LinkGuard: A Lightweight State-Aware Runtime Guard Against Link Following Attacks in Windows File System

Bocheng Xiang, Yuan Zhang, Hao Huang, Fengyu Liu, Youkun Shi
Fudan University, China
{*bcxiang23, huangh21, fengyuliu23, 21110240048*}*@m.fudan.edu.cn, yuanxzhang@fudan.edu.cn*

***Abstract*—Link Following (LF) attacks in the Windows file system allow adversaries to stealthily redirect benign file operations to protected files by abusing crafted combinations of symbolic links (link chains), thereby enabling arbitrary manipulation of protected files. Such attacks typically manifest as either single-step attacks or multi-step attacks, depending on the sequencing of the constructed link chain. Existing countermeasures against LF attacks either rely on heavyweight modeling or suffer from poor compatibility and limited applicability, and none provide comprehensive protection across different types of LF attacks.**

In this paper, we present *LinkGuard*, a lightweight state-aware runtime guard against LF attacks targeting Windows systems. The novelty of *LinkGuard* lies in its two-stage design: The first stage aims to improve defense efficiency by performing dynamic subject filtering, which monitors only file operations and associated subjects involved in the creation and following of link chains; The second stage applies FSM-based rule matching to precisely defend LF attacks, ensuring effective and accurate defense. We evaluate *LinkGuard*'s prototype across five representative Windows systems to validate its compatibility. On a dataset of 70 real-world vulnerabilities, *LinkGuard* successfully mitigates all single-step attacks and 95.45% of multi-step attacks, with zero false positives on benign operations. On average, *LinkGuard* only incurs 1% overhead in microbenchmarks and 3.4% overhead in real-world application workloads, while adding a negligible 5 ms latency on benign file operations.

I. Introduction

File systems constitute an indispensable pillar of contemporary operating systems. Unfortunately, the prevailing lack of system-wide sanity checks on symbolic links [1] has fostered a fertile ground for Link Following attacks (LF attacks). In such attacks, adversaries abuse crafted constructed combinations of symbolic links (referred to in this paper as *link chains*) to redirect otherwise benign file operations in programs toward protected system files, thereby achieving local privilege escalation (LPE) [2], sensitive information disclosure [3], and permanent denial of service (DoS) [4]. Additionally, LF attacks can be categorized into single-step and multi-step attacks, depending on the sequencing of the constructed link chain. The threat is especially acute in the Windows ecosystem: given its market share of more than 70% of all end-user terminals [5], virtually every Windows-based application that relies on the file system is susceptible to LF attacks. As of August 2025, over 1,000 vulnerabilities associated with LF attacks on Windows have been assigned CVE identifiers [6], underscoring the pervasiveness and the severity of this attack surface in Windows file systems.

The few existing studies are predominantly Linux-centric and lack practical applicability to Windows. They either fail to achieve complete protection, rely on heavyweight modeling, or depend on Linux-specific system assumptions that hinder portability. For instance, Chari et al. [7] proposed a path-safety predicate enforced on individual file operations, which cannot capture inter-operation context and thus fails against multi-step LF attacks. Seaborn [8] introduced a heavyweight sandbox mechanism that incurs substantial overhead and relies on deep Linux kernel integration, making it impractical to port. Currently, there is no practical and comprehensive defense against LF attacks on Windows systems.

To address this gap, we are highly motivated to design an effective and efficient defense approach against LF attacks in the Windows file system. We begin with an empirical study of existing countermeasures in practice, which reveals that prior defenses either require substantial manual modeling effort or lack compatibility across Windows versions. More critically, none can simultaneously defend against both single-step and multi-step attacks. Our study further yields two key observations that guide the design of our approach. First, in all observed LF attacks, the link chain is created and later followed by different subjects (i.e., users). Second, despite the complexity of involved operations, LF attacks consistently follow traceable file-state transitions with recognizable patterns.

Based on the empirical study, we present *LinkGuard*, a lightweight state-aware runtime guard against LF attacks in the Windows file system. *LinkGuard* is designed with two primary goals: (G1) Practicality, ensuring minimal performance overhead for both the system and running applications; and (G2) Effectiveness and Extensibility, meaning it can robustly defend against both single-step and multi-step attacks, while remaining adaptable to previously unknown LF attacks. Achieving these goals poses two major challenges:

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.242943
www.ndss-symposium.org

- **Challenge 1**: **How to efficiently filter the vast number of file operations unrelated to LF attacks?** Monitoring all system-wide file operations incurs an obviously prohibitive overhead. However, due to the lack of systematic insights in prior work, we still lack an efficient method to filter out irrelevant file operations and retain only those malicious operations introduced by LF attacks.
- **Challenge 2**: **How to effectively mitigate both types of LF attacks in real time?** The diverse and complex file operation sequences in LF attacks render per-case modeling impractical. Consequently, we lack a battle-tested and reusable strategy that can effectively mitigate LF attacks in real time without relying on extensive manual modeling.

At a high level, two insights derived from observations directly inform the design of *LinkGuard*. First, since the file operations involved in creation and following of link chains in LF attacks are initiated by different subjects (e.g., attackers construct the link chain, while victim programs subsequently perform file operations that follow it), *LinkGuard* narrows its monitoring scope to only those operations related to link chains, thus avoiding costly system-wide monitoring. Second, despite the complexity of file operation sequences during LF attacks, there exists a traceable minimal subset of operations that constitutes the core of such attacks. *LinkGuard* encodes these state transition rules of such operations into parallel finite-state machines (FSMs), enabling efficient detection and mitigation of both single-step and multi-step attacks.

Concretely, *LinkGuard* operates in two coordinated stages to achieve real-time defense. In the first stage (i.e., dynamic subject filtering), *LinkGuard* monitors I/O Request Packets (IRPs) [9], which are data structures used by Windows to facilitate communication between user-mode applications and kernel-mode drivers, focusing on those related to link chain creation and following. For each such request, it identifies the associated subject. If the subjects involved in link chain creation and following differ, *LinkGuard* records all cross-subject file operations along with their subjects. These records are then forwarded to the second stage for further analysis. In the second stage (i.e., real-time attack detection), the collected operations are dynamically transformed into a *Cross-Subject Chain Graph (CSCG)*, a directed graph that captures the dependencies between involved subjects, their file operations, and the evolving file states during the creation and following of the link chain, with edges reflecting the temporal order of operations. Based on this, *LinkGuard* performs state-aware detection by parallel matching each path in CSCG against a set of predefined FSMs. Specifically, if the FSM transitions into an attackable state along any path, the corresponding IRP request is immediately terminated to prevent the LF attack.

For compatibility considerations, we implemented a prototype of *LinkGuard* based on *MiniFilter* [10]. To further assess portability and adaptability, we deployed the prototype on five representative Windows versions using a quad-core Intel i7-1165G7 CPU and 16 GB RAM. Based on a systematically curated dataset from our empirical study, we constructed an evaluation set that contains 70 real-world vulnerabilities in 55 distinct programs. Our evaluation results show that *LinkGuard* successfully defends against all of single-step attacks (26/26) and 95.45% (42/44) of multi-step attacks, with zero false positives on benign file operations. In terms of performance, *LinkGuard* introduces an average of 1% overhead in microbenchmarks and 3.4% in real-world application workloads, while adding a negligible 5 ms latency on benign file operations, which is negligible in practice.

The contributions of this paper are summarized as follows:

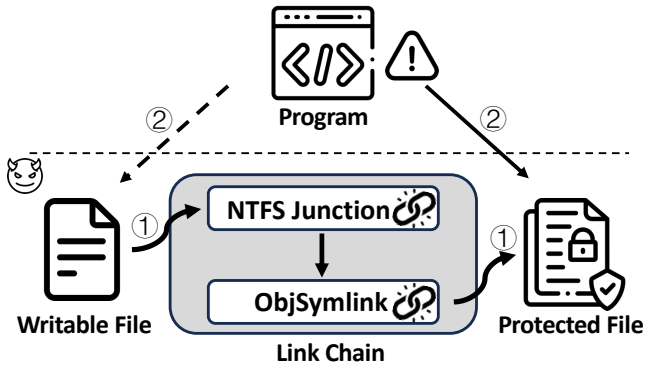
- We conduct the first systematic empirical study of existing countermeasures against LF attacks in practice, and for the first time, categorize and reveal six classes of defenses. Our analysis provides their fundamental weaknesses that prevent them from effectively mitigating. In addition, we provide key observations and novel insights that lay the foundation for designing a more principled and practical defense.
- Building on these insights, we propose *LinkGuard*, a lightweight, state-aware runtime guard against LF attacks in the Windows file system. To the best of our knowledge, *LinkGuard* is the first approach that achieves effective protection against LF attacks in Windows while maintaining compatibility and deployability.
- We evaluate the prototype of *LinkGuard* on five representative Windows versions. On a dataset of 70 real-world vulnerabilities, *LinkGuard* mitigates all single-step attacks and 95.45% of multi-step attacks, with zero false positives on benign file operations. In terms of performance, it introduces an average of 1% overhead in microbenchmarks and 3.4% overhead in real-world application workloads, while adding only 5 ms latency on benign file operations.

II. Background

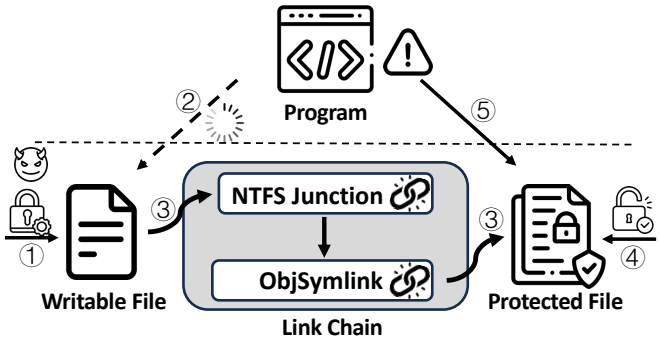
A. Symbolic Links and File Opportunistic Locks.

Symbolic Links. To facilitate flexible file access and compatibility, Windows provides several types of links, among which three forms of symbolic links are commonly used in practice. 1) *NTFS Symbolic Links* [1] are similar to those in Unix-like systems and can point to arbitrary files or directories using absolute or relative paths. 2) *NTFS Junctions* [11] operate similarly to symbolic links but are restricted to directories. In other words, a junction is a directory-only mount point that supports linking directories located on different local volumes. In contrast to symbolic links, junctions can be created without administrator privileges, provided that the user has write permissions to the target directory. 3) *Object Symbolic Links (ObjSymlinks)* [12] are a distinct type of symbolic link that operates on Windows object namespaces [13]. As a result, their scope is confined to the specific object directory in which they reside. For example, a link under the `\RPC Control` namespace is only effective when accessed through this particular namespace. Some namespaces are writable [14], enabling ObjSymlink creation by unprivileged users.

File Opportunistic Locks. Opportunistic locks [15] are a synchronization mechanism in Windows that allow a process to temporarily control access to a file in order to prevent



(a) Overview of a single-step Link Following Attack



(b) Overview of a multi-step Link Following Attack

Fig. 1: Two Types of Link Following Attacks

concurrent modifications by other users or processes. When a process acquires an opportunistic lock with write permissions, subsequent conflicting operations from other processes are blocked until the lock is released.

B. Link Following Attacks

Link Following attacks (LF attacks) can be categorized into single-step and multi-step types. At the core of these types lies the *Link Chain*, a critical attack vector that combines discrete junctions with ObjSymlinks. It achieves the same level of symbolic redirection as NTFS symbolic links, providing a fully functional and unrestricted symbolic link capability within the Windows file system. In Figure 1, assume that the writable file (vulnerable) is `C:\Temp\writable.txt`, and the protected file is `C:\protected.txt`.

The construction sequence of the link chain determines the type of LF attacks. In a single-step attack, as illustrated in Figure 1a, which typically involves a direct redirection within a single file operation (e.g., directly deleting a file), ① the attacker first constructs the link chain, which consists of a junction from `C:\Temp` to `\RPC Control`, and an *ObjSymlink* from `\RPC Control\writable.txt` to `C:\protected.txt`. ② When the program performs file operations on the writable file, it first traverses the junction, which redirects operations to the `\RPC Control` namespace. Then follows the ObjSymlink created at `\RPC Control\writable.txt`, ultimately resolving to `C:\protected.txt`. As a result, any file operations on the writable file are redirected to the protected file.

Multi-step attacks are often used in scenarios where winning a race condition is essential, such as in TOCTOU-based [16] LF attacks, to maximize the adversary's chance of success. As illustrated in Figure 1b, ① the attacker first places an opportunistic lock on the writable file; ② any file operations on the locked file are suspended; ③ the attacker then creates the link chain pointing to the protected file; ④ once the link chain is constructed, the attacker releases the lock; and ⑤ the previously blocked operation resumes and targets the protected file. Overall, multi-step attacks involve complex file manipulations. Moreover, the file operations of the attacker and the program are non-orthogonal; in other words, their operations are interleaved and coupled, which substantially increases the difficulty of detection.

C. Threat Model and Scope

In our threat model, we assume that the attacker has obtained the identity of a standard user account [17] within a multi-user Windows system, with regular user privileges that are strictly constrained and limited file operation capabilities. Attackers can achieve such security consequences through LF attacks: (1) Denial of Service (DoS) [4], for instance, by deleting critical protected configuration files; (2) Privilege Escalation [2], where vulnerable program, either those owned by other users or those running with elevated privileges, are abused via LF attacks to access, modify, or overwrite unauthorized files, enabling both horizontal and vertical privilege gains; and (3) Sensitive Information Disclosure, where attackers perform LF attacks to redirect benign file manipulations from programs to sensitive files (e.g., password stores or user data), causing unintended leakage of confidential information.

This work focuses on detecting and defending against both known types of LF attacks, namely the single-step and multi-step attacks. Unlike prior work [18] [19] [20] [21] on such vulnerability detection, which focuses solely on privileged programs, our approach targets all running programs in the system. This broader scope is essential, as non-privileged programs can also be victims of LF attacks, particularly in horizontal privilege escalation scenarios. However, achieving comprehensive coverage introduces additional challenges to the performance and efficiency of *LinkGuard*. To address this, *LinkGuard* is automatically deployed with elevated privileges during system initialization, ensuring reliable and consistent monitoring across the entire system.

D. Prior Works

To the best of our knowledge, no prior work has systematically examined existing countermeasures against LF attacks in the Windows file system, let alone developed efficient mechanisms for defense. In this section, we first review prior defense mechanisms, all of which are designed for Linux systems, and then analyze their fundamental limitations in mitigating LF attacks, which highlight why directly adapting such approaches to the Windows file system is ineffective.

Prior related works. In the Linux kernel, the `protected_symlinks` flag [22] enforces identity-based access control between the creator and accessor of a symbolic link to prevent unauthorized traversal. Among existing defenses against LF attacks, Basu et al. [23] proposed recording whether a resource is system-protected or application-protected, as a basis for implementing a defense strategy that integrates security enforcement directly within the kernel by leveraging its path resolution subsystem to prevent LF attacks. Building on a different perspective, Chari et al. [7] chose not to focus on access checks but instead applied a "path safety" check to each path element to prevent programs from following unsafe paths. Alternatively, Seaborn et al. [8] proposed a sandbox system that intercepts file operations via a fixed-privilege user-level process, enforcing confinement by replacing the C library [24].

Drawbacks of prior works. First, all existing defense efforts related to LF attacks are designed specifically for Linux-based systems. Due to fundamental differences between the two file systems [25], particularly Windows's descriptor-based access control model [26] and its more complex file operation interfaces, these approaches are inherently non-transferable. For example, porting the `protected_symlinks` mechanism to Windows would conflict with its access control design and risk functional inconsistencies. Even if ported, they remain ineffective in practice. For example, the work by Chari et al. [7] introduced the `safe-open` tool to prevent link following in `open` system calls. However, in the presence of multi-step attacks described in Section II-B, this approach fails. In these cases, opportunistic locks ensure that the file path appears benign at the time of the `open` call, while the attacker creates the link chain afterward and releases the lock to redirect the access, thereby bypassing the defense. As a result, such approaches are fundamentally incapable of mitigating multi-step attacks on Windows. Second, approaches that monitor the global path resolution process in Linux via hooking (e.g., Basu et al. [23]) require extensive compatibility adaptations and incur substantial performance overheads [27], making them impractical for both users and developers.

III. Empirical Study

In this section, we conduct an empirical study of real-world LF attacks and their corresponding countermeasures to systematically summarize the defense adopted by developers in practice, along with their inherent weaknesses. Our analysis is guided by the following research questions: ❶ What are the most commonly adopted countermeasures in practice by developers? (in Section III-B) ❷ What are the weaknesses of these countermeasures? (in Section III-C) ❸ What key observations emerge from this study? (in Section III-D)

A. Study Methodology

In this study, we adopt a vulnerability-driven methodology to systematically collect and analyze real-world countermeasures against LF attacks. The insight underlying this approach is that developers tend to directly encode their defense countermeasures in the patches applied to mitigate these vulnerabilities. Therefore, we identify confirmed LF attack cases from the past five years by querying the National Vulnerability Database (NVD) [28] using a combination of relevant keywords and CWE identifier, particularly CWE-{59, 63, 64} [29] [30] [31]. For each retrieved case, we examine its associated references and advisory materials to extract and classify the corresponding mitigation adopted in practice.

Notably, explicit descriptions of patching strategies are often absent from public advisories. In such cases, we perform manual analysis by downloading and setting up both the vulnerable and patched versions of the affected software. For CVEs with available attack scripts or detailed descriptions, we attempt to reproduce them and use tools like Procmon [32] to trace the invocation stack of file operations, thereby locating the fix implementation in the updated version. For cases lacking sufficient exploit detail, we perform binary differencing between the two versions, focusing primarily on additions or removals of file operations pseudocode. During this process, we rely on IDA [33] and BinDiff [34] for disassembly and differential analysis. The entire analysis was conducted in parallel by three authors. In cases of disagreement over the interpretation of a countermeasure, discussion was held to reach a consensus; if disagreement remained unresolved, we consulted upstream developers or vendors. When no definitive conclusion could be reached, the case was discarded.

In total, we analyzed all countermeasures of 152 CVEs over a period of six weeks. Despite the substantial time investment, we believe that the findings of this study not only offer valuable insights to the community and help raise broader awareness of defenses against LF attacks, but also provide key guidance that informed the design of *LinkGuard*.

B. Types of Countermeasures

Corresponding to the attack scenario depicted in Figure 1, countermeasures fall into three complementary dimensions: protecting files (C1, C2), restricting link chain construction (C3, C4), and limiting excessive privilege usage (C5, C6). The total percentage can exceed 100% since individual cases may employ multiple countermeasures. In this paper, we use the term countermeasure in a broad sense, referring to both post-exploitation mitigation measures and proactive designs or implementations intended to prevent such attacks.

(C1) File ACL Hardening. (67/152, 44.08%) This type of countermeasure protects files by enforcing strict access control lists (ACLs) [35] on file resources throughout program execution. It prevents unnecessary users from obtaining excessive permissions to manipulate files or directories, thereby indirectly preventing the files handled by the program from being exposed to attacks. In practice, developers commonly remove all permissions for regular users or grant only read access, thereby ensuring that attackers cannot modify or misuse these files. Appendix A illustrates how developers applied ACL hardening in the case of `CVE-2024-21835` [36]. Due to its simplicity, straightforward deployment, and immediate effectiveness, this is the most widely adopted countermeasure.

(C2) Secure Path Binding. (12/152, 7.89%) This countermeasure aims to bind temporary and user-specific directories involved during program execution to trusted secure file paths, thereby preventing attackers from manipulating files within them. In Windows, developers typically use environment variables [37] (e.g., `%ProgramData%`) to access these directories for storing intermediate or user-specific data. However, some of these directories are often configured with overly permissive access controls and lack proper restriction mechanisms [38], allowing attackers to manipulate files and perform LF attacks. Binding these directories to controlled secure paths effectively mitigates the risk and ensures that the program operates only on trusted file locations.

```
1  int __stdcall WinMain() {
2  -  wchar_t * path = GetEnv(ProgramData, ···);
3  +  wchar_t * path = GetEnv(SYSTEM_APPDATA, ···);
4     wchar_t * file = StrCat(path, "VBoxSDS.log");
5     ···
6     DeleteFile(file);
7  }
```

Fig. 2: The Secure Path Binding Countermeasure Analysis in CVE-2024-21111

Take `CVE-2024-21111` [39] as an example. As shown in Figure 2, the LF attack arises because the directory is resolved by the environment variable *ProgramData* (Line 2). Since *ProgramData* points to an attacker-controllable path, the attacker can construct a link chain to the file `VBoxSDS.log` (Line 6) to enable the LF attack. To mitigate this risk, the developer rebinds the path to a protected directory resolved by the environment variable *SYSTEM_APPDATA* (Line 3), which is inaccessible to regular users.

(C3) Redirection Guard. (18/152, 11.84%) This countermeasure implemented by Microsoft [40] restricts the construction of link chains by allowing file operations to follow only trusted symbolic links (i.e., created by administrators), while intercepting and blocking untrusted ones. Appendix B shows a practical example of enabling Redirection Guard at the process level. Interestingly, we found that among the 18 CVE cases analyzed, 17 instances of this countermeasure appeared in Microsoft-developed software, highlighting its near absence in third-party applications. This phenomenon can be attributed to two main reasons. First, this countermeasure imposes stringent requirements on the Windows system version (requiring Windows 11 22H2 or later). Second, it adopts a single, rigid trust policy in which only symbolic links created by administrators are deemed trustworthy, which can easily compromise the usability of the program. Consequently, for third-party applications vendors who need to maintain compatibility with a wide range of Windows versions and prioritize application usability, this countermeasure is not acceptable in practice.

(C4) File Name Randomization. (8/152, 5.26%) This countermeasure aims to restrict the construction of a link chain by randomizing the file name. As discussed in Section II-B, an essential prerequisite for creating a link chain is that the attacker must know the name of a writable file. By introducing name randomization, such as using universally unique identifier (i.e., UUID) or random string, developers prevent attackers from accurately identifying or predicting the file name, thereby effectively thwarting LF attacks. As shown in Figure 10 in Appendix C, the countermeasure for `CVE-2025-3617` [41] mitigates potential LF attacks during log file creation by placing the log in a directory with an automatically generated eight-character random name (e.g., `unui3bp.5ks`), making the path unpredictable for attackers, thereby significantly hindering the construction of a usable link chain.

(C5) Program Least Privilege. (15/152, 9.87%) This type of countermeasure mitigates LF attacks that lead to DoS or vertical privilege escalation by restricting unnecessary privilege usage in privileged programs, thereby preventing unauthorized file manipulation as described in Section II-C. To counter this threat and enforce the principle of least privilege [42], developers implement this countermeasure primarily by adopting two strategies: ① employing the Windows-specific mechanism called *Impersonation* [43], where programs temporarily adopts the identity of a low-privilege user during certain operations, rather than executing them with elevated privileges; and ② directly dropping unnecessary privileges, ensuring the program no longer runs with elevated privileges. For instance, the countermeasure for `CVE-2024-44193` [44] directly reduces the program's privilege level from `SYSTEM` [45] to `LOCAL SERVICE` [46] to minimize its file operation scope.

(C6) File Path Validation. (39/152, 25.66%) This countermeasure enforces strict path validation prior to file operations, thereby ensuring that programs execute only correct and safe operations. In other words, it aims to prevent misuse of file operations and thus reduce the risk of exposure to LF attacks.

In practice, developers typically validate the file path before performing any operation by checking whether the path is a symbolic link (e.g., via `GetFileInformationByHandle` API [47]) or by verifying that the resolved final path obtained from the file handle (e.g., via `GetFinalPathNameByHandle` API [48]) matches the original intended path. As shown in Figure 11 in Appendix D, the developer introduced explicit path validation by adding the function `IsPathALink` (Line 3) to mitigate LF attacks. The implementation details are illustrated in Figure 11b: specifically, the program retrieves file metadata using `GetFileInformationByHandle` (Line 3) and then checks whether the number of symbolic links associated with the file handle exceeds one (Line 4), thereby validating that the path does not resolve to any link.

C. Weaknesses of Countermeasures

This subsection summarizes the fundamental weaknesses and why they fall short of providing complete protection.

First, modeling-based countermeasures are inherently demanding (C1, C4, C6), as they require extensive identifica-

tion and instrumentation of relevant file operations, imposing substantial implementation and maintenance burdens on developers. For example, enforcing path validation (C6) demands developers to instrument every file operation throughout the codebase, while ACL hardening (C1) and name randomization (C4) require exhaustive enumeration of all security-sensitive files. Such thorough modeling is time-consuming and error-prone in practice, often relying on ad hoc heuristics.

Next, countermeasures suffer from poor compatibility and limited applicability (C2, C5), which means they can only mitigate LF attacks for a subset of programs or file resources rather than providing comprehensive coverage. For instance, secure path binding (C2) protects only temporary files or files whose paths are resolved via environment variables, while least privilege (C5) is feasible only for programs that do not genuinely require elevated privileges to function correctly.

Finally, and more critically, the countermeasures lack effectiveness in fully defending against LF attacks (C3), meaning that they can still be bypassed by attackers under certain conditions. For example, even with *Redirection Guard*, if the file path remains attacker-controlled, specially crafted UNC paths [49] can circumvent this protection. Similarly, due to the inherent time gap between path validation and subsequent file operations, a window for race conditions exists, thus allowing attackers to exploit the multi-step attack described in Section II-B to bypass file path validation.

D. Key Observations

In this part, we present two key observations distilled from our analysis of these 152 real-world attack cases. Moreover, we will explain in Section IV-B how these observations directly inform the design of *LinkGuard*.

Observation#1: There is always a link chain construction and a link-following file operation, both of which inevitably occur as part of the complete attack process. Of particular importance, these two operations are cross-subject by nature (fully defined in Section IV-C): attackers typically initiate the creation of the link chain, while the victim program follows the link and performs dangerous file operations. Together, these cross-subject operations, i.e., link chain construction (by attackers) and link-following file operations (by programs), constitute the key signature of LF attacks.

Observation#2: LF attacks often appear as complex and unordered sequences of file operations, but they in fact reflect specific file operation states and observable transitions between them. Regardless of the number or types of operations involved, each attack inherently follows a specific state transition pattern. Let us revisit Figure 1. In single-step attacks, the transition proceeds from an initial state where the link chain is fully constructed to a final state where the program follows the link to access a protected target. In multi-step attacks, the process extends from the program encountering an opportunistic lock to the attacker constructing and finalizing the link chain while the program is blocked, and finally to the program resuming execution and following the link. These structured and correlated transitions highlight the feasibility of applying state-aware modeling in our system design.

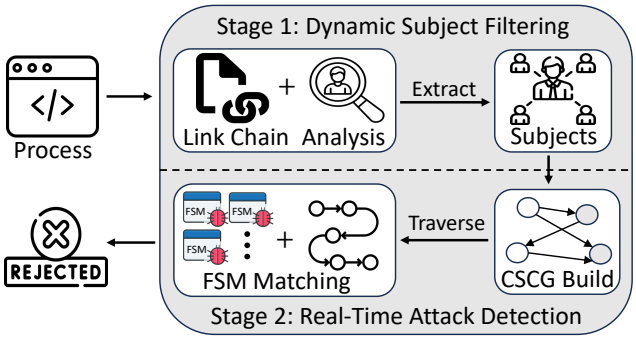


Fig. 3: Design Overview of *LinkGuard*

IV. System Design

A. Design Goals and Challenges

We aim to achieve two key design goals when protecting against LF attacks in the Windows file system at runtime.

G1: Practicality. *LinkGuard* should accurately detect and mitigate LF attacks with low runtime overhead, minimal runtime latency, and high compatibility, thereby ensuring its practicality for real-world deployment.

G2: Effectiveness and Extensibility. *LinkGuard* should effectively mitigate both known types of LF attacks, namely, single-step and multi-step attacks. Moreover, the design should facilitate ease of extension, allowing rapid adaptation to mitigate newly identified LF attacks, that is, by incorporating their characteristic behaviors into FSM rules for timely defense.

Building on the design goals and weaknesses of both prior works in Section II-D and existing countermeasures in Section III-C, providing a more robust defense against LF attacks requires tackling two key challenges in mitigation:

(1) How to efficiently filter the vast number of file operations unrelated to LF attacks? Considering that the Windows file system involves a massive and diverse range of file operations, exhaustively monitoring every single file operation would significantly increase performance overhead (as further discussed in Section VI-F) and simultaneously capture a large volume of "link-following-unrelated" operations, which could interfere with the mitigation of comparatively scarce attacks and ultimately hinder *LinkGuard* from achieving G1.

(2) How to effectively mitigate both types of LF attacks in real time? We require a battle-tested yet reusable approach capable of mitigating two types of LF attacks. To achieve G2, our approach should not only detect both single-step and multi-step attacks that involve diverse types of file operations and complex operation sequences, but also avoid the well-known heavy modeling burden inherent in traditional defenses, ensuring it can be easily adapted with minimal manual effort.

B. Overview and Insights

In this subsection, as illustrated in Figure 3, we present the overview of *LinkGuard*, a lightweight state-aware run-

time guard against link-following attacks in the Windows file system. We further elaborate on how the key observations in Section III-D inspire the key insights that address two primary challenges and ultimately fulfill our two design goals.

Insight-1 & Solution-1. (Stage 1) Based on *Observation#1*, our key insight is that since LF attacks are inherently cross-subject, it is unnecessary to monitor all file operations system-wide. Instead, we focus solely on whether the construction and following operations of the link chain are performed by different subjects (e.g., different user identities). Accordingly, the first stage of *LinkGuard* performs an efficient dynamic subject filtering. This involves selectively monitoring and recording file operations related to link chain creation and following at runtime, and analyzing whether they originate from distinct subjects. If so, *LinkGuard* extracts the involved subjects along with their corresponding link-related file operations for further detection. In this way, *LinkGuard* significantly reduces monitoring overhead by filtering out a large volume of operations irrelevant to link-following, thereby achieving G1.

Insight-2 & Solution-2. (Stage 2) Based on *Observation#2*, our key insight is that LF attacks inherently involve observable transitions between file operation states, which enables us to design a set of lightweight and parallelizable finite-state machines (FSMs) that monitor whether these transitions reach attackable terminal states and thereby intercept both single-step and multi-step attacks concurrently. Specifically, since these transitions span across subjects, *LinkGuard* first consolidates the subjects and their associated file operations (Extracted during Stage 1) into a Cross-Subject Chain Graph (CSCG), a directed graph that reflects the temporal and dependency relationships among file operations initiated by different subjects. Each path in the CSCG is then concurrently propagated to FSM instances specialized for detecting different types of LF attack, which independently evaluate whether a stateful attack condition is satisfied. By encoding only a small number of transition rules into parallel FSMs, *LinkGuard* enables effective detection of both known attack types. Moreover, it remains easily extensible to previously unknown LF attacks by incrementally adding corresponding rule combinations into FSMs. Therefore, we can successfully achieve G2.

C. Dynamic Subject Filtering

This subsection details how *LinkGuard* dynamically identifies and then filters cross-subject relationships and extracts subjects within their all file operations involving the link chain, including both creation and subsequent following.

LinkGuard first focuses on monitoring primary file operations based on whether they participate in link chain construction and following. In Windows, file operations are reflected in the kernel layer as IRP (I/O Request Packet) requests [9]. Therefore, instead of monitoring a wide range of high-level APIs in user space, *LinkGuard* directly inspects IRP requests corresponding to relevant operations in the kernel. This design significantly reduces overhead and improves monitoring efficiency. Table I illustrates the classification of these operations together with their associated IRP request types. It is worth emphasizing that we regard both `Create/Open` and `Lock` operations as being involved in both link chain construction and link following. The former is because both constructing and following a link chain rely on file handles, while the latter is particularly relevant in multi-step attacks: after an opportunistic lock is acquired, the attacker typically performs link chain construction, and once the lock is released, the victim program proceeds with link-following operations. Based on this classification, *LinkGuard* selectively monitors only these essential operations, thereby significantly improving efficiency by avoiding the need for global analysis of all file API calls. Subsequently, *LinkGuard* performs subject identification for each monitored file operation.

Definition-1 (Subject). Let op be a file operation executed by a thread T in the Windows system. We define the *subject* of op, denoted by $\mathsf{Sub}(op)$, as follows:

$$\mathsf{Sub}(op) = \begin{cases} \mathsf{Token}(T) & \text{if } op \text{ succeeds} \\ \mathsf{Group}(f, p) & \text{if } op \text{ fails during execution} \end{cases} \quad (1)$$

where $\mathsf{Token}(T)$ denotes the effective access token [58] used by thread T at the time of executing op, and $\mathsf{Group}(f, p)$ denotes the set of users who belong to a group with specific permission p (i.e., `FULL_ACCESS`) on the target file f.

To accommodate the complexity of Windows permission management model, particularly the use of impersonation [59] and token substitution [60], the definition in Equation (1) performs identification at the granularity of individual threads. Specifically, when a file operation op succeeds, we define $\mathsf{Sub}(op)$ as the user associated with the effective access token used by the executing thread. For example, if a privileged process (e.g., running as the `SYSTEM` user) employs impersonation to create a file on behalf of a regular user (e.g., `Alice`), the actual subject of the operation is the impersonated user `Alice`, rather than the original privileged context `SYSTEM`. In contrast, when op fails, we adopt the conservative principle, defining the subject as the set of users belonging to the group that holds the permission on the target file. This conservative strategy is motivated by the observation that users with `FULL_ACCESS` constitute the minimal set of principals capable of executing the intended operation, and thus serve as a conservative lower bound for the actual subject set. Ultimately, *LinkGuard* ensures that every file operation can be consistently associated with a subject, or occasionally with a set of subjects, which is crucial for accurate real-time detection and mitigation of LF attacks in subsequent stages.

Extract. *LinkGuard* leverages the MiniFilter [10] framework to register callback functions both before (pre-operation [61]) and after (post-operation [62]) each file operation. Specifically, it monitors and analyzes both the construction of the link chain and the subsequent file-following operations. If both operations are performed by the same subject, *LinkGuard* considers the symbolic link traverse to be benign and takes no further action, a pattern frequently observed in legitimate use cases, such as when users access files through desktop shortcuts [63]. In contrast, when these operations are carried

TABLE I: **Monitored file operations and corresponding IRP requests in link chain construction and following. *LinkGuard* tracks selected file operations at the IRP level during link chain creation and following to determine whether they are performed by different subjects.**

Primary Operation	Corresponding IRP requests/flags	link chain construction	link chain following
Create/Open	`IRP_MJ_CREATE` [50]	✓	✓
Write	`IRP_MJ_WRITE` [51]	✗	✓
Rename	`FileRenameInformation` [52]	✗	✓
Delete	`IRP_MJ_SET_INFORMATION` [53], `FILE_FLAG_DELETE_ON_CLOSE` [54]	✗	✓
Set DACL	`IRP_MJ_SET_SECURITY` [55]	✗	✓
Symbolic Link Create	`IRP_MJ_FILE_SYSTEM_CONTROL` [56]	✓	✗
Lock	`FSCTL_REQUEST_OPLOCK` [57]	✓	✓

✓/✗: involved / not involved

out by different subjects, such cross-subject behavior is regarded as potentially malicious by *LinkGuard*. For example, as illustrated in Figure 1a, the link chain is created by the attacker, while the actual file operation that follows the symbolic link is performed by the victim program. Clearly, these operations are initiated by distinct subjects (recall that, under our threat model, the attacker and the victim program do not share the same subject identity). In such cases, *LinkGuard* extracts all file operations involving distinct subjects during the link chain construction and following. Along with each operation, it also records the associated operation metadata, including the identified subject, execution timestamp, status, and target file, to facilitate precise graph construction.

D. Real-Time Attack Detection

In this subsection, we detail the second stage of *LinkGuard*, which focuses on the runtime defense of LF attacks. This stage consists of two core components: the construction of the Cross-Subject Chain Graph (CSCG) and the FSM-based rule matching for attack detection across subjects.

CSCG Building. To uncover potential attacks, *LinkGuard* integrates file operations across different subjects into a unified structure. This cross-subject perspective is essential: only by correlating operations that are otherwise isolated within individual subjects can we detect coordinated behaviors indicative of LF attacks. More importantly, the transitions toward attackable states, central to the detection logic of FSMs, span multiple subjects, making it necessary to capture such inter-subject dependencies for effective detection.

Definition-2 (Cross-Subject Chain Graph). Let $\mathcal{O} = \{op_1, op_2, \ldots, op_n\}$ be the set of link-related file operations extracted from Stage 1 of *LinkGuard*, each associated with subject. We define the Cross-Subject Chain Graph (CSCG) as a directed graph $G = (V, E)$, where:

- $V = \mathcal{O}$, i.e., each node represents a file operation op_i;
- Each op_i is a tuple (f_i, s_i, t_i, r_i), where f_i is the target file, s_i is the subject, t_i is the execution timestamp, and r_i is the result status (e.g., *STATUS_SUCCESS*);
- E is a set of directed edges representing the temporal order between operations across different subjects. Formally, for any path $(op_{i_1}, op_{i_2}, \ldots, op_{i_k})$ in G, it holds that $t_{i_1} \leq t_{i_2} \leq \cdots \leq t_{i_k}$, ensuring that execution flows logically and chronologically from earlier to later actions. Moreover, each adjacent pair (op_i, op_j) on a path must satisfy a file dependency condition: their target files f_i and f_j must either refer to the same file, reside in a parent-child directory relationship, or share a common directory (e.g., the files `C:\dir\a` and `C:\dir\b` share the directory `C:\dir`).

LinkGuard incrementally constructs the CSCG by continuously monitoring file operations at runtime. This dynamic construction ensures that the graph evolves in real time to incorporate the most recent file operation sequences, which is essential for the timely detection of LF attacks. Although the CSCG dynamically expands during execution, its growth is strictly bounded in our design. Each graph is released after completing FSM-based matching, and a size threshold is enforced to prevent uncontrolled expansion and to maintain stable memory usage over time. Specifically, during the construction of the CSCG, *LinkGuard* adds a directed edge from node v_a to node v_b if the operation represented by v_a precedes that of v_b, and the target files f_a and f_b satisfy the dependency relation defined earlier (i.e., identical paths, parent-child directories, or shared directories). During construction, *LinkGuard* applies several practical pruning strategies. In particular, if a node has neither valid predecessors nor successors under the dependency rule, it is discarded, as such isolated operations cannot contribute to any valid state transition in the FSM. Likewise, when the target file path resides in a protected directory (e.g., `C:\Windows\system32`), *LinkGuard* terminates further construction since such paths are inherently beyond the attacker's control. Notably, when the number of nodes in a CSCG exceeds a predefined threshold and new nodes are added, *LinkGuard* removes the earliest recorded nodes to keep the graph size bounded. Of greater significance, the compactness of the resulting CSCG stems from the design of Stage 1, which filters and retains only file operations related to the construction and following of link chains. This selective inclusion fundamentally determines the maintainability of CSCG, as further demonstrated in Section VI-F, laying the groundwork for fast graph traversal and lightweight FSM-based rule matching.

FSM-based Rule Matching. *LinkGuard* traverses the CSCG using depth-first search [64] (DFS) to enumerate all time-ordered paths across subjects. Each path is treated as an independent execution trace and is fed into every FSM instance in parallel for rule-based matching. After completing the traversal

and all FSM-based rule matching, the corresponding CSCG is released, ensuring that memory consumption remains stable and preventing any risk of self-induced resource exhaustion. Here, the FSM transition rules are distilled from *Observation#2* and encode the minimal state sequences necessary to characterize LF attacks. As illustrated in Figure 4, the figure depicts how different FSMs concurrently detect single-step and multi-step attacks. These FSMs are lightweight by design and operate in parallel. Here, parallelism does not refer to technical concurrency, but rather to *LinkGuard*'s capability to simultaneously support rule checking for both single-step and multi-step attacks.

For each enumerated path in CSCG, *LinkGuard* initiates the matching process from the initial states of multiple FSMs in parallel. Note that a state transition occurs when a node along the path v satisfies a predefined rule-based transition condition. Such a condition typically corresponds to a semantically critical operation (e.g., link chain creation) that represents an inherent and indispensable step in an LF attack. The FSM advances through states until it reaches a terminal "attackable" state, which represents a confirmed exploit condition. Upon reaching such a state, *LinkGuard* immediately terminates the execution of the corresponding operation. This is ensured by the fact that, at the moment the FSM completes its matching and enters an attackable state, the operation is still within its pre-operation phase. Leveraging this timing, *LinkGuard* performs two actions in sequence to prevent the operation from being carried out: (1) setting operation's return status to `ACCESS_DENIED`, and (2) suppressing the dispatch of the operation request to the kernel execution layer (i.e., `NTFS.sys` [65]).

It is worth noting that although the computational complexity of *LinkGuard* 's defense process is $O(n^3)$, it remains acceptable in practice. This is because CSCG growth is bounded by design through size threshold control and graph release after FSM-based matching, and its size under real workloads is inherently small (as discussed in Section VI-D). Considering the reset conditions of FSMs, each FSM instance is reset to its initial state either upon reaching an attackable state and completing its handling or after the full traversal of the given path. This ensures that *LinkGuard* can continuously detect and intercept both single-step and multi-step LF attacks in progress, without interference across independent execution traces. As illustrated in Figure 4, nodes in CSCG that satisfy transition rules for single-step attack are marked in blue, while those satisfying multi-step attack rules are marked in green. Traversing the CSCG yields multiple execution paths that reflect cross-subject interactions. Among them, the path $P_1 = \{v_1, v_3, v_4, v_6\}$ corresponds to a single-step attack, while the path $P_2 = \{v_1, v_2, v_4, \ldots, v_5, v_7\}$ spans multiple operations and ultimately triggers a multi-step attack. During real-time detection, each FSM (FSM_i) operates in parallel and starts from its initial state s_0. For single-step FSM, a transition occurs when the matched path satisfies the corresponding rule conditions (i.e., v_1 and v_6 in P_1), eventually reaching an attackable state s_2. Similarly, the multi-step FSM transitions through $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$ upon matching rule conditions (v_2, v_4, and v_7 in P_2). It is worth emphasizing that each FSM defines only a small number of rule-based transitions over the CSCG. This bounded and rule-driven structure enables lightweight and parallel matching, which is critical for achieving timely and effective detection in real-world runtime environments.

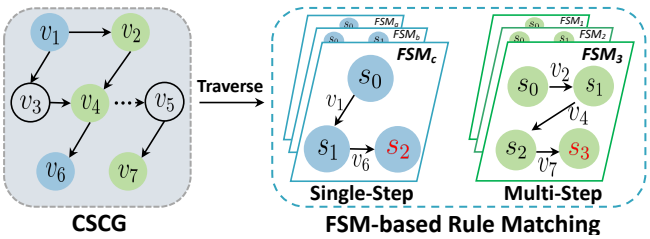


Fig. 4: **The second stage: *LinkGuard* conducts parallel FSM matching over the CSCG to detect coexisting single-step and multi-step LF attacks**

V. Implementation

To ensure broad compatibility across Windows platforms, we implement *LinkGuard* as a kernel-mode system driver. As summarized in Table II, this design choice aligns with Microsoft's official driver architecture [66] and guarantees seamless integration with both desktop and server editions across architectures (x86 & x64). Notably, ARM64 support is only available on Windows 10 version 1903 and Server 2022 or later, due to platform-level restrictions on kernel-mode driver deployment. This limitation originates from Windows OS itself rather than from *LinkGuard* 's design.

TABLE II: **Platform compatibility of *LinkGuard* across architectures and Windows versions.**

Windows Version Range	Supported Architectures
Windows 7–8.1 (Desktop)	x86 / x64
Windows 10–11 (Desktop)	x86 / x64 / ARM64*
Server 2008 R2–2019	x64
Server 2022–2025	x64 / ARM64

* ARM64 support requires Windows 10 version 1903+

To monitor file operations, *LinkGuard* leverages pre-operation [61] and post-operation [62] callbacks registered via the MiniFilter framework [10], which provides a standard mechanism for intercepting and filtering I/O requests. Pre-callbacks allow malicious actions to be intercepted before execution, while post-callbacks ensure access to complete operation metadata, including outcome status that is only available after execution. *LinkGuard* is implemented in C/C++ with about 3.9K lines of code. Specifically, it includes about 2.9K lines for the core implementation (1.3K for Stage 1 and 1.6K for Stage 2), 0.8K lines for header files, and 0.2K lines for CMake build and configuration scripts. We share our artifact [1] to facilitate future research.

[1] https://doi.org/10.5281/zenodo.17481221

VI. Evaluation

In this section, we evaluate the prototype of *LinkGuard* in terms of its security, defense effectiveness, and performance, with the aim of answering the following questions:

- **RQ1:** Does *LinkGuard* pose any security risks that introduce new potential LF attacks or permit evasion of its mitigation mechanisms? (in Section VI-B)
- **RQ2:** How effective is *LinkGuard* in mitigating LF attacks? (in Section VI-C)
- **RQ3:** Does *LinkGuard* introduce false positives or interfere with benign file operations? (in Section VI-D)
- **RQ4:** What is the overhead introduced by *LinkGuard* in terms of system and application performance? (in Section VI-E)
- **RQ5:** Does the first stage of *LinkGuard* contribute to CSCG maintainability and the overall defense efficiency? (in Section VI-F)

A. Experimental Setup

1) Experimental Environment: To comprehensively assess the compatibility of *LinkGuard* across a wide range of deployment scenarios, we conduct all experiments on five representative Windows environments: Windows 7 SP1, Windows 10 (22H2), Windows 11 (24H2), Windows Server 2008 SP2, and Windows Server 2025 (24H2). These platforms span the earliest, latest, and intermediate versions supported by *LinkGuard*. All systems are configured with a 4-core Intel Core i7-1165G7 CPU and 16 GB RAM, and run on bare-metal hardware to ensure accurate measurement of kernel-level performance. All experiments are conducted on hard disk drives (HDDs) to simulate real-world deployment scenarios. Unless otherwise noted, all evaluations are executed concurrently across these systems to validate the robustness and version portability of our prototype implementation.

2) Dataset: No single dataset can fully capture both the effectiveness and performance of *LinkGuard*. To this end, we design three complementary datasets, each targeting a specific aspect of the prototype's evaluation:

- **Dataset-α (Vulnerable Applications):** This dataset is derived from the vulnerability set in Section III-A, and is used to thoroughly assess the effectiveness of *LinkGuard* in defending against LF attacks. Specifically, from our set of 152 vulnerabilities, we select a subset of cases that meet the following criteria: (i) the vulnerable software package is publicly available, and (ii) either detailed reproduction instructions or exploit scripts are provided. Based on these criteria, the final dataset consists of 70 CVEs across 55 applications from diverse functional domains. A detailed breakdown of included and excluded cases in this dataset is provided in Appendix E.
- **Dataset-β (Microbenchmarks):** This dataset is used to conduct microbenchmarks that evaluate the performance overhead introduced by *LinkGuard* under I/O-intensive workloads, with a particular focus on file system behavior. Given that *LinkGuard* primarily monitors file system operations (rather than network or UI-related behaviors), we focus on I/O-intensive workloads that place significant pressure on the underlying file system. The selected programs include well-known I/O benchmark tools such as IOzone [67] and DiskSpd [68], which are widely used in measuring file system performance under stress.
- **Dataset-γ (Real-world Applications):** This dataset is used to evaluate the runtime performance of *LinkGuard* on representative real-world applications and its false positive rate on benign file operations. It includes widely used, production-grade applications that span diverse application domains, such as web servers (e.g., Apache [69]), relational databases (e.g., Microsoft SQL Server (MSSQL) [70]), file transfer utilities (e.g., WinSCP [71]), and compression tools (e.g., 7-Zip [72]), reflecting typical workloads encountered in both desktop and server environments.

B. RQ1: Security Analysis

1) Security Analysis for Possible Attacks: We have verified that *LinkGuard* does not introduce any new LF attack surfaces. At runtime, it passively monitors and conditionally blocks file operations without generating auxiliary file activities, thereby avoiding any exploitable side effects. In addition, *LinkGuard* operates entirely within the kernel and is securely deployed in protected directories to prevent unauthorized modification. We further examined the prototype using well-established static analysis tools, including Cppcheck [73] and Flawfinder [74], to ensure the absence of typical memory vulnerabilities.

2) Security Analysis for Potential Evasion: Under our threat model, we also consider adversaries attempting to infer or evade *LinkGuard*'s FSM-based mitigation. However, such efforts are impractical, as attackers lack privileged access and cannot bypass kernel-level monitoring. In addition, *LinkGuard* exposes no user-space interfaces (e.g., IOCTL [75] and RPC [76]), thus preventing direct inspection. Indirect inference through behavioral probing is also infeasible due to the vast space of possible file operation sequences and the minimal nature of FSM transitions, which capture only essential steps required for real attacks. Any attempt to evade detection would break these steps, rendering the attack ineffective.

C. RQ2: Defense Effectiveness

Table III presents the defense effectiveness of *LinkGuard* on dataset-α, which includes 70 real-world vulnerabilities. We categorize the evaluated vulnerabilities based on the types of file operations involved in the LF attacks and their corresponding security impacts. Specifically, DoS attacks are primarily associated with file creation, overwrite, and deletion operations; sensitive information disclosure often arises from permission assignment and file relocation (moving and copying), where protected files are exposed to unauthorized users; and privilege escalation typically involves file relocation or deletion, allowing attackers to gain system-level privileges by deleting files [77] or access files belonging to other users.

Overall, *LinkGuard* successfully mitigates 97.1% (68/70) of LF attacks in our evaluation, including all single-step attacks

TABLE III: ***LinkGuard*'s effectiveness for mitigating real-world LF attacks in dataset-α**

Operation Category	Impact	Attack Type	
		S	M
File Creation and Overwriting	Denial of Service	15 / 15	4 / 4
File Move and Copy	Information Disclosure	1 / 1	8 / 8
File Permission Assignment	Information Disclosure Privilege Escalation	7 / 7	2 / 2
File Deletion	Denial of Service Privilege Escalation	3 / 3	28 / 30
Overall	–	26 / 26	42 / 44
Proportion	–	100%	95.45%

S: Single-step attacks. M: Multi-step attacks.

(26/26) and 95.45% (42/44) of multi-step attacks. This high defense coverage is consistent with our design expectations. Since *LinkGuard* triggers block only when the FSM reaches an attackable state, signaling that the critical file operation of the LF attack is about to be performed, it does not produce false negatives for exploitable vulnerabilities. Meanwhile, we observe no false positives on benign file operations, with detailed results presented in Section VI-D. In addition, *LinkGuard* can be easily adapted to defend against potential new or previously unknown LF attacks, requiring only minimal engineering effort to extend the FSM transition rules accordingly.

Attacks Mitigation. *LinkGuard* successfully mitigates all single-step LF attacks and 42 out of 44 multi-step LF attacks (95.45%) within dataset-α. As part of the mitigation of single-step attacks, let us consider `CVE-2023-2939` [78] as an illustrative example. The privileged Chrome installer [79] writes crash report files to an unprotected directory (`C:\Windows\Temp\Crashpad\`) without verifying the file path. From the viewpoint of *LinkGuard*'s defense logic, the attack involves a transition from a link chain created by a non-privileged subject to a privileged program performing a follow-up write operation. This behavior is naturally captured by the FSM, allowing the FSM to naturally reach the attackable state just before the file write occurs, at which point *LinkGuard* enforces a timely denial of the operation. Similarly, in the mitigation of multi-step attacks, `CVE-2024-44193` [80] serves as a representative example. The installation of iTunes [81] registers a privileged service named `AppleMobileDeviceService`, which first checks for the existence of files in an unprotected directory (`C:\ProgramData\Apple\Lockdown\`) and then deletes them without verifying whether they are symbolic links. *LinkGuard* effectively mitigates this attack by capturing a structurally predictable "locked-then-triggered" sequence, where the attacker sets an opportunistic lock that is subsequently hit by the victim program, and then constructs the link chain before the operation is triggered upon lock release. This recurring transition pattern is reliably recognized by the FSM, enabling timely and precise mitigation.

Insufficient Mitigation. We identify two vulnerabilities that *LinkGuard* fails to fully mitigate, both stemming from a shared root cause. Taking `CVE-2025-21373` [82] as an example, the attack uses an NTFS-formatted removable media (e.g., USB) on which attackers can pre-construct a malicious link chain in an isolated environment where *LinkGuard* is not deployed. When the device is subsequently connected to a system protected by *LinkGuard*, the creation of link chain is no longer observable for *LinkGuard*, rendering the defense ineffective. However, since this class of attacks requires physical access to target machines, a condition that lies outside the scope of our threat model, we consider the insufficient mitigation under such extreme conditions to be acceptable.

D. RQ3: False Positive Evaluation

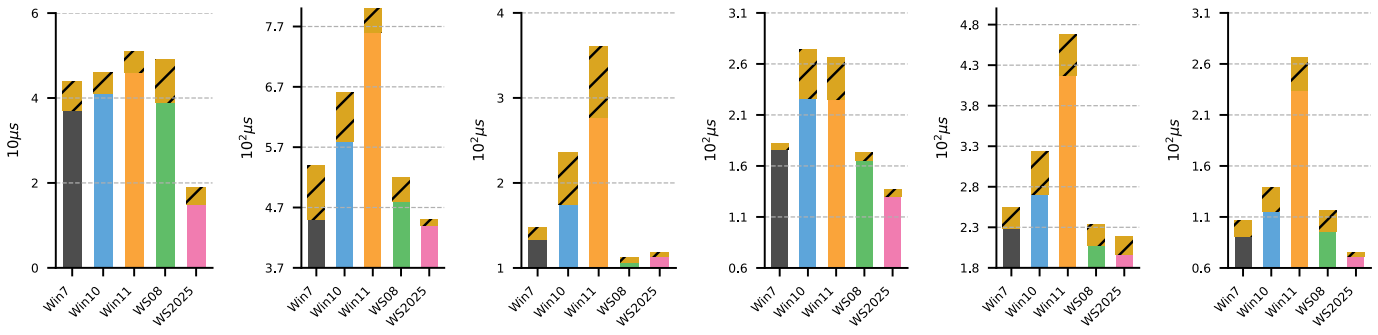
While our effectiveness evaluation on dataset-α did not reveal any false positives, we further assessed this aspect using dataset-γ to more comprehensively answer *RQ3*. This dataset reflects realistic production environments involving multiple interacting subjects. It includes widely used applications such as Apache and MSSQL, thereby covering representative multi-component workloads. Such diversity enables us to evaluate *LinkGuard* under environments with frequent cross-subject interactions, ensuring that it operates correctly without disrupting benign file activities. To evaluate this aspect, we deploy *LinkGuard* across the five Windows environments discussed earlier and configure it with dataset-γ. Each environment is equipped with production-relevant applications and operated continuously for 72 hours under normal usage.

TABLE IV: **False positive evaluation across different environments in dataset-γ**

Env.	Operation Category				Cross-Subj Ops	CSCG N/E	FP
	FC	FM	FD	FPA			
Win7	3M	420	1.2k	108	35k	34k/34k	0
Win10	792k	9.4k	2.6k	336	4.7k	1.7k/1.8k	0
Win11	20M	3.9k	503k	2.9k	11.9k	3.1k/ 3.4k	0
Srv 2008	1.2M	15k	6.9k	48	1.8k	0.2k/0.2k	0
Srv 2025	2.8M	160	1.8k	768	11.4k	10.5k/10.8k	0
Avg.	5.56M	5.8k	103k	0.8k	12.8k	12.4k/12.5k	0

FC/FM/FD/FPA denote four categories of file operations: File Creation and Overwriting, File Move and Copy, File Deletion, and File Permission Assignment, respectively. **Cross-Subj Ops** refers to the number of valid cross-subject file operations. **CSCG N/E** indicates the average CSCG size in nodes and edges, and **FP** represents the number of false positives.

As shown in Table IV, while per-environment file activity is high (e.g., file creation averages 5.56M events), only about 12.8k operations involve cross-subject interactions. Furthermore, the CSCGs constructed from these interactions remain compact, with an average of approximately 12.4k nodes and 12.5k edges, representing a highly compressed value compared to the overall volume of file operations in this multi-component workload. This contrast indicates that even under multi-component workloads, the CSCG remains compact and manageable in practice. The relatively small size of the CSCG



(a) `open` (b) `read` (c) `write` (d) `create` (e) `move` (f) `delete`

Fig. 5: Execution time of standard file operations across five Windows versions. Std.Dev. below 8.0%

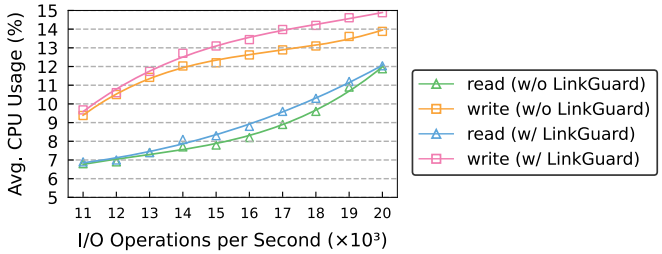


Fig. 6: Impact of *LinkGuard* on average CPU usage under varying IOPS for read and write. Std.Dev. below 0.5%

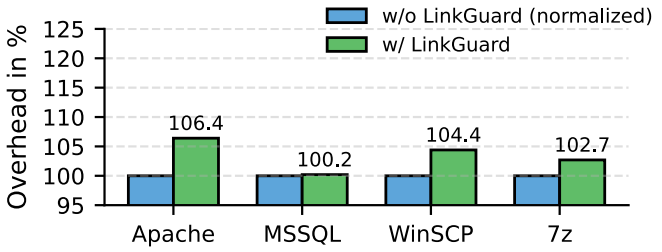


Fig. 7: Normalized overhead of different Windows common applications. (Avg. Increase: ∼3.4%, Std. Dev.: $< 2.4\%$)

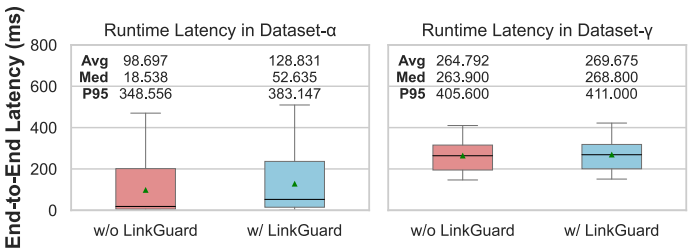


Fig. 8: Runtime latency under defense scenarios (dataset-α) and benign deployment in real-world applications (dataset-γ).

can be attributed to two factors: first, symbolic link creation and follow behaviors are inherently rare in Windows systems, and cross-subject occurrences of such operations are even less frequent; second, *LinkGuard* applies practical pruning strategies during CSCG construction (in Section IV-D) to exclude those CSCGs with minimal relevance to potential attack behaviors. Finally, *LinkGuard* did not incorrectly intercept any of these benign operations, resulting in zero false positives. This outcome meets our expectations, as the FSM transition rules in *LinkGuard* are strictly derived from the lower bound of file operation sequences necessary to trigger LF attack sequences that are characteristic of malicious behavior and rarely observed in benign workflows.

E. RQ4: Performance Evaluation

In this part, we evaluate the performance impact of *LinkGuard* to answer RQ4. Specifically, we use dataset-β to assess system-level microbenchmark overhead introduced by *LinkGuard*, and dataset-γ to evaluate performance on real-world application workloads. All experiments are also executed concurrently across the five Windows environments described earlier, implicitly reinforcing the compatibility of our prototype throughout the evaluation process.

*1) **Microbenchmarks**:* We conduct microbenchmark experiments using dataset-β to evaluate the execution time overhead introduced by *LinkGuard* for standard file operations, as well as its broader impact on system-level performance.

Operation Execution Time: Figure 5 presents the measured execution time for six representative file operations. The colored bars represent the performance of the baseline system, while the yellow bars with black stripes indicate the additional execution overhead introduced by *LinkGuard*. Specifically, *LinkGuard* increases the execution time by an average of 10.6% across all tested environments. In particular, Windows 10 and 11 exhibit higher overheads compared to other systems, with execution time increases for standard file operations ranging approximately from 12.2% to 35%. In contrast, other environments incur relatively minimal overhead (0.5%–8.3%). In terms of standard file operations, *LinkGuard* results in an average execution time increase of 10.8%. Among these, `write` operations exhibit the most noticeable variance, with execution time increasing by approximately 4.4% to 34.8%, while other operations show only modest increases.

CPU Overhead: We further evaluate the system-level performance overhead introduced by *LinkGuard* during read and write operations, specifically targeting I/O-intensive workloads. Figure 6 illustrates the average CPU usage under in-

creasing IOPS, with cubic spline regression applied to smooth the observed trends. For read operations, the additional CPU overhead introduced by *LinkGuard* diminishes progressively as IOPS increases, with the usage curves for the *LinkGuard* and baseline systems eventually converging. This suggests that the marginal cost of monitoring `read` becomes negligible under high I/O loads. In contrast, for write operations, the average CPU usage shows a small but consistent increase, stabilizing around a 1% overhead even at higher IOPS levels. These trends indicate that *LinkGuard* imposes only minimal and statistically stable performance overheads, which are acceptable for practical deployment.

*2) **Overhead on Real-world Application workloads**:* Along with the microbenchmarks, we also assess the runtime overhead introduced by *LinkGuard* on real-world application workloads. To this end, we conduct performance analysis using dataset-γ, which includes representative production-grade programs such as Apache [69], MSSQL [70], WinSCP [71], and 7-Zip [72]. Figure 7 shows the overall overhead of each application compared to the native system, with an average increase of approximately 3.4% across all applications.

Apache [69] serves a 64 KB file in response to 1,000 requests issued by the ApacheBench [83]. Overall, deploying *LinkGuard* introduces an average performance overhead of approximately 6.4%, which has no substantive impact on Apache's performance when compared to typical network conditions, such as transient congestion or latency fluctuations.

MSSQL [70] is evaluated using the HammerDB [84] benchmark, where each SQL query internally triggers `read` or `write` operations. Deploying *LinkGuard* results in only a 0.2% performance overhead, yielding performance that is virtually identical to that of the native system.

WinSCP [71] transfers a 1 GB file from a local network host. During the download, a `write` operation is issued every 32 KB of received data, resulting in over 5,500 `write` operations to complete the transfer. With *LinkGuard* deployed, the overall performance overhead increases by 4.4%, which is consistent with the microbenchmark results for `write`.

7-Zip [72] decompresses a 1.5 GB source archive (i.e., Linux kernel 6.15.6 source archive). During this process, 7-Zip opens each file, reads its contents through in-memory buffers, and writes the extracted data to disk. The observed performance overhead closely aligns with the microbenchmark results for the `open`, `read`, and `write` operations presented in Figure 5.

Runtime Latency: We further evaluate the latency introduced by *LinkGuard*. As shown in the left box plot of Figure 8, on 20 randomly selected vulnerable applications from dataset-α, the average and P95 (i.e., the 95th percentile) latency increased $\sim$30 ms after *LinkGuard* was deployed to mitigate corresponding LF attacks. In contrast, the right plot shows that for benign file operations in real-world applications from dataset-γ, both metrics increased by only $\sim$5 ms. This difference is expected, as malicious operations typically trigger more FSM transitions, leading to higher processing costs, and we consider the additional latency introduced during LF attack mitigation to be acceptable for real-time protection.

F. RQ5: Ablation Studies

In this subsection, we systematically evaluate the contribution of the first stage of *LinkGuard* to both the maintainability of CSCG and the overall performance. Specifically, we isolate Stage 1, namely, efficient subject filtering, to examine how it reduces unnecessary monitoring of irrelevant subjects and operations, thereby improving detection precision and preserving the structural clarity of CSCG for long-running defense.

TABLE V: Comparison of *LinkGuard* and *LinkGuard*$_{NF}$ in CSCG maintainability and performance metrics

Baseline	Nodes	Edges	Latency (ms)	CPU (%)
LinkGuard	53	71	128.8	6.41
LinkGuard$_{NF}$	4919.5	1351	1611.1	14.46
Avg. Increase	**9197.2%**	**1802.8%**	**1150.8%**	**125.6%**

Accordingly, we compare *LinkGuard* with the variant *LinkGuard*$_{NF}$ in which the first stage is replaced by system-wide file operation monitoring, in order to assess its impact during the defense. Table V presents the performance metrics when defending against LF attacks by *LinkGuard* and its variant *LinkGuard*$_{NF}$, which omits efficient subject filtering.

From the perspective of CSCG maintainability, *LinkGuard* produces substantially smaller graphs than *LinkGuard*$_{NF}$, demonstrating that most concurrent file operations are irrelevant to LF attack mitigation. The first stage effectively filters out such noise, keeping CSCG construction lightweight and traversal overhead negligible. This compact design also enables efficient graph-based reasoning across file operations. In terms of performance, *LinkGuard*$_{NF}$ incurs about 1,150

VII. Discussion

Compatibility. While our evaluation focuses on five representative versions of the Windows operating system, *LinkGuard* is designed to be broadly compatible with a wide range of real-world Windows environments, as discussed in Section V. Moreover, since LF attacks are fundamentally characterized by the cross-subject creation and traversal of link chains, we believe that the underlying principles of our prototype can be readily adapted to other platforms where such attacks are feasible, such as MacOS, Linux, and Android, provided that similar file system semantics exist.

Self-Security. To the best of our knowledge, *LinkGuard* does not introduce any known vulnerabilities, including LF attacks, and operates entirely within the kernel space. In addition, we also examined the prototype to verify the absence of typical memory safety issues. While an attacker may theoretically terminate the process of *LinkGuard* to compromise its defense by using advanced techniques such as *Bring Your Own Vulnerable Driver* (BYOVD) [85] or other undisclosed kernel-level exploits, such capabilities fall outside our assumed threat model. We consider this class of adversaries overly powerful for the scope of this work. Defending against BYOVD-based attacks aimed at disabling arbitrary security mechanisms is an orthogonal research problem and remains out of scope.

VIII. Related Work

LF Vulnerabilities Detection. In recent years, a growing body of research has focused on the analysis and detection of LF vulnerabilities. Pak et al. [86] and Zhao et al. [87] analyzed several real-world CVEs and their corresponding patches, primarily focusing on elucidating the typical exploitation process of LF attacks and summarizing the associated code-level fixes. While Lee and Lu et al. [19], [21] employed static code analysis to identify potential LF vulnerabilities in executable programs, Xiang and Yu et al. [18], [20] adopted a dynamic analysis approach by interacting with target programs to uncover such vulnerabilities at runtime. The vulnerabilities covered in these works also form part of our dataset, thereby indirectly supporting the development of *LinkGuard*.

File System Defenses. A range of defense mechanisms has been proposed to enhance the security and robustness of file systems. Lin et al. [88] applied the concept of mimicry-based defense in a distributed framework to protect file systems against various threats. For file-based TOCTOU [16] attacks, prior works [89], [90], [91], [92], [93], [94] primarily rely on static or dynamic analysis at the user-space level for detection, while others [95], [96], [97], [98], [99] focus on runtime prevention by monitoring kernel-space level access. Despite substantial efforts toward file system protection, to the best of our knowledge, no existing work has effectively addressed LF attacks within the Windows file system.

IX. Conclusion

This paper presents the first systematic study of defenses against Link Following (LF) attacks in the Windows file system. Motivated by observations from our empirical analysis, we propose *LinkGuard*, a lightweight, state-aware runtime guard against LF attacks. To the best of our knowledge, *LinkGuard* is the first approach capable of effectively mitigating LF attacks targeting Windows systems. We evaluate a prototype implementation on five representative Windows systems to validate its compatibility. On a dataset of 70 real-world vulnerabilities, *LinkGuard* successfully mitigates all single-step attacks and 95.45% of multi-step attacks, with zero false positives on benign file operations. On average, it introduces 1% overhead in microbenchmarks and 3.4% overhead in real-world application workloads, with a negligible 5 ms latency on benign file operations.

Acknowledgement

We thank the anonymous reviewers for the helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (62172105, U2436207) and the Shanghai Pilot Program for Basic Research - Fudan University 21TQ1400100 (21TQ012). Yuan Zhang is the corresponding author.

Ethics Considerations

This work raises no ethical concerns. All experiments, including the evaluation of *LinkGuard* on real-world LF vulnerabilities and benign applications, were conducted entirely within isolated, locally controlled test environments. No interaction with production systems, third-party infrastructure, or user data occurred during the development, testing, or evaluation of our system. The dataset used consists solely of publicly disclosed vulnerabilities, and all program traces were collected on test instances under our full control.

References

[1] Microsoft, "Symbolic links," https://learn.microsoft.com/en-us/windows/win32/fileio/symbolic-links, 2023.

[2] Delinea, "Windows privilege escalation: A comprehensive guide," https://delinea.com/blog/windows-privilege-escalation, 2025.

[3] "Cwe-200: Exposure of sensitive information to an unauthorized actor," https://cwe.mitre.org/data/definitions/200.html, 2024, mITRE, Common Weakness Enumeration.

[4] "Denial of service (dos)," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/denial-of-service, 2025.

[5] Wikipedia, "Microsoft windows," https://en.wikipedia.org/wiki/Microsoft_Windows, 2025, accessed: 2025-07-21.

[6] MITRE Corporation, "CVE - Common Vulnerabilities and Exposures," https://cve.mitre.org/, 2025, accessed: July 2025.

[7] S. Chari, S. Halevi, and W. Z. Venema, "Where do you want to go today? escalating privileges by pathname manipulation." in *NDSS*. Citeseer, 2010.

[8] M. Seaborn, "Plash: tools for practical least privilege," 2008.

[9] "I/o request packets (irps)," https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/i-o-request-packets, 2024.

[10] "About file system filter drivers," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/about-file-system-filter-drivers, 2024.

[11] "Ntfs junctions," https://learn.microsoft.com/en-us/windows/win32/fileio/hard-links-and-junctions, 2023.

[12] "Dosdevice symbolic links," https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-ms-dos-device-names, 2023.

[13] "Windows object namespaces," https://learn.microsoft.com/en-us/windows/win32/termserv/kernel-object-namespaces, 2023.

[14] J. Forshaw, "Windows Exploitation Tricks: Exploiting Arbitrary Object Directory Creation for Local Elevation of Privilege," https://googleprojectzero.blogspot.com/2018/08/windows-exploitation-tricks-exploiting.html, 2018.

[15] "File opportunistic locks," https://learn.microsoft.com/en-us/windows/win32/fileio/opportunistic-locks, 2023.

[16] "Cwe-367: Time-of-check time-of-use (toctou) race condition," https://cwe.mitre.org/data/definitions/367.html, 2024.

[17] "standard user account in windows," https://learn.microsoft.com/en-us/windows-server/remote/multipoint-services/create-a-standard-user-account, 2025.

[18] B. Xiang, Y. Zhang, F. Liu, H. Huang, Z. Lin, and M. Yang, "Pig in a poke: Automatically detecting and exploiting link following vulnerabilities in windows file operations," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

[19] Y.-T. Lee, H. Vijayakumar, Z. Qian, and T. Jaeger, "Static detection of filesystem vulnerabilities in android systems," *arXiv preprint arXiv:2407.11279*, 2024.

[20] C. Yu, Y. Xiao, J. Lu, Y. Li, Y. Li, L. Li, Y. Dong, J. Wang, J. Shi, D. Bo *et al.*, "File hijacking vulnerability: The elephant in the room," in *Proceedings of the Network and Distributed System Security Symposium*, 2024.

[21] J. Lu, F. Gu, Y. Wang, J. Chen, Z. Peng, and S. Wen, "Static detection of file access control vulnerabilities on windows system," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 16, p. e6004, 2022.

[22] The Linux Kernel Community, "Linux kernel documentation: Filesystem sysctl parameters," https://docs.kernel.org/admin-guide/sysctl/fs.html, 2025, accessed: Oct. 28, 2025.

[23] A. A. Basu, "Your name is my name: Attacking and defending programs from name collisions," Ph.D. dissertation, The Pennsylvania State University, 2025.

[24] GNU, "The GNU C Library (glibc)," https://www.gnu.org/software/libc/, 2024.

[25] Y. Bassil, "Windows and linux operating systems from a security perspective," *arXiv preprint arXiv:1204.0197*, 2012.

[26] "Access control," https://learn.microsoft.com/en-us/windows/security/identity-protection/access-control/access-control, 2025.

[27] W. Zhang, P. Liu, and T. Jaeger, "Analyzing the overhead of file protection by linux security modules," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 393–406.

[28] "National Vulnerability Database," https://nvd.nist.gov/, 2024. [Online]. Available: https://nvd.nist.gov/

[29] "Cwe-59: Link following error," https://cwe.mitre.org/data/definitions/59.html. [Online]. Available: https://cwe.mitre.org/data/definitions/59.html

[30] "Cwe-63: Windows path link problems," https://cwe.mitre.org/data/definitions/63.html. [Online]. Available: https://cwe.mitre.org/data/definitions/63.html

[31] "Cwe-64: Improper handling of symbolic links in windows," https://cwe.mitre.org/data/definitions/64.html. [Online]. Available: https://cwe.mitre.org/data/definitions/64.html

[32] Microsoft, "Process monitor (procmon)," https://learn.microsoft.com/en-us/sysinternals/downloads/procmon, 2025.

[33] Hex-Rays, "Hex-Rays IDA Pro," https://www.hex-rays.com/, 2020. [Online]. Available: https://www.hex-rays.com/

[34] Google, "Bindiff: A tool for binary comparison," https://github.com/google/bindiff, 2025.

[35] "Access control lists (acls)," https://learn.microsoft.com/en-us/windows/win32/secauthz/access-control-lists. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/secauthz/access-control-lists

[36] Intel Corporation, "Intel security advisory intel-sa-01066," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01066.html, 2024, accessed: July 29, 2025.

[37] Microsoft, "Usmt recognized environment variables," https://learn.microsoft.com/en-us/windows/deployment/usmt/usmt-recognized-environment-variables, 2024, accessed: Oct. 23, 2025. [Online]. Available: https://learn.microsoft.com/en-us/windows/deployment/usmt/usmt-recognized-environment-variables

[38] ——, "Default group permissions for ProgramData folder," https://learn.microsoft.com/en-us/answers/questions/4328564/default-group-permissions-for-programdata-folder, 2024.

[39] MDSec, "Cve-2024-21111: Local privilege escalation in oracle virtualbox," https://www.mdsec.co.uk/2024/04/cve-2024-21111-local-privilege-escalation-in-oracle-virtualbox/, 2024, accessed: 2025-06-15.

[40] "Policy csp - configureredirectionguardpolicy," https://learn.microsoft.com/en-us/windows/client-management/mdm/policy-csp-printers#configureredirectionguardpolicy. [Online]. Available: https://learn.microsoft.com/en-us/windows/client-management/mdm/policy-csp-printers#configureredirectionguardpolicy

[41] National Vulnerability Database, "Cve-2025-3617," https://nvd.nist.gov/vuln/detail/CVE-2025-3617, 2025.

[42] S. Motiee, K. Hawkey, and K. Beznosov, "Do windows users follow the principle of least privilege? investigating user account control practices," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*, 2010, pp. 1–13.

[43] "Impersonation - windows process security," https://learn.microsoft.com/en-us/windows/win32/com/impersonation, 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/com/impersonation

[44] "CVE-2024-44193," National Vulnerability Database (NVD), 2024. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-44193

[45] Microsoft, "LocalSystem Account," https://learn.microsoft.com/en-us/windows/win32/services/localsystem-account, 2023.

[46] ——, "LocalService Account," https://learn.microsoft.com/en-us/windows/win32/services/localservice-account, 2023.

[47] "Getfileinformationbyhandle function (fileapi.h) - win32 apps," https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-getfileinformationbyhandle, 2025.

[48] "Getfinalpathnamebyhandlea function (fileapi.h) - win32 apps," https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-getfinalpathnamebyhandlea, 2025.

[49] "File path formats on windows - unc paths," https://learn.microsoft.com/en-us/dotnet/standard/io/file-path-formats#unc-paths, 2025.

[50] "Irp_mj_create," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/irp-mj-create, 2025.

[51] "Irp_mj_write," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/irp-mj-write, 2025.

[52] "File_rename_information structure," https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/ns-ntifs-_file_rename_information, 2025.

[53] "Irp_mj_set_information," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/irp-mj-set-information, 2025.

[54] "File_flag_delete_on_close," https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea#parameters, 2025.

[55] "Irp_mj_set_security," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/irp-mj-set-security, 2025.

[56] "Irp_mj_file_system_control," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/irp-mj-file-system-control, 2025.

[57] "Fsctl_request_oplock," https://learn.microsoft.com/en-us/windows/win32/fileio/fsctl-request-oplock, 2025.

[58] Microsoft, "Access Tokens," https://learn.microsoft.com/en-us/windows/win32/secauthz/access-tokens, 2023, accessed: July 31, 2025.

[59] "Impersonation," https://learn.microsoft.com/en-us/windows/win32/com/impersonation, 2025.

[60] "Replace a process level token," https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-10/security/threat-protection/security-policy-settings/replace-a-process-level-token, 2025.

[61] Microsoft, "PreOperation Callback Routines," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/writing-preoperation-callback-routines, 2023.

[62] ——, "PostOperation Callback Routines," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/writing-postoperation-callback-routines, 2023.

[63] "Shortcut," https://en.wikipedia.org/wiki/Shortcut_(computing)#Microsoft_Windows. [Online]. Available: https://en.wikipedia.org/wiki/Shortcut_(computing)#Microsoft_Windows

[64] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[65] R. Russon and Y. Fledel, "Ntfs documentation," *Recuperado el*, vol. 1, p. 2, 2004.

[66] "Building arm64 drivers," https://learn.microsoft.com/en-us/windows-hardware/drivers/develop/building-arm64-drivers, 2025.

[67] D. Capps and W. Norcott, "Iozone filesystem benchmark," http://www.iozone.org.

[68] Microsoft, "Diskspd: A storage performance tool," https://github.com/microsoft/diskspd, 2023.

[69] Apache Software Foundation, "Apache http server," https://httpd.apache.org/, accessed: July 9, 2025.

[70] Microsoft Corporation, "Microsoft sql server," https://www.microsoft.com/en-us/sql-server, 2024, accessed: 2025-07-15.

[71] "Winscp: Free sftp, scp, s3 and ftp client for windows," https://winscp.net/, accessed: July 9, 2025.

[72] "7-zip: A free file archiver," https://www.7-zip.org/, accessed: July 9, 2025.

[73] D. Marjamäki, "Cppcheck: A static analysis tool for c/c++," 2023. [Online]. Available: https://cppcheck.sourceforge.io

[74] D. A. Wheeler, "Flawfinder: A security-oriented static code analysis tool," 2023. [Online]. Available: https://github.com/david-a-wheeler/flawfinder

[75] "Using i/o control codes (ioctls)," https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/using-i-o-control-codes, 2025.

[76] "Remote procedure calls (rpc)," https://learn.microsoft.com/en-us/windows/win32/rpc/remote-procedure-calls, 2024.

[77] Z. D. Initiative, "Abusing arbitrary file deletes to escalate privilege and other great tricks," https://www.zerodayinitiative.com/blog/2022/3/16/abusing-arbitrary-file-deletes-to-escalate-privilege-and-other-great-tricks, 2024.

[78] Google, "Cve-2023-2939: Insufficient validation in file system api in google chrome on windows," https://issues.chromium.org/issues/40063745, 2023.

[79] Google, "Download and install Google Chrome," https://support.google.com/chrome/answer/95346?hl=en, 2024.

[80] "Cve-2024-44193 proof-of-concept exploit," https://github.com/mbog14/CVE-2024-44193, 2025.

[81] Apple, "iTunes for Windows," https://support.apple.com/en-us/106372, 2024.

[82] Z. D. Initiative, "Cve-2025-21373," https://www.zerodayinitiative.com/blog/2025/2/11/the-february-2025-security-update-review, 2025.

[83] Apache, "Apache http server benchmarking tool," https://httpd.apache.org/docs/current/programs/ab.html, 2024, accessed: 2025-07-17.

[84] HammerDB, "HammerDB Benchmarks," https://www.hammerdb.com/benchmarks.html, 2024.
[85] A. Monzani, A. Parata, A. Oliveri, S. Aonzo, D. Balzarotti, A. Lanzi *et al.*, "Unveiling byovd threats: Malware's use and abuse of kernel drivers," in *Network and Distributed System Security (NDSS) Symposium 2026*. NDSS, 2026, pp. 1–19.
[86] T. B. B. Pak, "What's under the hood? root cause and patch analyses of elevation of privilege vulnerabilities in the windows operating system," in *IRC Conference on Science, Engineering and Technology*. Springer, 2025, pp. 168–180.
[87] B. Zhao, W. Feng, Q. Guo, Y. Sun, F. Gu, B. Zhang, X. Gong, and H. Li, "Favdisco: Modeling and discovering file access vulnerabilities," *ACM Transactions on Software Engineering and Methodology*, 2025.
[88] Z. Lin, K. Li, H. Hou, X. Yang, and H. Li, "Mdfs: A mimic defense theory based architecture for distributed file system," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 2670–2675.
[89] D. Dean and A. J. Hu, "Fixing races for fun and profit: how to use access (2)." in *USENIX security symposium*, 2004, pp. 195–206.
[90] C. Ko, G. Fink, and K. Levitt, "Automated detection of vulnerabilities in privileged programs by execution monitoring," in *Tenth Annual Computer Security Applications Conference*. IEEE, 1994, pp. 134–144.
[91] B. Goyal, S. Sitaraman, and S. Venkatesan, "A unified approach to detect binding based race condition attacks," in *Int'l Workshop on Cryptology & Network Security (CANS)*, 2003, p. 16.
[92] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 15–pp.
[93] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva, "Portably preventing file race attacks with user-mode path resolution," *IBM Res., Yorktown Heights, NY, USA, Tech. Rep. RC24572 (W0806-008)*, 2008.
[94] A. Aggarwal and P. Jalote, "Monitoring the security health of software systems," in *2006 17th International Symposium on Software Reliability Engineering*. IEEE, 2006, pp. 146–158.
[95] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman, "{RaceGuard}: Kernel protection from temporary file race vulnerabilities," in *10th USENIX Security Symposium (USENIX Security 01)*, 2001.
[96] C. Ko and T. Redmond, "Noninterference and intrusion detection," in *Proceedings 2002 IEEE Symposium on Security and Privacy*. IEEE, 2002, pp. 177–187.
[97] E. Tsyrklevich and B. Yee, "Dynamic detection and prevention of race conditions in file accesses," in *12th USENIX Security Symposium (USENIX Security 03)*, 2003.
[98] K.-S. Lhee and S. J. Chapin, "Detection of file-based race conditions," *International Journal of Information Security*, vol. 4, no. 1, pp. 105–119, 2005.
[99] P. Uppuluri, U. Joshi, and A. Ray, "Preventing race condition attacks on file-systems," in *Proceedings of the 2005 ACM symposium on Applied computing*, 2005, pp. 346–353.

APPENDIX A
METHODOLOGY APPENDIX

A. File ACL Hardening

This example shows a typical ACL-hardening strategy, where developers remove all access rules not associated with `SYSTEM` and disable inheritance. Notably, the use of `SetAccessControl` and `SetAccessRuleProtection` reflects an upper-layer abstraction over the native `SetSecurityFile` API.

```
public static void SecureDataDirectory() {
    string path = @"C:\ProgramData\Intel\Intel Extreme Tuning Utility";
    // Get current ACL
    DirectoryInfo dirInfo = new DirectoryInfo(path);
    DirectorySecurity acl = dirInfo.GetAccessControl();
    // Remove all access rules not belonging to SYSTEM
    foreach (FileSystemAccessRule rule in acl.GetAccessRules() {
        string identity = rule.IdentityReference.Value;
        if (!identity.Contains("SYSTEM")) {
            acl.RemoveAccessRule(identity, rule.FileSystemRights, rule.AccessControlType);
        }
    }
    // Protect ACL from inheritance and apply updated rules
    acl.SetAccessRuleProtection(isProtected: true, preserveInheritance: false);
    dirInfo.SetAccessControl(acl);
}
```

Fig. 9: **Countermeasure of CVE-2024-21835: decompiled code illustrating File ACL Hardening.**

B. Code Example Code for Enabling Redirection Guard

Listing 1 shows a practical example of enabling Redirection Guard at the process level using the `SetProcessMitigationPolicy` API.

```
ULONG SetProcessMitigationMode(
    _In_ MODE_OPTION Option
) {
    ULONG Error = ERROR_SUCCESS;
    PROCESS_MITIGATION_REDIRECTION_TRUST_POLICY
        policy = {0};

    if (Option == MODE_OPTION::Enforce) {
        policy.EnforceRedirectionTrust = 1;
    } else if (Option == MODE_OPTION::Audit) {
        policy.AuditRedirectionTrust = 1;
    }

    if (!SetProcessMitigationPolicy(
            ProcessRedirectionTrustPolicy,
            &policy,
            sizeof(policy))) {
        Error = GetLastError();
        LogError(Error, "Failed to set
            mitigation policy");
    }

    return Error;
}
```

Listing 1: Code example: enabling Redirection Guard via `SetProcessMitigationPolicy`

C. File Name Randomization.

This countermeasure introduces a randomly generated subdirectory (via `GenerateRandomName`) in the temporary path to prevent attackers from predicting and constructing link chains at known locations. As shown in lines 9–10 of the patch in Figure 10, the original two-level path (`TempPath/FileName`) is replaced with a three-level path (`TempPath/RandomName/FileName`), thereby inserting an unpredictable directory layer to mitigate LF attacks.

```
1  void CreateLogFile(char *FileName) {
2    …
3    char LogFilePath[MAX_PATH];
4    char TempPath[MAX_PATH];
5    char RandomName[16];
6   // Generate unpredictable subdir
7  + GenerateRandomName(RandomName,
8  "%08X.%03X");
   -  snprintf(LogFilePath, MAX_PATH, "%s\\%s",
   TempPath, FileName);
9  + snprintf(LogFilePath, MAX_PATH, "%s\\%s\\
10 %s", TempPath, RandomName, FileName);
11   CreateFile(LogFilePath);
12 }
```

Fig. 10: **Countermeasure of CVE-2025-3617: pseudocode illustrating File Name Randomization.**

D. File Path Validation

In the patch for `CVE-2023-35342`, the developers mitigate the vulnerability by invoking `GetFileInformationByHandle` API to retrieve file metadata and inspecting the number of associated symbolic links, thereby identifying and blocking potentially unsafe access.

```
1  void __fastcall  InitTraceFile (TrcLib *this) {
2        wchar_t * FileName = "Wiatrace.log";
3  +     if (!TrcLib::IsPathALink(FileName)){
4           CreateFileW(FileName, ⋯);
5  +     }
6  }
```

(a) Patched Code with File Path Validation

```
1  bool IsPathALink (const wchar_t * FileName){
2        HANDLE FileW = CreateFileW(FileName, ⋯);
3        if (GetFileInformationByHandle(FileW, &FileInfo) &&
4  FileInfo.NumberOfLinks > 1){
5           return TRUE;
6     }
7        return FALSE;
8  }
```

(b) Implementation Details of `IsPathALink`

Fig. 11: The File Path Validation Countermeasure Analysis in CVE-2023-35342

TABLE VI: **Detailed Breakdown of Dataset-α. This table presents the attack types and corresponding countermeasure classifications for each CVE. We provide statistics on the excluded cases. In total, 82 CVEs were excluded, including 61 without accessible execution environments and 21 with available environments but insufficient technical details to reproduce the exploitation.**

#	CVE ID	Attack Type	Countermeasures
1	CVE-2018-12148	S	C1
2	CVE-2018-12168	S	C1
3	CVE-2018-6261	M	C1
4	CVE-2019-11114	S	C1
5	CVE-2019-13382	M	C1
6	CVE-2020-0668	S	C1
7	CVE-2020-11474	S	C5
8	CVE-2020-9682	S	C6
9	CVE-2021-25261	M	C1
10	CVE-2021-26862	M	C5
11	CVE-2021-28313	S	C6
12	CVE-2021-43237	S	C6
13	CVE-2022-22262	M	C1
14	CVE-2022-28225	M	C1
15	CVE-2022-30523	M	C1
16	CVE-2022-32450	S	C5
17	CVE-2022-38604	M	C6
18	CVE-2022-38699	S	C1
19	CVE-2022-41120	S	C1, C6
20	CVE-2022-44704	M	C1
21	CVE-2023-20178	M	C1
22	CVE-2023-21752	M	C6
23	CVE-2023-28868	M	C5
24	CVE-2023-28869	S	C5
25	CVE-2023-28892	S	C6
26	CVE-2023-29343	M	C1, C6
27	CVE-2023-32163	S	C2, C5
28	CVE-2023-35342	M	C6
29	CVE-2023-36047	M	C6
30	CVE-2023-36723	S	C3
31	CVE-2023-36874	M	C6
32	CVE-2023-42099	M	C1
33	CVE-2023-50915	S	C1
34	CVE-2024-11399	M	C1
35	CVE-2024-11857	S	C1
36	CVE-2024-20656	S	C5
37	CVE-2024-21111	M	C2
38	CVE-2024-21447	M	C6
39	CVE-2024-26238	M	C6
40	CVE-2024-27460	S	C6
41	CVE-2024-28916	M	C3
42	CVE-2024-30033	M	C6
43	CVE-2024-3037	S	C6
44	CVE-2024-35204	S	C1
45	CVE-2024-38022	S	C3
46	CVE-2024-38084	M	C3
47	CVE-2024-38393	S	C3
48	CVE-2024-43114	M	C1
49	CVE-2024-44193	M	C5
50	CVE-2024-45315	M	C5
51	CVE-2024-45316	M	C1
52	CVE-2024-49051	M	C4
53	CVE-2024-49107	M	C3
54	CVE-2024-6974	M	C1
55	CVE-2024-8404	M	C6
56	CVE-2024-8405	M	C6
57	CVE-2024-9871	M	C1
58	CVE-2025-0651	M	C3
59	CVE-2025-20099	S	C1
60	CVE-2025-21204	M	C1
61	CVE-2025-21347	S	C2
62	CVE-2025-24287	M	C1
63	CVE-2025-24327	M	C1
64	CVE-2025-25230	M	C1
65	CVE-2025-29975	M	C3
66	CVE-2025-32721	M	C3
67	CVE-2025-32817	M	C4, C5
68	CVE-2025-3617	M	C1
69	CVE-2025-48443	M	C1
70	CVE-2025-49680	S	C3

S: Single-step attacks. M: Multi-step attacks. **C1–C6 Countermeasures:** C1: ACL Hardening, C2: Secure Path Binding, C3: Redirection Guard, C4: Name Randomization, C5: Least Privilege, C6: File Path Validation.

E. Detailed Breakdown of Dataset-α

APPENDIX B
ARTIFACT APPENDIX

This is the artifact evaluation appendix for this paper. In this paper, we present *LinkGuard*, a lightweight state-aware runtime guard against LF attacks targeting Windows systems. The novelty of *LinkGuard* lies in its two-stage design: The first stage aims to improve defense efficiency by performing dynamic subject filtering, which monitors only file operations and associated subjects involved in the creation and following of link chains; The second stage applies FSM-based rule matching to defend LF attacks, ensuring effective and accurate defense precisely.

We evaluate the prototype across five representative Windows systems to validate its compatibility. On a dataset of 70 real-world vulnerabilities, *LinkGuard* successfully mitigates all single-step attacks and 95.45% of multi-step attacks, with zero false positives on benign operations. On average, *LinkGuard* only incurs 1% overhead in microbenchmarks and 3.4% overhead in real-world application workloads, while adding a negligible 5 ms latency on benign file operations.

A. Description & Requirements

1) How to access: The artifact can be downloaded from the Zenodo record[2]. All artifacts can be accessed by simply extracting the downloaded archive.

2) Hardware dependencies: We run our experiments on a VMware virtual machine hosted on an HP laptop running Windows 11. The artifacts can be used on either physical or virtual machines. It is recommended to configure the environment with at least 4 CPU cores, 16 GB RAM, and 200 GB of free disk space, running Windows 11 as the operating system.

3) Software dependencies: A Windows system is required for Artifact Evaluation. We recommend using Windows 11 24H2. The other software dependencies are listed below.

- **Building**. (1) Visual Studio 2022 is required to compile the executables and kernel drivers necessary for *LinkGuard*, along with the corresponding Windows Driver Kit (WDK)[3] and Windows SDK[4]. (2) CMake[5] is used to leverage the build toolchain provided by Visual Studio, producing the final executable and driver binaries.
- **Running**. (1) VMware Workstation[6] is required to load the provided virtual machine, which encapsulates the complete execution environment, including the prepared dataset and all necessary dependencies for running *LinkGuard*.

4) Benchmarks: The artifact provided virtual machine includes two pre-installed vulnerable applications and their corresponding exploit scripts; reproduction instructions are documented in the `quick-reproduction-guide.pdf`. When *LinkGuard* is deployed and activated on this VM, attempts to reproduce the LF attacks using the supplied scripts are detected by *LinkGuard* and prevented, causing the exploit attempts to fail.

B. Artifact Installation & Configuration

Given that deploying *LinkGuard* requires numerous Windows-specific dependencies and intricate system configurations, such as enabling unsigned driver installation, we provide a `Quick-Deployment-Guide.pdf`. We recommend using the `Quick-Deployment-Guide`, which offers a preconfigured virtual machine environment integrating *LinkGuard* and all necessary datasets, to facilitate evaluation.

C. Major Claims

LinkGuard's defense against Link-Following attacks primarily achieves the following two goals:

- (G1): *LinkGuard* accurately detects and mitigate LF attacks with low runtime overhead, minimal runtime latency, and high compatibility, thereby ensuring its practicality for real-world deployment.
- (G2): *LinkGuard* should effectively mitigate both known types of LF attacks, namely, single-step and multi-step attacks. Moreover, the design should facilitate ease of extension, allowing rapid adaptation to mitigate newly identified LF attacks, that is, by incorporating their characteristic behaviors into FSM rules for timely defense.

D. Evaluation

LinkGuard successfully mitigates 97.1% (68/70) of LF attacks in our evaluation, including all single-step attacks (26/26) and 95.45% (42/44) of multi-step attacks.

Due to responsible disclosure constraints and the fact that some exploit scripts were independently developed by the authors, we cannot distribute the full set of affected software and attack scripts. Instead, the artifact includes a virtual machine preloaded with two representative vulnerable applications and their corresponding exploit scripts (see `quick-reproduction-guide.pdf`). Deploying and activating *LinkGuard* on this VM prevents the supplied LF attack reproductions, thereby demonstrating the practical effectiveness of our defense.

E. Notes

Note that although we provide a preconfigured VM for quick deployment of *LinkGuard*, Windows driver-installation semantics prevent *LinkGuard* from being activated out-of-the-box within that VM. Detailed instructions for running the VM and activating *LinkGuard* are provided in the `Quick Deployment Guide.pdf`.

[2] https://doi.org/10.5281/zenodo.17481221

[3] https://learn.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk#download-icon-for-sdk-step-2-install-sdk

[4] https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/

[5] https://cmake.org/download/

[6] https://www.vmware.com/products/desktop-hypervisor/workstation-and-fusion