B. Input Deception Policies

Intercepting an API call is the initial step. The core functionality of the Deception Engine lies in determining how to respond to those intercepted calls. Since our framework is deployed only after a process has been externally flagged as suspicious, such as by an analyst, Endpoint Detection and Response (EDR) tool, or anti-malware system, it operates in a strictly **active mode**, applying deception policies immediately and consistently. This design avoids the need for runtime decision-making, behavioral triggers, or passive logging within the framework itself.

Decoy Injection: The hook returns synthetic data in place of the real input. This straightforward strategy directly feeds false information to the keylogger. For instance, imagine malware polling GetAsyncKeyState to capture a user's password as they type. Under a Decoy Injection policy, when the user presses "P", the keylogger might instead receive a code corresponding to, say, "M", followed by a sequence of characters spelling out a fake password. In our prototype, we scripted decoy payloads such as "Password123!" or realistic keyboard patterns to simulate user typing. If the attacker is exfiltrating keystrokes in real time or operating in an automated fashion, this tactic poisons their data entirely. We found Decoy Injection particularly effective against rapid key thieves, as it fills their logs with plausible-looking but incorrect credentials.

Input Perturbation: Rather than full fabrication, the engine makes slight, often random modifications to the legitimate input data. The goal is to degrade the fidelity of stolen data while remaining subtle. Our framework can, for example, flip the case of every third character the user types, introduce small delays in reporting keystrokes, or inject an occasional extra keystroke that would be benign (like an innocuous typo) from the user's perspective. This way, the attacker's logged keystroke sequences will not exactly match what the user actually entered. In one test scenario, we enabled a perturbation policy that would randomly swap two characters in a password being captured. The genuine login still succeeded (because the real keystrokes went through to the application unaltered), but the keylogger's record contained the wrong characters, thwarting any later misuse of that password.

Input modification is configurable at deployment via a masking ratio (e.g., 30% or 100%), enabling partial perturbation of the keystroke buffer through randomized index selection. This reduces overhead while maintaining plausible deception, especially valuable under high-frequency polling, where full modification may introduce latency. To ensure efficiency, the system adopts a context-agnostic strategy that avoids semantic input analysis. While context-aware deception (e.g., leveraging NLP or LLMs) is feasible, it introduces additional complexity and cost, hence is left as future work.

C. Hook Integrity and Tamper Resilience

While the Hooking Layer and Deception Engine mislead attackers, an advanced adversary may attempt to disable the framework by tampering the hooks. Hence, we prioritized hook integrity from the start, guided by real attacker behaviors. The Hook Integrity Manager implements layered defenses to detect and counter such interference. Hook Integrity Manager uses memory protection attributes such as `PAGE_NOACCESS` or `PAGE_GUARD` to safeguard its internal data and control logic. It executes as a privileged watchdog thread outside common scanning regions, reducing its exposure to tampering. These protections help isolate the Hook Integrity Manager from untrusted threads while maintaining operational transparency.

Active Hook Verification: EasyHook does not provide built-in support for verifying the integrity of installed hooks. To address this limitation, our framework implements custom logic to detect and recover from hook tampering in real time. After installing a hook using `LhInstallHook`, we capture a lightweight hash or byte-level snapshot of the target function's prologue. A dedicated watchdog thread periodically compares the current memory content of each hook target with the expected detour layout. If an attacker attempts to unhook (e.g., restoring the original function bytes via `memcpy` or `WriteProcessMemory`), the tampering is detected and the Hook Integrity Manager immediately restores the hook. This self-healing process preserves the continuity and stealth of the deception engine. In our evaluation, tampering was typically corrected within milliseconds and did not introduce measurable performance degradation.

Clone DLL Detection: Some advanced malware attempt to evade userland hooks by loading fresh copies of common libraries such as `user32.dll` or `ntdll.dll` under alternate names or paths. By calling functions from these clean instances, the malware can bypass previously installed hooks. While effective against static hooking, our framework extends EasyHook with custom logic to detect and counter such evasion. Since EasyHook does not monitor module load events by default, we hook low-level loader functions, such as `LdrLoadDll`, within the target process. When a new module is loaded, our hook inspects its export table and code sections to identify clones of known system libraries. If matched, we dynamically apply the same set of hooks to the new module using `LhInstallHook` API. This clone-aware mechanism ensures continued interception even when malware attempts to establish an unmonitored API path through DLL shadowing. In our evaluation, this method successfully detected and re-hooked a dynamically loaded clean copy of `user32.dll`, neutralizing the bypass attempt in real time.

Trampoline Obfuscation: By default, EasyHook installs inline detours using common jump instructions such as `JMP rel32` (`E9`) or indirect jumps via pointers (e.g., `FF25`). These opcode patterns are widely recognized and easily identified through memory scans that target the prologues of commonly hooked functions. To enhance stealth, we implemented a custom trampoline generator that avoids these recognizable signatures. Our obfuscation logic constructs semantically equivalent but structurally different sequences, including `PUSH address; RET` and `MOV reg, address; JMP reg`. We further randomize the detour memory layout and insert benign padding instructions to disrupt signature-based scanning. These transformations are applied dynamically during hook installation, yielding a distinct hook fingerprint for each deployment. In evaluation, our obfuscation consistently evaded detection by simulated attacker tools that relied on signature matching. While this functionality extends beyond EasyHook's native features, it significantly increases resistance to static memory analysis and reverse engineering.

Guard Pages on Hooked Code: As an additional layer of tamper detection, we implemented page-level memory protections to secure the code regions of hooked functions. After installing a hook, our framework modifies the memory protection of the corresponding page using `PAGE_GUARD` or `PAGE_NOACCESS`, depending on operating system support. Any unauthorized read or write access to this memory, such as an attempt to inspect or modify the function prologue, will trigger an access violation. The framework registers a structured exception handler (SEH) that intercepts these violations, verifies the integrity of the hook, and immediately restores it if tampering is detected. In stricter configurations, the attacking thread or process is terminated to prevent further manipulation attempts. This method effectively turns the hooked memory region into a tripwire, enabling real-time detection and response. In our controlled evaluation, this method successfully raised an `access violation` when a simulated keylogger attempted to overwrite the prologue of `GetAsyncKeyState`. The triggered exception allowed our framework to detect and restore the hook before any further bypass activity could occur. We note that this guard-based approach is not universally applicable, as some system DLL or shared memory regions may not reliably support guarded permissions due to OS level constraints. Nonetheless, in applicable scenarios, it provided a strong layer of hook resilience with minimal overhead. Table II summarizes the defensive measures implemented by the Hook Integrity Manager. By integrating these defenses, the framework maintains resilience against tampering. The Hook Integrity Manager converts anti-hooking behaviors into observable events, allowing timely recovery with negligible performance overhead.

IV. Implementation and Evaluations

A. Prototype Implementation

We tested our deception framework on virtualized Windows 10 and Windows 11 environments. Our deception framework was manually injected into target processes using EasyHook's remote injection API. We tested it against both open-source and real-world malware samples. Evaluation metrics included CPU overhead, success of decoy injection, and resilience to tampering. Hook integrity was verified using memory hash

TABLE II
DEFENSIVE MEASURES IMPLEMENTED BY THE HOOK INTEGRITY MANAGER

Defense Mechanism	Description	Tamper Detection Method	Reaction
Active Hook Verification	Detects tampering by periodically comparing current memory at each hook site with the expected detour layout. Reinstalls hooks if tampering is detected.	Memory comparison with expected detour layout.	Reapply detour patch immediately.
Clone DLL Detection	Hooks module loader to detect duplicate DLLs (e.g., user32.dll). Applies the same set of hooks to newly loaded modules if a match is found.	Analysis of module characteristics (e.g., export table signatures or code sections) on load.	Apply hooks on duplicate modules
Trampoline Obfuscation	Replaces standard JMP instructions with alternate sequences to avoid signature recognition via memory scanning.	Avoidance of common jump signatures.	Avoid detection from memory scanners.
Guard Pages on Hooked Code	Modifies memory protection of target function's page using PAGE_GUARD or PAGE_NOACCESS to catch tampering via access violations.	Access violation triggers via structured exception handler (SEH).	Intercept and restore hook; Optionally terminate thread.

comparisons and structured exception handler (SEH)-based validation. To validate the design of our deception framework, we developed two functionally equivalent prototypes using EasyHook and Microsoft Detours. Our goal was to determine whether the core ideas: the hook-based interception, decoy injection, and tamper resilience could be implemented effectively across different hooking libraries, each with their own engineering trade-offs.

The EasyHook-based prototype provided a high-level API, support for both 32-bit and 64-bit processes, and native .NET integration. These features enabled rapid development and iterative testing with minimal low-level engineering. In contrast, the Detours-based implementation offered fine-grained control over instruction patching. We extended it with custom logic for randomized trampoline generation and runtime hook validation to enhance stealth and robustness. While this version required more manual engineering effort, particularly for handling edge cases in 64-bit systems, it validated the generality and portability of our framework design. For clarity and consistency, our evaluation focuses on the EasyHook-based prototype, which served as the basis for case studies and performance testing. The complete source code for both variants is publicly available at [10].

B. Case Study: Deceiving WarzoneRAT, a Real-World Polling-Based Keylogger

We deployed our framework against WarzoneRAT (a.k.a. Ave Maria RAT) [24], a widely documented remote access trojan with a built-in keylogger. We selected WarzoneRAT as a case study because it represents a common but effective class of keyloggers. Its polling-based approach, though simple, generates high API call volume, making it ideal for testing the stability, performance, and transparency of our framework under realistic conditions. As mentioned it's keylogging relies on a stealthy polling approach: it repeatedly calls the Windows API GetAsyncKeyState to check the state of each key, rather than installing a hook via the OS. This design makes it stealthier in some respects and is fairly typical of many simple yet effective keyloggers in the wild.

Attacker's Technique: WarzoneRAT's keylogger uses a polling loop that repeatedly calls `GetAsyncKeyState` across virtual key codes (0x08–0xFE) to detect pressed keys. It supplements this with calls to `GetForegroundWindow`, `GetWindowTextW`, and `GetKeyboardState` to capture the active window title and modifier keys (e.g., Shift, CapsLock), allowing it to reconstruct accurate, timestamped keystroke logs tagged by application context. These logs are periodically written to disk, building a detailed timeline of user activity such as: "[Browser – 10:30AM] username: Alice" followed by "[Browser – 10:30AM] password: secret123".

Deception Deployment: We evaluated our framework by injecting a hook-enabled DLL into the WarzoneRAT process at runtime. For this experiment, we used the EasyHook-based implementation. Our focus was on subverting WarzoneRAT's polling-based keylogger, which repeatedly invokes `GetAsyncKeyState` to collect keystrokes. We installed an inline hook on `GetAsyncKeyState` and redirected all invocations to a custom detour handler governed by our *Decoy Injection* policy. Rather than attempting full behavioral emulation or adaptive input generation, our approach focuses on direct substitution of keystroke content with carefully crafted, plausibly structured decoy sequences. During evaluation, the detour handler replaced the output of `GetAsyncKeyState` with a predefined but realistic-looking credential string—such as `"hrSmith2025!"`—designed to mimic user input patterns commonly observed in authentication workflows. The injection sequence also included synthetic tab and enter keystrokes to simulate form navigation and submission. This substitution-based deception simplifies runtime complexity while still producing high-fidelity artifacts that closely resemble genuine user behavior from the attacker's perspective.

Results and Impact: During the evaluation, our framework successfully deceived WarzoneRAT without disrupting its execution or revealing the presence of instrumentation. The malware continued to invoke `GetAsyncKeyState` at its normal polling frequency, unaware that our hook was intercepting the calls. When triggered during a simulated login scenario, the detour returned a predefined but fake decoy sequence (e.g., `"hrSmith2025!"`), including synthetic `TAB` and `ENTER` keystrokes to mimic form submission. The actual user input remained fully hidden. The resulting keylog file (`keylog.log`) contained only the decoy data, complete with accurate window titles and timestamps, producing output that appeared indistinguishable from genuine user behavior. The malware's auxiliary functions—such as modifier key checks and file I/O—proceeded normally, and no detection logic was triggered. To measure runtime performance impact, we used the Windows Performance Monitor (`perfmon`) to track CPU utilization of the instrumented process during high-frequency polling sessions. Simulated user interactions were performed repeatedly over a 60-second interval, with the keylogger running under normal conditions. Across five independent runs, the measured CPU overhead introduced by our hooking and detour logic remained consistently below one percent. The host system remained fully responsive, and no input lag or application slowdown was observed during these tests.

Overhead Across Deception Policies: We also examined the runtime overhead introduced by our two deception policies: Decoy Injection and Input Perturbation. For static content theft (e.g., clipboard or credential files), Decoy Injection imposes negligible overhead, as it substitutes predefined content without parsing. In contrast, Input Perturbation requires runtime processing to apply masking ratios, increasing computational cost. In polling-based keyloggers like WarzoneRAT, which calls `GetAsyncKeyState` approximately 131 times per minute, full Decoy Injection requires modifying all 131 calls. However, applying a 20 percent perturbation ratio reduces this to only 26 modified responses, lowering CPU load while still corrupting the keylog. This tradeoff allows our framework to balance deception fidelity with performance efficiency based on the threat model.

C. Broader Evaluation Against Real-World Malware

To generalize beyond a single malware instance, we tested our framework against fifty keylogger samples as mentioned in Section III-A. During our analysis, we observed that several malware families including Agent Tesla, Lokibot, AppleSeed, Azorult, and Astaroth employed basic sandbox and virtualization evasion techniques. These behaviors involved checks for the presence of debuggers, timing anomalies, or signs of an analysis environment using APIs such as `IsDebuggerPresent`, `GetTickCount`, and `NtQueryInformationProcess`. When successful, these checks would cause the malware to suppress its payload execution or terminate early in order to avoid analysis or detection. To prevent these evasive techniques from interfering with full execution, we extended our hooking layer to intercept the relevant evasion-related APIs. Our detour logic returned spoofed values that emulated a normal, non-instrumented environment. For instance, debugger detection APIs were forced to return false, and timing outputs were adjusted to eliminate inconsistencies. As a result, the malware continued its execution path and reached the keylogging stage without detecting the instrumentation.

For each sample, we confirmed the malware attempted to invoke one or more input collection APIs listed in Table I, and verified our framework intercepted them correctly. In all fifty cases, the Hooking Layer captured the targeted API calls and redirected them to the Deception Engine, which returned either crafted decoy data or perturbed input. The malware executed normally, recorded fake inputs, and never logged real user data. None of the samples triggered crashes or evasion logic in response to our instrumentation. This evaluation confirms that our framework reliably deceives a broad range of keylogger families, even when facing common evasion techniques. We next evaluate its resilience against active tampering and anti-hooking defenses.

D. Case Studies: Against Anti-Hooking Attacks

To rigorously evaluate our framework's resilience, we developed a custom "super keylogger" that combines multiple input-logging methods with advanced anti-hooking techniques. This case study simulates a sophisticated attacker using diverse strategies while actively evading defensive hooks. Building it ourselves gave us full control to observe the interplay between attack and defense. The process also drew on real-world malware behaviors and AI-assisted development to ensure it was both comprehensive and realistic.

Building a Multifaceted Keylogger: We used an LLM (GPT-4o) to assist in generating and refining keylogger components, which accelerated the implementation of multiple logging methods and ensured comprehensive coverage. Through iterative prompts (e.g., "generate code to log keys via GetAsyncKeyState," "what other Windows APIs capture keystrokes?"), we built a modular keylogger integrating five distinct data-capture techniques: (1) direct polling via GetAsyncKeyState and GetKeyState, (2) event hooking through SetWindowsHookEx, (3) GUI message spying using functions like PeekMessage, (4) clipboard monitoring via OpenClipboard and GetClipboardData, and (5) network interception of HTTP POSTs by hooking HttpSendRequest and WSASend, simulating form grabbing. These methods reflect the breadth of userland keylogging seen in real malware.

We implemented each method in its own thread, allowing the keylogger to simultaneously poll keys, monitor clipboard activity, and intercept network traffic—maximizing redundancy and data capture. In tests on Windows 10 and 11, the keylogger reliably recorded all expected input, including clipboard changes and mock web submissions, even producing overlapping logs across techniques. Functionally, it behaved like a super keyloggers—merging traditional and advanced techniques into a single, potent data-theft tool. This confirmed that our deception targets were up against a realistic and comprehensive threat. All prompts, source code, and modular implementations used to construct this keylogger are publicly available in our supplementary materials [10].

Integrating Anti-Hooking Techniques and Evaluating Framework Resilience: Next, we equipped the keylogger with four anti-hooking strategies [15]–[18] to simulate an attacker actively trying to evade hook-based defenses like ours. One technique was inspired by the real-world malware SmokeLoader [15], known for its aggressive anti-hooking behavior. Another was derived from academic literature, reflecting techniques commonly discussed in research [18]. The remaining two [16], [17] were adapted from open-source implementations available on GitHub, mimicking what a skilled attacker could easily integrate. Together, these made the keylogger an actively evasive threat capable of challenging advanced hook defenses.

1) Runtime Unhooking via .text Restoration: This technique involves the keylogger actively removing inline hooks by restoring the original bytes of targeted API functions in memory [15]. For example, it marks the memory of `GetAsyncKeyState` as writable using `VirtualProtect`, then restores the original bytes—either previously saved before the hook was installed or retrieved from a clean copy of the DLL. If our framework had patched a `JMP` instruction at the beginning of `GetAsyncKeyState`, the malware would overwrite it with the original opcodes. This is a direct assault on inline hooks; if successful, control flow no longer jumps to our detour and instead executes the API normally. We have observed real-world malware using this tactic to disable userland security monitors.

Outcome: We anticipated this to be a substantial challenge, as it targets the core of our Hooking Layer. However, the Hook Integrity Manager's active verification loop responded almost instantly. As soon as the keylogger overwrote the first few bytes, the watchdog thread's subsequent hash check flagged the discrepancy. Our framework immediately re-patched the hook in the function's prologue. From the keylogger's perspective, it had successfully unhooked the API and proceeded without further checks, unaware that within milliseconds, the hook was reinstated. Since the keylogger didn't continuously verify the hook status, it assumed success and continued calling `GetAsyncKeyState`, all while our detour fed it synthetic keystrokes. We confirmed this by inspecting its logs—only decoy inputs such as `"Admin123!"` were recorded post-unhooking. The keylogger remained unaware of the deception, validating our rapid detection and self-healing hook strategy as a defense against memory patching attacks.

2) Import Address Table (IAT) Rebinding to Clean DLLs: In this technique, the keylogger circumvents hooked APIs by loading a clean copy of a system DLL [17]. Using `LoadLibraryEx` with specific flags, it loads a fresh instance of `user32.dll` into a different memory location. This new instance, absent during our original hook injection, contains unpatched functions. The malware then uses `GetProcAddress` to resolve function addresses (e.g., `GetAsyncKeyState`) and updates its Import Address Table or local function pointers to reference these clean copies. This enables the malware to call APIs from the unmonitored DLL, bypassing our instrumentation.

Outcome: Our framework countered this using the Hook Integrity Manager's clone-detection. Since we had hooks placed on Windows loader routines, any new DLL load event triggered inspection. Upon identifying the newly loaded module as a clone of `user32.dll`, the framework scanned its export table, injected the same hooks, and rewired malware's function pointers to route through our detours. Consequently, while the malware believed it had sidestepped our monitoring, we had already hooked the new entry points. The keylogger resolved and invoked what it assumed was an unhooked function, but execution flowed straight into our deception logic. Again, the logged keystrokes were decoys. This emphasized the importance of real-time module load monitoring. While it introduces some complexity, our experiments showed the added hooks incurred minimal overhead and ensured effective coverage.

3) Manual API Resolution (Alternate DLL Loading without IAT): An evolution of IAT rebinding, this method avoids import tables altogether. The malware manually resolves addresses of critical APIs at runtime and invokes them directly through function pointers [18]. For instance, it loads a clean copy of `ntdll.dll` or `user32.dll`, uses `GetProcAddress` to resolve `GetAsyncKeyState`, and stores the resulting address. It then calls the function via `(*funcPtr)()`, bypassing static imports and any hooks placed there.

Outcome: We generalized our clone-module defense by maintaining a registry of detoured functions by name and module base address. When a new DLL is loaded, we hook all matching exports. As a result, the function pointers resolved by the malware point to already-detoured functions. Since manual resolution eliminates future interception opportunities, our response must occur during module load. We optimized our loader hook to apply detours preemptively. We also ensured our trampolines were either position-independent or appropriately adjusted for each base address. During tests, the malware resolved `GetAsyncKeyState`, stored the address, and called it—unaware that the address already led to our handler. No anomalies were detected by the keylogger, confirming the efficacy of our proactive strategy. While this approach requires hooking multiple module instances, our modular architecture reused detour logic efficiently, maintaining low overhead.

4) In-Memory Signature Scanning for Hooks: In this defensive maneuver, the keylogger scans memory to detect the presence of inline hooks [16]. Many malware strains implement such scanning routines, checking whether the first few bytes of sensitive APIs match known patterns or appear tampered. Our keylogger version scanned APIs like `GetAsyncKeyState` and `SetWindowsHookEx` for recognizable signatures, such as near jumps `(E9)` or indirect jumps. Upon detecting a potential hook, it either avoided calling the function or logged a warning, thereby reducing the chance of interacting with a monitored or manipulated function.

Outcome: Our defense relied on stealth-by-diversity, implemented through trampoline obfuscation. Our detours avoided conventional byte patterns. For example, a hook stub might begin with a `PUSH` followed by a `RET`, deviating from typical `JMP` signatures. We also introduced random padding and varied hook placement through address layout randomization. During experiments, the keylogger ran its scan before logging began. It failed to detect our hooks and proceeded to log activity—entirely unaware it was capturing only fake keystrokes. This reinforced the importance of remaining covert in deception-based defenses. Once the adversary suspects tampering, they may disengage or adapt. Our obfuscated hooks maintained the attacker's trust in the environment, ensuring continued operation under false pretenses.

We evaluated our framework against a custom "Super" keylogger implementing .text restoration, IAT rebinding, manual API resolution, and in memory hook detection. Despite efforts to remove hooks, load clean DLLs, resolve APIs manually, and detect in-memory modifications, our framework neutralized each evasion method in real time. The keylogger was limited to logging only decoy inputs, with real user activity unaffected. This test confirmed the resilience of our combined use of inline hooking, integrity validation, and deception.

V. Related Work

To the best of our knowledge, this is the first system to combine API hooking for deception with integrated defenses against anti-hooking techniques. Existing deception frameworks often fail under tampering, while anti-hooking solutions generally lack deceptive capabilities. As these approaches address distinct aspects of the problem space, direct empirical comparison is infeasible. Our work bridges this gap through a resilient design.

Deception has long been a strategic defense in cybersecurity, with early methods relying on honeypots, honeytokens, and honeywords to mislead adversaries and collect telemetry [25]–[27]. These mechanisms have since evolved into dynamic frameworks that leverage moving target defense [28], Markov decision processes [29], and game-theoretic planning [30] to adapt deception strategies in real time. Systems such as PhantomFS [31] and others [32]–[34] offer scalable platforms for orchestrating and evaluating deception assets.

In the context of keyloggers, deception has been explored through the injection of false input to mislead or expose surveillance software. Simms et al. [11] proposed a decoy keyboard mechanism to confuse keyloggers, while Wazid et al. [12] developed a honeypot-based framework that simulates user interaction to trigger malicious behavior. These works focus on misdirection without addressing the advanced evasion capabilities. Complementing deception, a substantial body of work focuses on detection. Nasaka et al. [1] and Al-Husainy [2] monitored API usage to flag suspicious behavior. HookTracer [3] performed forensic memory analysis to detect hook artifacts, while Muthumanickam et al. [4] targeted inline and IAT hook tampering in userland processes. However, these approaches also lack defenses against anti-hooking techniques.

More sophisticated adversaries, such as those behind SmokeLoader [15], actively dismantle or bypass defensive instrumentation. SmokeLoader, for instance, restores original function prologues to remove inline hooks. Other public implementations demonstrate evasive tactics including Import Address Table rebinding and alternate DLL loading [17]. In-memory hook detection using signature scans, as seen in various malware samples [16], further complicates defensive visibility. These strategies highlight a growing asymmetry: while detection improves, malware continues to evolve to evade naive userland monitoring hookings entirely.

VI. Discussion and Conclusion

Effectiveness and Resilience: Our experience shows that hooking-based deception is both feasible and effective for countering userland keyloggers. In all test scenarios, keyloggers operated under the illusion of capturing real input, while our framework injected fabricated data. For example, when deployed against WarzoneRAT, a real-world RAT with polling-based keylogging, our hook on GetAsyncKeyState redirected keystroke queries to our deception engine, which returned realistic decoys. These were logged by the malware with correct context and timestamps, keeping actual user input secure. User experience was unaffected and system overhead remained negligible as presented in IV-B.

To evaluate resilience under a stronger adversarial model, we developed a custom super keylogger combining behaviors from diverse malware families. Guided by malware reports, academic literature, and LLM-assisted synthesis, it simulates diverse keylogging strategies without using multiple samples. The keylogger integrates five input-capture techniques and incorporates four advanced evasion methods: one drawn from real-world malware, one from an academic publication, and two from widely used open-source tools. These include .text section restoration, IAT rebinding, alternate DLL loading, and signature-based memory scanning, as detailed in Section IV-D. Despite full knowledge of our defense mechanisms, the adversary was unable to bypass the system. The Hook Integrity Manager mitigated each evasion attempt through proactive hook verification, clone DLL tracking, and randomized trampoline construction. These results validate the soundness and resilience of hooking our strategy. The combination of runtime input randomization and multi-layer tamper resistance significantly increases the complexity of static and behavioral evasion. Successful circumvention would require memory inspection or runtime probing, which contradicts keylogging stealth and proved ineffective in our evaluation.

Limitations and Challenges: Despite promising results, several limitations emerged. First, our framework operates solely in user space, excluding kernel-level and hardware keyloggers, which bypass the API layer. Extending deception to lower layers would require OS or driver-level support, which we leave for future work. Second, our framework assumes a prior detection event. It is not a standalone detector; rather, it activates once a process is flagged as suspicious by a security analyst or detection tool. In our evaluations, we manually injected the module, but in real deployments, this would need to be automated and tightly integrated with existing detection pipelines. Finally, hooking in adversarial settings introduces engineering complexity. Though EasyHook and MS Detours ease initial development, they lack tamper resilience. We added trampoline obfuscation, clone-module tracking, and memory protections, requiring deep understanding of the Windows memory model and low-level tuning to ensure stability and performance.

Lessons for Practitioners: From this experience, several key lessons emerge for practitioners building deception systems. First, designing for evasion from the start is essential. Hook integrity must be treated as a first-class concern, as adversaries will attempt to tamper with instrumentation. Our framework's success depended on early integration of tamper detection and restoration. Second, deception only works if the attacker believes it. Static or naive decoys risk exposure. In our framework, plausible fake inputs (e.g., passwords with tab/enter) increased believability. Adding variability or randomization in future versions may further enhance stealth. Third, anti-hooking itself can serve as a detection signal. Our design demonstrates that tampering attempts offer forensic value, and deception frameworks can double as sensors for identifying sophisticated threats. Finally, the balance between stealth and response must be carefully considered. We prioritized transparency by not killing threads or processes even during tampering. However, a higher-assurance environment may require stronger countermeasures. Policy-driven flexibility allows tailored responses per threat context.

In conclusion, this work demonstrates that deception via API hooking is a practical and resilient defense against userland keyloggers. By intercepting and manipulating attacker-visible API calls, and protecting hook integrity, we misled malware without affecting user experience. Our findings show adversary-aware instrumentation and stealthy deception form a lightweight, effective alternative to traditional detection and blocking. In future work, we plan to integrate the framework with detection pipelines for automatic engagement and extend our techniques to other attack vectors.

References

[1] K. Nasaka, T. Takami, T. Yamamoto, and M. Nishigaki, "A keystroke logger detection using keyboard-input-related api monitoring," in *2011 14th International Conference on Network-Based Information Systems*. IEEE, 2011, pp. 651–656.

[2] Y. Al-Hammadi and U. Aickelin, "Detecting bots based on keylogging activities," in *2008 Third International Conference on Availability, Reliability and Security*. IEEE, 2008, pp. 896–902.

[3] A. Case, R. D. Maggio, M. Firoz-Ul-Amin, M. M. Jalalzai, A. Ali-Gombe, M. Sun, and G. G. Richard III, "Hooktracer: Automatic detection and analysis of keystroke loggers using memory forensics," *Computers & Security*, vol. 96, p. 101872, 2020.

[4] K. Muthumanickam and E. Ilavarasan, "An effective method for protecting native api hook attacks in user-mode," *Research Journal of Applied Sciences, Engineering and Technology*, vol. 9, no. 1, pp. 33–39, 2015.

[5] M. S. I. Sajid, J. Wei, B. Abdeen, E. Al-Shaer, M. M. Islam, W. Diong, and L. Khan, "Soda: A system for cyber deception orchestration and automation," in *Proceedings of the 37th Annual Computer Security Applications Conference*, 2021, pp. 675–689.

[6] M. S. I. Sajid, J. Wei, E. Al-Shaer, Q. Duan, B. Abdeen, and L. Khan, "Symbsoda: configurable and verifiable orchestration automation for active malware deception," *ACM Transactions on Privacy and Security*.

[7] M. M. Islam, A. Dutta, M. S. I. Sajid, E. Al-Shaer, J. Wei, and S. Farhang, "Chimera: Autonomous planning and orchestration for malware deception," in *2021 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2021, pp. 173–181.

[8] S. Ahmed, A. M. Rahman, M. M. Alam, and M. S. I. Sajid, "Spade: Enhancing adaptive cyber deception strategies with generative ai and structured prompt engineering," in *2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2025.

[9] M. S. I. Sajid, J. Wei, and E. Al-Shaer, "randecepter: Real-time identification and deterrence of ransomware attacks," in *2025 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2025, pp. 1–9.

[10] sahmed09, "Deception keylogger: Supplementary materials," https://github.com/sahmed09/HookBasedDeceptionAgainstKeylogging, 2025, accessed: 2025.

[11] S. Simms, M. Maxwell, S. Johnson, and J. Rrushi, "Keylogger detection using a decoy keyboard," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2017.

[12] M. Wazid, A. Katal, R. Goudar, D. Singh, A. Tyagi, R. Sharma, and P. Bhakuni, "A framework for detection and prevention of novel keylogger spyware attacks," in *2013 7th International Conference on Intelligent Systems and Control (ISCO)*. IEEE, 2013, pp. 433–438.

[13] Easyhook, "Easyhook: The reinvention of windows api hooking," https://easyhook.github.io/, 2015, accessed: 2025-05-20.

[14] Microsoft, "Detours: A software package for monitoring and instrumenting api calls on windows," https://github.com/microsoft/Detours, 2016, accessed: 2025-05-20.

[15] Malware and Stuff, "Examining smokeloader's anti-hooking technique," https://malwareandstuff.com/examining-smokeloaders-anti-hooking-technique/, 2020, accessed: 2025-05-20.

[16] dtm, "Defeating userland hooks (ft. bitdefender)," https://0x00sec.org/t/defeating-userland-hooks-ft-bitdefender/12496, 2019, accessed: 2025-05-20.

[17] N. Cano, "Reloadlibrary: A quick-and-dirty anti-hook library proof of concept," https://github.com/nickcano/ReloadLibrary, 2019, accessed: 2025-05-20.

[18] T. Apostolopoulos, V. Katos, K.-K. R. Choo, and C. Patsakis, "Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks," *Future Generation Computer Systems*, vol. 116, pp. 393–405, 2021.

[19] A. Arfeen, S. Ahmed, M. A. Khan, and S. F. A. Jafri, "Endpoint detection & response: A malware identification solution," in *international conference on cyber warfare and security (ICCWS)*. IEEE, 2021.

[20] S. Ortolani, C. Giuffrida, and B. Crispo, "Bait your hook: a novel detection technique for keyloggers," in *International workshop on recent advances in intrusion detection*. Springer, 2010, pp. 198–217.

[21] M. B. Ahmed, M. Shoikot, J. Hossain, and A. Rahman, "Key logger detection using memory forensic and network monitoring," *International Journal of Computer Applications*, vol. 975, p. 8887, 2019.

[22] D. Pillai and I. Siddavatam, "A modified framework to detect keyloggers using machine learning algorithm," *International Journal of Information Technology*, vol. 11, no. 4, pp. 707–712, 2019.

[23] AV-TEST Institute, "Malware statistics & trends report," https://www.av-test.org/en/statistics/malware/, 2025, accessed: 2025-05-20.

[24] Splunk Threat Research Team, "Defending the gates: Understanding and detecting ave maria (warzone) rat," https://www.splunk.com/en_us/blog/security/defending-the-gates-understanding-and-detecting-ave-maria-warzone-rat.html, 2023, accessed: 2025-05-20.

[25] S. Kyung, W. Han, N. Tiwari, V. H. Dixit, L. Srinivas, Z. Zhao, A. Doupé, and G.-J. Ahn, "Honeyproxy: Design and implementation of next-generation honeynet via sdn," in *2017 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2017, pp. 1–9.

[26] M. L. Godakanda and D. M. Cook, "Honey in depth: A multi-layered deception framework for internet of things security," in *2024 7th International Conference on Information and Communications Technology (ICOIACT)*. IEEE, 2024, pp. 119–124.

[27] A. R. Petrunić, "Honeytokens as active defense," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2015.

[28] G. Pagnotta, F. De Gaspari, D. Hitaj, M. Andreolini, M. Colajanni, and L. V. Mancini, "Dolos: A novel architecture for moving target defense," *IEEE Transactions on Information Forensics and Security*, 2023.

[29] O. Hayatle, H. Otrok, and A. Youssef, "A markov decision process model for high interaction honeypots," *Information Security Journal: A Global Perspective*, vol. 22, no. 4, pp. 159–170, 2013.

[30] A. H. Anwar, C. A. Kamhoua, N. O. Leslie, and C. Kiekintveld, "Honeypot allocation for cyber deception under uncertainty," *IEEE Transactions on Network and Service Management*, 2022.

[31] J. Lee, J. Choi, G. Lee, S.-W. Shim, and T. Kim, "Phantomfs: File-based deception technology for thwarting malicious users," *IEEE Access*, vol. 8, pp. 32 203–32 214, 2020.

[32] J. H. Jafarian and A. Niakanlahiji, "A deception planning framework for cyber defense." in *HICSS*, 2020, pp. 1–10.

[33] Q. Duan, E. Al-Shaer, M. Islam, and H. Jafarian, "Conceal: A strategy composition for resilient cyber deception-framework, metrics and deployment," in *2018 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2018, pp. 1–9.

[34] M. M. Islam, E. Al-Shaer, and M. A. B. U. Rahim, "Email address mutation for proactive deterrence against lateral spear-phishing attacks," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2020.