RESEARCH-ARTICLE

Error Messages to Fuzzing: Detecting XPS Parsing Vulnerabilities in Windows Printing Components

YUNPENG TIAN, Huazhong University of Science and Technology, Wuhan, Hubei, China

FENG DONG, Huazhong University of Science and Technology, Wuhan, Hubei, China

JUNHAI WANG, Huazhong University of Science and Technology, Wuhan, Hubei, China

MU ZHANG, The University of Utah, Salt Lake City, UT, United States

ZHINIANG PENG, Huazhong University of Science and Technology, Wuhan, Hubei, China

ZESEN YE

View all

Open Access Support provided by:

The Hong Kong Polytechnic University

Huazhong University of Science and Technology

The University of Utah

PDF Download
3719027.3744807.pdf
27 February 2026
Total Citations: 0
Total Downloads: 1754

Published: 22 November 2025

Citation in BibTeX format

CCS '25: ACM SIGSAC Conference on Computer and Communications Security
October 13 - 17, 2025
Taipei, Taiwan

Conference Sponsors:
SIGSAC

Error Messages to Fuzzing: Detecting XPS Parsing Vulnerabilities in Windows Printing Components

Yunpeng Tian[*][†][‡]
yunpeng.tian@polyu.edu.hk
Huazhong University of Science and Technology
Wuhan, China

Feng Dong[*][‡]
dongfeng@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Junhai Wang[‡]
junhaiwang@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Mu Zhang
muzhang@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

Zhiniang Peng[‡][§]
jiushigujiu@gmail.com
Huazhong University of Science and Technology
Wuhan, China

Zesen Ye
whhhitc@gmail.com
Beijing CyberKunlun Technology Co., Ltd
Beijing, China

Xiapu Luo
csxluo@comp.polyu.edu.hk
The Hong Kong Polytechnic University
Hong Kong, China

Haoyu Wang[‡][§]
haoyuwang@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Abstract

Windows printing services remain a notable vector for attacks. Previous studies have predominantly targeted vulnerabilities within various control aspects of printing services, such as spooler services and firmware updates. Yet, we contend that an essential aspect of data processing—the document parser within printer drivers—has been overlooked in past research. We present a coverage-based fuzzing system, PrintXPSurge, specifically crafted to detect weaknesses in the XPS printer driver's parsing function. To craft semantically correct XPS files, we leverage a *large language model-assisted repair approach* to automate the creation of semantically correct XPS files that comply with necessary constraints. To ensure our fuzzing process effectively interacts with the XPS printer driver, we develop a *progressive state reconstruction* method that addresses individual dependency requirements across the entire printing service workflow. Furthermore, when a crash is detected, we employ backtracing to confirm its origin in the XPS parser, isolating it from other components in the pipeline. Our evaluation reveals that PrintXPSurge surpasses existing top Windows fuzzers in performance, successfully identifying 102 bugs in 10 drivers from major brands, including **17 zero-day** vulnerabilities confirmed by Microsoft and third-party vendors.

CCS Concepts

• **Security and privacy → Operating systems security**;

Keywords

Windows Security, Vulnerability Discovery, Printer Security

ACM Reference Format:
Yunpeng Tian, Feng Dong, Junhai Wang, Mu Zhang, Zhiniang Peng, Zesen Ye, Xiapu Luo, and Haoyu Wang. 2025. Error Messages to Fuzzing: Detecting XPS Parsing Vulnerabilities in Windows Printing Components. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744807

1 Introduction

Windows printing services continue to be a significant attack vector. Notable exploits, such as the widely recognized Print Nightmare vulnerability [7] and the MS10-061 bug [32] associated with the Stuxnet attacks [10], have demonstrated the potential for attackers to run arbitrary code and gain complete control over affected systems. Print bugs played a role in Stuxnet and Print Nightmare, and account for 9% of all Windows cases reported to MSRC [39], highlighting their significance in the security landscape. Despite the numerous patches that have been released for critical components, this type of printing vulnerabilities not only present a complex repair challenge but also face the potential of being circumvented post-repair, leading to persistent attack surfaces even a

[*]Both authors contributed equally to this research.
[†]The Hong Kong Polytechnic University, Hong Kong, China
[‡]Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.
[§]Co-corresponding author.

decade later [56]. Similarly, academic research [8] has highlighted how weaknesses in printer firmware can be leveraged to introduce covert malware that performs network reconnaissance and data theft. Additionally, the widespread availability of network printers offers attackers convenient avenues for mounting attacks, which might be challenging through other means, further compounding the security challenges.

Existing research has focused on finding vulnerabilities in various *control units* of printing services, including spooler services and firmware update mechanisms. However, we argue that a critical element in *data processing* – the document parser embedded in printer drivers – has not received sufficient attention in prior investigations. In fact, exploiting vulnerabilities in the parsing components might present a lower barrier than targeting elements such as task schedulers or embedded firmware. This is because an attacker can utilize *legitimate* and *openly accessible* pathways to stealthily submit an apparently harmless input file, avoiding the more overtly dubious activities associated with installing a malicious driver or altering firmware components.

To tackle this problem, we suggest employing fuzzing techniques to automatically and systematically identify vulnerabilities in the parsing component of Windows printer drivers. Specifically, our attention is directed towards the standard driver model [45] introduced since Windows 8, known as the XPS printer driver (XPSDrv [48]). This model is noteworthy for its backward compatibility with the older Microsoft printer driver architecture, GDI, and its ability to support both PostScript and non-PostScript printers and plotters.

Currently, XPS is widely used in Windows printers, and most printing tasks in Windows involve some form of XPS conversion [39]. Regardless of the file type being printed (e.g., PDF), it will ultimately be converted to XPS, and the process handling this task (PrintFilterPipelineSVC) is a source of numerous memory corruption vulnerabilities.

Many efforts [2–4, 13, 14, 17, 50] have been made to develop grammar-aware fuzzers for testing applications that process highly structured documents. These tools are capable of automatically generating well-structured document files, such as XML, based on known specifications [4], grammar inference [4], or machine learning techniques [14]. However, these tools are not directly applicable to our specific challenge. The reason is that a valid XPS file must adhere not only to the basic XML syntax but also to the particular semantics and dependencies unique to XPS. For instance, the value assigned to a "`Path.Data`" attribute adheres to a defined syntax within vector graphics; specific attributes are required to be sequenced in accordance with the rendering workflow.

Moreover, much of the existing research has concentrated on applying grammar-aware fuzzing techniques to open-source software. In contrast, **our challenge lies in enabling fuzzing for the XPS driver within a closed-source environment**. A few previous studies [5, 15, 18, 21, 25, 54] have attempted to apply fuzzing techniques to Windows applications. Notably, the state-of-the-art Windows fuzzing tool WINNIE [25] has leveraged dynamic tracing to extract control flows within target applications, facilitating the automatic creation of fuzzing harnesses by uncovering hidden code dependencies.

Yet adopting a similar strategy for the XPS driver, which is more internally oriented than many externally-facing applications, presents additional complexities. This is largely due to the intricate *inter-program* interactions and *data/control dependencies* inherent in the printing service workflow. The deep dependency chain across multiple printing modules makes it extremely difficult, if not impossible, to use a single harness program to satisfy all necessary dependencies at the entry point of the printing workflow.

Overview of PrintXPSurge. To overcome these hurdles, we introduce a coverage-based fuzzing system, PrintXPSurge, designed to identify flaws in the XPS printer driver's parsing mechanism. To craft semantically correct XPS files, we take into account both general XML standards and specific XPS constraints. To ensure our fuzzing process effectively interacts with the XPS printer driver, we develop a *progressive state reconstruction* method that addresses individual dependency requirements across the entire printing service workflow. To improve the accuracy of XPS files, we introduce an *LLM-assisted repair approach* for correcting the format of XPS files. Then, to verify that a detected crash is precisely caused by the interested XPS parser, rather than other components of the pipeline, we perform backtracing to identify the root cause of the observed crash. To the best of our knowledge, we are the first to tackle the Window XPS driver fuzzing problem, in a *semantics* and *workflow* aware fashion. Our assessment has confirmed the efficacy of our input and harness generation techniques, allowing PrintXPSurge to outperform current leading Windows fuzzers and uncover **17 zero-day vulnerabilities**, which Microsoft and third-party vendors have validated.

Summary. PrintXPSurge makes the following contributions:

- Our research pioneered the investigation of the XPS parsing module in Windows printers, revealing its significant vulnerability to cyber threats.
- We created PrintXPSurge[1], a novel tool designed to identify vulnerabilities in the XPS component of Windows printers. PrintXPSurge proficiently scrutinizes printer binary files, detecting a variety of critical issues such as Use-After-Free (UAF), null pointer dereferences, and memory corruptions (Section 3).
- Using PrintXPSurge to examine XPS drivers on Windows platforms, we identified 102 bugs across ten leading-brand printers. Among these, 13 zero-days were found in official Microsoft components and were assigned CVEs by Microsoft (Section 4).

2 Background

In this section, we will first review the basics of printer models and their XPS print architecture and documentation. Then, we will describe attack scenarios for XPS printers.

2.1 XPS Printing Framework

The Windows printing architecture is a complex system involving various components and services, designed to ensure compatibility and enhance the development of printing applications and drivers. This architecture includes a print spooler program and a series of print drivers, with the print spooler managing and coordinating

[1]We open source part of the data and non-commercial code at https://github.com/PrinterRepo/PrintXPSurge to facilitate research in this area.

the operations of these drivers. The print spooler is a crucial component responsible for retrieving the appropriate printer drivers and loading them. For more detailed information on the Windows printing architecture, refer to the Appendix A.

In the Windows operating system, there are two main graphics interfaces used for printing: XPS and GDI. XPS (XML Paper Specification), developed by Microsoft starting from Windows Vista, is intended to replace the aging GDI (Graphics Device Interface). The XPS framework offers a more modern and flexible approach to rendering print jobs. Under the XPS framework, applications do not directly invoke drivers; instead, print data is generated through the operating system's XPS service to produce a universal XPS file (see Appendix B) intended for printing.

The main components of an XPS driver system include: Filter, Filter pipeline, Filter pipeline manager, Filter configuration file, XPS Spooler Process, and WPF application. The render module of an XPSDrv printer driver contains filters that process and render the contents of the XPS spool file for output to the printer. This rendering component is responsible for converting the drawing commands received from the application into the command data required by the printer and then sending it to the printer for rendering. Figure 1 shows the workflow of the entire rendering module and highlights the XPS-related components.

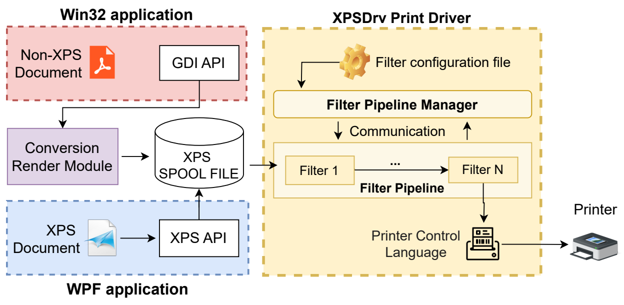

Figure 1: XPS Print Path

The `Filter Pipeline Manager` is responsible for data transformation. It can read and write page markup, access the content of other components, create new components, and associate them with pages. Additionally, it can generate entirely new documents and pages, dissociate relationships between pages, enumerate resources on a page, and operate with both XPS and stream-based filters. In the system driver, print files are ingested as XPS stream files and managed via a configuration file among various filters. This configuration file, structured in XML, delineates the taxonomy of filters, specifies the interfaces of each filter, and stipulates the input/output formats pertinent to each filter. For flexibility and reuse, each filter performs a specific print processing function. For example, one filter might apply a watermark, while another performs accounting. Ultimately, the file is converted into a PS or PCL language recognized by most printers through the `MSxpsPCL6` or `MSxpsPS` filter and delivered to the printer for printing via the print stream interface.

2.2 Attack Scenario

The components and functionalities of the Windows printing architecture are quite complex and embody certain hazardous designs with potential risks. Compared to vulnerabilities in other system modules (e.g., `Win32k.sys`), those associated with printers can often be exploited more simply and directly, heightening their allure as attack vectors.

Penetrating an organization's network is orthogonal to escalating privileges within its local network. Even after gaining initial access, external attackers must still exploit vulnerabilities to compromise privileged internal servers. In the past, during a traditional full-scale attack on the Windows operating system, attackers would first exploit user-level vulnerabilities (e.g., in browsers like Edge) to bypass application mitigations and gain initial access to the system through remote code execution (RCE). Once they have access to the system, attackers typically seek a primitive capable of arbitrary memory read/write operations in the kernel. This can be achieved through objects such as `Bitmap` [52] or `Palette` [9], which are components of `Win32k.sys`. `Win32k.sys` is a critical module in the Windows kernel responsible for handling user interface and graphical display functionalities. By leveraging this primitive, attackers can access the token of the system process with PID 4, which is usually a core component of the Windows operating system and possesses the highest level of privileges. By reading and replicating the security token of this process, attackers can extend these elevated privileges to other processes, thereby achieving privilege escalation. To successfully execute such attacks, attackers must satisfy two key prerequisites: the ability to perform arbitrary memory reads and writes within kernel memory spaces.

In stark contrast, the exploitation of printer vulnerabilities does not rely on such complexity. Various Windows components related to printing, like `spoolsv.exe` (the print spooler service) and print pipeline handlers, usually operate with high privilege levels. Thus, if vulnerabilities allow the execution of attacker code, these components are already running in a high-privilege context, enabling the attacker to directly assume said elevated privileges and carry out malicious activities. In these cases, the privilege escalation grants `SYSTEM` privileges rather than the more powerful kernel privilege. These actions may include, but are not limited to, deploying malware, tampering with system files, or undertaking other potentially destructive actions, posing a severe threat to system security.

In modern network environments, printing functionality commonly relies on network interactions to facilitate remote document printing. While this convenience improves efficiency, it also increases the risk of network attacks, particularly those that may enable attackers to gain control over remote devices. Although print services are typically confined to local networks, their broad accessibility makes them high-value targets. Attackers can exploit these services to escalate privileges without needing the complex prerequisites required for kernel-level exploits, making them highly attractive targets.

XPS drivers are crucial for various printing services, but they are not included by default in Windows. Users have to download and install them manually. Nevertheless, attackers can still find ways to circumvent this limitation through other methods. Starting with Vista, the "Point and Print" [41] feature allows users to install built-in print drivers through a print server without needing administrator privileges. Attackers can exploit the "Point and Print" feature to install vulnerable XPS printer drivers. By leveraging these vulnerabilities, even attackers with lower privileges can load malicious DLLs and achieve remote code execution. Notably, the well-known Windows Print Nightmare vulnerabilities CVE-2021-34527[1] and CVE-2021-1675 are examples of vulnerabilities that execute code in the Windows printing service through the loading of malicious DLLs. Specifically, CVE-2021-1675 allows attackers to bypass safety mechanisms in the `PfcAddPrinterDriver` function, enabling the installation of malicious drivers on a targeted print server. If the attacker can also control user accounts in the domain, they could further connect to the Domain Controller's Spooler service and exploit the same vulnerability to install malicious drivers on the Domain Controller, thereby gaining control of the entire domain environment. Attackers successful in exploiting this vulnerability possess the ability to execute arbitrary code with `SYSTEM` privileges, which enables them to conduct various malicious acts, such as installing unauthorized software, viewing, altering, or deleting data, or even creating new accounts with full user privileges.

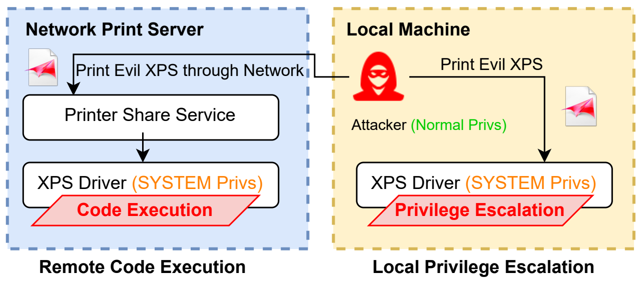

Figure 2: Attack Scenario

We describe a typical attack scenario in Figure 2. In the context of XPS attack scenarios, an attacker with lower-level privileges only needs to submit a meticulously crafted XPS file to the target printer's queue. This document, for instance, could leverage internal resources and objects within the XPS format as primitives for orchestrating heap feng shui [53] to manipulate memory. When local and network print services parse the XPS file, a vulnerability within the `printfilterpipelinesvc.exe` module—which operates with `NT Authority\Local Service` privileges—can be exploited. Consequently, this allows the attacker to potentially achieve Local Privilege Escalation (LPE) and Remote Code Execution (RCE) attacks [26].

3 Design of PrintXPSurge

This section introduces the technical details of PrintXPSurge. Figure 3 provides an overview to the workflow of PrintXPSurge. The objective of this tool is to efficiently and accurately detect vulnerabilities related to the printer XPS components.

3.1 Overview

In this section, we provide an overview of the three phases in which PrintXPSurge detects printer XPS vulnerabilities.

Phase 1: Input Generation. Firstly, we create well-formed XPS files as fuzzing inputs. To do so, we initially construct basic XML files using well-known XPS specifications. Then, we create XPS specific constraints and incorporate these constraints into the generation and mutation of the basic XML files.

Phase 2: Progressive Runtime State Reconstruction. To activate the driver functions, we construct a fuzzing harness tailored to fulfill the execution prerequisites of the driver code. In order to bypass the need for rebuilding complex internal dependencies between the XPS driver and other elements within the printing workflow, our approach uses snapshots to progressively construct the runtime for the entire workflow. Consequently, we utilize dynamic tracing techniques to map out both control and data dependencies at the workflow's interface.

Phase 3: Document Format Repair. At this step, we capture the internal error messages from the driver and use a large language model to repair the incorrect XPS format based on these error messages. In cases where there are errors due to missing resources, we recover the runtime dependencies and introduce the resource files referenced by the generated XPS files into the fuzzing settings.

Phase 4: Bug Localization. Adopting a workflow-level approach for generating the testing framework simplifies the process but introduces ambiguities. When a crash occurs in the XPS driver or other workflow components, it's unclear if it's due to known superficial vulnerabilities or deeper issues. We resolve these ambiguities by eliminating duplicate vulnerabilities, ensuring fuzz testing can explore deeper paths without obstruction. We use tools like TinyInst to collect coverage data, accurately identify root causes, and manually patch issues to prevent recurrence.

3.2 Input Generation

Our goal is to generate well-formed XPS files which can be used to test the parsing component in XPSDrv. Particularly, we aim to identify *deep bugs stemming from the improper processing of* ***well-structured XPS files containing ill-intentioned content***, as opposed to *shallow crashes induced by* ***disorganized XPS files***. As a result, our expected inputs must satisfy several key requirements:

(1) A well-formed XPS file must be a valid XML file. XPS is built atop XML format. A syntactically correct XPS file must follow the XML specification.
(2) An acceptable input must adhere to the XPS specification, encompassing correct XPS tag names and maintaining proper relationships among pertinent tags, as well as accurate associations between tags and their corresponding content.
(3) An expected test input must not be prematurely rejected before it reaches the XPS parser due to runtime errors. It should accommodate all runtime environmental requirements so as to successfully trigger the parsing component (see details in Section 3.4).

Basic XML File Generation. We follow the classic generation-based fuzzing technique [55] to generate well-structured XML files. In particular, we use the XML Schema Definition (XSD) [58] language to define XML templates so that any files generated from these templates must follow XML syntax. In principle, we can define any arbitrary XML templates with random tag names. Yet to simplify the subsequent step of creating valid XPS files, we compile a comprehensive list of tag names utilized in XPS files as outlined in the XPS specification [31], and exclusively use this collection of strings for tag definition.

XPS Constrained Mutation. We then mutate the generated XML files with legitimate XPS tags, applying structural mutations to

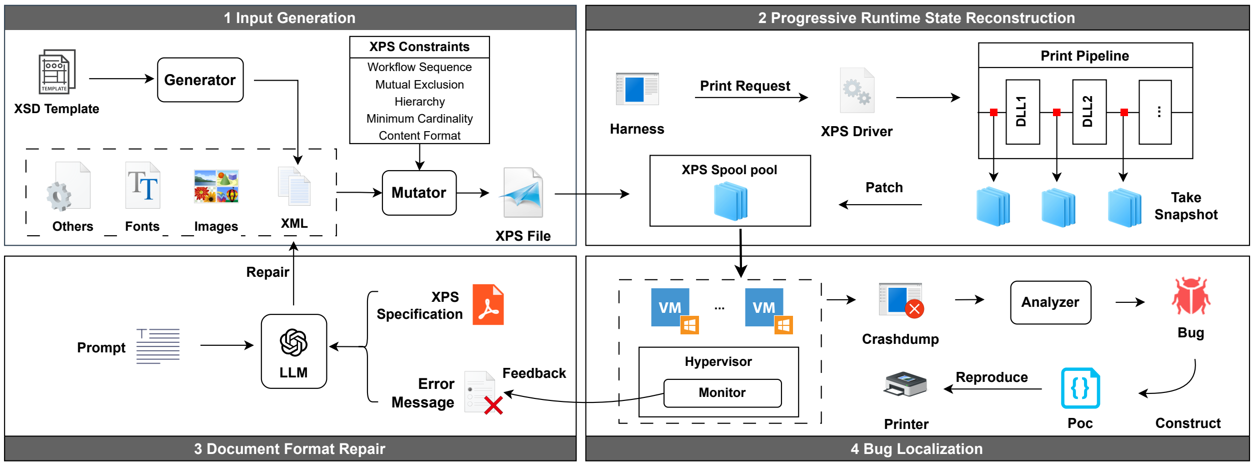

Figure 3: Overview of PrintXPSurge

reconstruct their hierarchy, including shuffling element order and modifying nesting relationships. Our mutation is guided by the unique constraints in XPS files. Specifically, we first randomly alter our seed XML files via adding, removing, and changing elements and properties, and perform value mutations to alter the element contents and attribute values. We then verify whether the modified files conform to XPS-specific constraints and finally discard those that cause any violations.

XPS Constraints. To our study, XPS specific constraints can be put into five categories, as depicted in Table 1, which originate from the semantics of rendering workflow. Notably, the complexity of the XPS specification requires tedious manual efforts to interpret the lengthy textual documentation, summarize all the constraints, and implement a generator that adheres to these rules. To reduce the manual effort involved, we developed an *LLM-assisted "repair-then-patch"* approach (see details in Section 3.4), where we employ a Large Language Model (LLM) to directly repair XPS files to improve their formatting accuracy. In fact, we manually defined only a limited number of key constraints. This is because the goal of our generation phase is not to ensure zero mistakes but rather to decrease the likelihood of producing ill-formed files.

Firstly, the elements in an XPS file are organized naturally in a hierarchy. For instance, a `Path` element in XPS is used to describe vector-based line graphics within a document. Because it defines the geometry of graphical components, it contains the descriptive elements such as `Data` specifying their shapes, `Fill` indicating the color to fill the shapes, and `Stroke` representing the color used for their outline. As a result, the precise parent-children relation of these elements is critical for a XPS parser to correctly interpret the layout of the document.

Secondly, some elements at the same level must be organized in certain order. In the aforementioned example, while most elements under a `Path` can be defined independently by design, there still exist implicit dependencies due to the workflow of the rendering pipeline. For instance, when three properties, `RenderTransform`, `Clip` and `OpacityMask` of a `Path` element are used together, the `RenderTransform` is typically applied first to the element, then the `Clip` is applied to the transformed element, and finally, the `OpacityMask` is applied. This sequence ensures that transformations do not affect the clipping region or opacity mask application.

Thirdly, multiple elements may belong to the same class and only one of the instances can be present at a certain level. For example, a `Brush` instance can be used to fill the shapes in a `Path`. Though a variety of `Brush` types (e.g., `ImageBrush`, `LinearGradientBrush`, etc.) are available, only one can be applied to one `Path`.

Fourthly, specific elements must appear at least once at a level. For instance, `PageContent.LinkTargets` is a property used within a `PageContent` element to define a collection of named destinations within a page. It needs at least one `LinkTarget` to fulfill its purpose of providing navigational anchors within the document.

Last but not least, the content of an individual property may follow a predetermined structures. There are some simple formats such as the encoding used for `Color` or the combination of offsets utilized by `Glyphs.Indices`. Yet complex protocols are also being employed: a `Path.Data` property uses a series of "letter" commands such as `M`, `L`, `Z` to describe the geometry of a path – for example, `M10,20 L30,40 Z` represents a path that first *moves* to point (10,20), *draws a line* to point (30,40), and then *closes*.

Formal Definition. To comprehensively capture these constraints, we formally define such constraints using linear temporal logic (LTL).

Definition 1. Let P be a set of atomic logical proposition symbols about the system $\{p_1, p_2, ...p_{|A|}\}$, e.g., there exists a `Path` element in an XPS file, and let $\Sigma = 2^A$ be a finite alphabet composed of these propositions. Then, the set of **LTL-based XPS constraints** is inductively defined by the grammar:

$$\varphi ::= true \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1\ U\ \varphi_2 \tag{1}$$

where $\neg$ and $\wedge$ denote negation and logical AND operators; $\varphi_1\ U\ \varphi_2$ indicates that φ_1 remains true until φ_2 becomes true; and $\bigcirc\varphi$ means φ is true in the next step. In addition, we also use the following redundant notations: $\varphi_1 \vee \varphi_2$ instead of $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2$ instead of $\neg(\varphi_1 \wedge \neg\varphi_2)$, the *eventually* operator $\Diamond\varphi$ instead of $(true\ U\ \varphi)$, and the *always* operator $\Box\varphi$ instead of $\neg\Diamond\neg\varphi$.

Table 1 illustrates the policies of example constraints. For example, we can specify the *Hierarchy* constraint as $\Box(\exists$`Path` $\in$

Table 1: XPS Constraint Categories

Category	Description	Example	Policy
Hierarchy	Certain element contains predetermined child elements.	`Path` has a child element `Data`.	$\Box(\exists\texttt{Path} \in \mathit{parent}) \rightarrow \Diamond(\exists\texttt{Data} \in \mathit{child})$
Workflow Sequence	Some elements at the same level need to be arranged in a specific sequence.	`RenderTransform` must happen first, followed by `Clip`, and followed by `OpacityMask`	$\Box\exists\texttt{RenderTransform} \rightarrow \Diamond\exists\texttt{Clip}$ $\Box\exists\texttt{Clip} \rightarrow \Diamond\exists\texttt{OpacityMask}$
Mutual Exclusion	Within a given level, only a single instance of an element type is permitted.	Only one `Brush` instance can be used under `Path.Fill`.	$\Box(\exists\texttt{Brush}_A \in \texttt{Fill}_{\texttt{child}}) \rightarrow \Box\neg(\exists\texttt{Brush}_B \in \texttt{Fill}_{\texttt{child}})$
Minimum Cardinality	Specific elements must appear at least once at a level.	At least one `LinkTarget` must exist in `PageContent. LinkTargets`.	$\Box\exists\texttt{PageContent.LinkTargets} \rightarrow \Diamond\exists\texttt{LinkTarget}$
Content Format	The content of an attribute must follow a certain format.	`Path.Data`, `Glyphs.Indices`, `Color` have specific formats	$\Box\exists\texttt{Data} \rightarrow \Box$*The value of*`Data` *must adhere to its format.*

$\mathit{parent}) \rightarrow \Diamond(\exists\texttt{Data} \in \mathit{child})$, which indicates that if a `Path` element exists in the parent level, there must be a `Data` field in the child level. Similarly, we can describe the requirement for *Mutual Exclusion*: $\Box(\exists\texttt{Brush}_A \in \texttt{Fill}_{\texttt{child}}) \rightarrow \Box\neg(\exists\texttt{Brush}_B \in \texttt{Fill}_{\texttt{child}})$, which dictates that if a `Brush` instance is already present as the child of `Fill`, the existence of another instance is *always* false.

3.3 Progressive Runtime State Reconstruction

Motivation. Since our fuzzing target, the XPS printer framework, is not a stand-alone program but a suite of applications within a client/server architecture with internal component communications governed by multiple protocols, and considering that key components of the printing subsystem function as self-starting system services within the operating system, our objective is to reconstruct a replicable runtime environment and workflow for the printer.

At a high level, this is the same problem that prior work WINNIE [25] also expects to solve. Particularly, to circumvent front-end GUI interaction of an individual Windows application so as to directly execute its back-end logic code, WINNIE uses dynamic tracing to extract the control dependencies between the front-end and the back-end, and automatically reconstructs a single harness program that can satisfy these dependencies.

In theory, it is possible to adopt WINNIE's approach to generate a single harness at the entry point of the printing pipeline – a printing application – for the entire workflow. However, in practice, this prior technique, designed to address synchronous dependencies within a single application, cannot easily solve our problem. Our challenge requires recovering *asynchronous relations among multiple separate yet coordinating printing modules*, such as `Winspool.drv`, `spoolsv.exe`, and `printfilterpipelinesvc.exe`.

To address this challenge, we propose a progressive state reconstruction method. The core idea is to gradually build the system states required by individual printing modules as the printing pipeline progresses, rather than creating a single initial state to trigger the entire pipeline. The rationale behind our design is that multiple printing modules, such as spoolers and filters, are loosely coupled by nature, each performing an independent task. Dependencies between modules can often be resolved at their input-output interfaces. Consequently, we leverage our knowledge of the printing workflow to generate stage-aware runtime states at the beginning of each module. These states address inter-module control and data dependencies, while the internal logic of spoolers and filters automatically satisfies intra-module dependencies.

Approach. More concretely, we begin by constructing a simple "seed" harness to initiate the printing workflow. Importantly, the purpose of this harness is not to meet all subsequent internal requirements but rather to serve as a substitute for the application that typically triggers the printing process. To achieve this, we follow WINNIE's approach and perform dynamic tracing on the application that initiates printing requests, enabling us to recover the control and data dependencies at the workflow's interface. Figure 4 demonstrates the part of the initial harness. Specifically, we identify the data dependency between input filename and XPS parser and therefore automatically generate a temporary file with a random name. Similarly, because the printer workflow requires a printer stream and an XPS OM Object Factory, we hence initialize these instances to feed the downstream logic.

```
int harness(wchar_t* filename)
{
  //Create a temporary file name
  char pszName[L_tmpnam] = { '\0' };
  tmpnam_s(pszName);

  //Generate a new printer stream
  if (FAILED(hr = StartXpsPrintJob(
    TEXT("PCL6"),     //Printer name
    NULL,             //jobName
    NULL,             //outputFileName
    NULL,             //progressEvent
    completionEvent, //completionEvent
    NULL,             //printablePagesOn
    0,                //printablePagesOnCount
    &job,             //xpsPrintJob
    &jobStream,       //documentStream
    NULL              //printTicketStream Ticket
  ))) {...}

  //Initialize an XPS OM Object Factory
  if (FAILED(hr = CoCreateInstance(
    __uuidof(XpsOMObjectFactory),
    NULL,
    CLSCTX_INPROC_SERVER, //CLSCTX enumeration
    __uuidof(IXpsOMObjectFactory),
    reinterpret_cast<void**>(&xpsFactory)
    ))) {...}
}
```

Figure 4: Part of the Initial Harness

After our harness initiates the beginning of the printing workflow, we introduce a snapshot-based approach. We run our fuzzing in a virtualized environment and, during the execution of the printing pipeline, generate a virtual machine snapshot based on the

Windows Hypervisor Platform API [38] between each pair of consecutive modules. Each snapshot serves as the input state for the next module. However, these states, especially those involving the data inputs, may not lead to the successful execution of the subsequent stage. Therefore, we must modify each snapshot to produce the correct state that allows the next stage to proceed.

To help understand the problem, we use Figure 5 to illustrate the entire process. A printing request is initialized in the harness, which sends an XPS file to the spooler service. The spooler service schedules the printing job and eventually dispatches this job to the pipeline service. The pipeline service controls a series of filter components, each of which performs a dedicated task. While the pipeline service coordinates all the filters and ensures that they are executed in a specific order, it only defines the interfaces for these filter functions. The actual implementations of these functions are part of the XPS driver, which are loaded by each interface at runtime. *The deep and asynchronous dependency chain makes it extremely difficult, if not impossible, for dynamic tracing to reconstruct the required initial state that can satisfy all subsequent printing stages.*

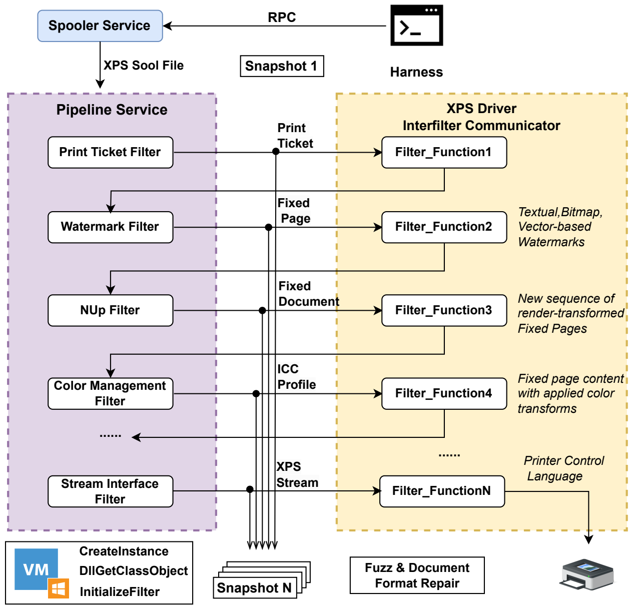

Figure 5: Printing Workflow

This approach is somewhat similar to the forced execution technique [51], which can negate condition check results to ensure the completion of program execution. However, our technique differs in two key aspects. Firstly, while the forced execution technique primarily focuses on patching control flows, our snapshot modifications also aim to generate correct data flows, such as valid file paths to existing resource files, which are critical for testing downstream printing drivers. Secondly, unlike prior work that aggressively circumvents condition checks within internal code logic, we only patch the states at interfaces, preserving the state consistencies within each module.

To achieve this, we must address two technical challenges: *(a)* precisely identifying the timing to collect snapshots, and *(b)* effectively modifying the snapshots to generate the required input states. The first challenge is generally addressed by detecting the loading of dynamically linked libraries, which indicates the initiation of a new filter module. To enhance precision, we incorporate additional domain knowledge, such as analyzing function names. This approach is discussed further in Subsection 3.5. The second challenge is tackled using large language models, as detailed in Subsection 3.4.

3.4 Document Format Repair

Motivation. Once we have collected a snapshot, we need to modify it to ensure that the adjusted state allows the successful execution of the next printing stage. The most naïve approach would be to blindly mutate the snapshot and monitor the execution of the next module until one mutation leads to successful completion.

To expedite this slow process, we resort to our **key observation**: the closed-source Windows driver does not simply crash when presented with ill-formed inputs; instead, it provides detailed error messages explaining ***why*** the input is invalid. For example, when specifying an `ImageBrush` object, an input such as `ImageBrush ImageSource=“.png” Originy=“22.33”` not only includes the required `ImageSource` attribute but also the incorrect `Originy` attribute. In this scenario, the error message – *“Originy=‘22.33’[98]: Unexpected attribute”* – precisely identifies the invalid part of the input. This feedback can be leveraged to guide a fuzzer in effectively adjusting its inputs.

Nevertheless, due to the closed nature of driver programs, the structures of their error messages are not publicly documented. To automatically interpret these feedback messages and “fix” the inputs accordingly, we propose leveraging large language models (LLMs). Recent surveys [22, 24] highlight significant efforts to incorporate LLMs into fuzzer input generation. However, most existing tools rely on publicly available knowledge about ***white-box*** systems that LLMs can learn from, such as well-known protocol specifications [28, 30, 59], open-source system implementations [62, 63], or code patterns associated with performance bottlenecks during fuzzing [60]. In contrast, using LLMs to interpret runtime feedback from ***black-box*** systems remains underexplored.

While recent work [61] leverages log data to guide the fuzzing of network protocols, it primarily focuses on using log information to infer the correct sequence of input events – e.g., ensuring that a session connected event precedes a session closed event – rather than addressing the structural correctness of individual input items. For instance, it does not handle cases where an `ImageBrush` object incorrectly includes an `Originy` attribute, which requires a more fine-grained semantic interpretation of feedback messages.

To the best of our knowledge, this work is the first to leverage LLMs for interpreting the root causes of ill-formatted structural inputs based on error messages generated by closed-source Windows software. Additionally, we advance this approach by automatically correcting erroneous inputs using runtime feedback, thereby enhancing and streamlining the automated fuzzing process.

Error Messages Collection. In our study, we analyzed a large number of XPS drivers and found that they all have internal exception handling mechanisms (e.g., `cmnException` in `MSxpsPCL6.dll`, a dynamic link library used as a filter in the Microsoft XPS driver). These functions provide detailed plaintext error messages that can

```
cmnErrorLog::Add(2LL, 1LL, 1LL,
↪  "xamlAttributes_t::parse",
    "Originy='22.33'[98]: Unexpected attribute;",
    "Originy='22.33'",
    *(_DWORD *)(*((_QWORD *)a2 + 24) + 140LL),
    (const wchar_t *)v12[1]);

cmnException::cmnException(
    (cmnException *)pExceptionObject,
    "xpsFont::loadNotify",
    "Could not load font; uri: /Resources/times.ttf");
    throw (cmnException *)pExceptionObject;
```

```
--- before.xml
+++ after.xml
@@ -1,3 +1,3 @@
-<ImageBrush ImageSource=".png" Originy="22.33"
+<ImageBrush ImageSource=".png"
 Viewbox="0,0,71.567,72.003" TileMode="None"
  ↪  ViewboxUnits="Absolute"
 ViewportUnits="Absolute" Viewport="0,0,1,1">
......
<Glyphs FontUri="/Resources/times.ttf"
↪   Indices="0,0,0,0" FontRenderingEmSize="16"
↪   Fill="#000000" OriginX="10" OriginY="220"
↪   UnicodeString="Sample Text" />
```

Figure 6: Repair Case of Unexpected Attribute and Missing Resoucrce

accurately pinpoint specific fields or elements where errors occur in the XPS file structure. We hook exception functions such as `cmnException` in `MSxpsPCL6.dll` to capture error keywords. When an error keyword appears excessively and a stage can be identified based on API signatures, we submit the error messages along with the associated XPS files to a large language model to facilitate the repair of these XPS files.

Large Language Model-Assisted XPS File Repair. We divided the 200-page XPS Specification into 16 slices for LLM processing. To enhance the LLM's understanding of specific terms in XPS formats that are not standard natural language words, such as the `ImageBrush` object or the `Originy` attribute, we fine-tune the model using XPS specifications and designed a six-step prompt to help the LLM understand what "repair" means. Figure 7 illustrates our prompt message. Specifically, we instruct the LLMs to address: (1) **syntactic errors** in basic data structures like matrices, (2) **spelling mistakes** in tag and attribute names, (3) **structural issues** in attribute hierarchies, (4) **semantic problems** arising from missing, redundant, or invalid attributes, and (5) **logic errors caused** by incorrect dependencies. However, we do not expect the LLMs to fix runtime errors – e.g., a referenced font file `/Resources/times.ttf` does not exist or the path is incorrect – as resolving these issues requires additional knowledge of the execution environment. Instead, we ask GPT-4 to (6) identify and report these problems without attempting to repair them. These issues will be addressed separately by "patching" the fuzzer runtime.

Figure 6 illustrates the repair of an unexpected attribute and missing resource by presenting the error message reported during the parsing of the XPS document, along with a comparison of the document before and after the correction. The first error message indicates a non-standard attribute. GPT-4 provided a relevant analysis for the erroneous XPS document fragment: *"The attribute* `Originy` *was identified by the system as unexpected. Upon verification, it was determined that the* `ImageBrush` *element in the XPS standard does not define this attribute, suggesting it may be a typographical error or unnecessary."* The large language model successfully identified `Originy` as a misspelled or superfluous attribute and automatically removed it. Similarly, if a `Glyphs` attribute is mistakenly closed with a `</Canvas>` tag, the LLM fixes it based on the message: *"Expecting close for* `Glyphs` *element but encountered Canvas."* A missing `</FixedPage>` tag results in the error: *"Missing FixedPage element, or element not closed."* The LLM is well-suited for fixing errors like the ones mentioned above, as error messages help pinpoint the exact location of the issue, allowing it to converge on a solution in just one attempt on average. After the correction, when a second error message, such as *"Could not load font; uri: ......"*, is encountered due to a missing resource, in contrast, the LLM will not attempt to resolve it, and "patching" fuzzer runtime will handle it.

Prompt

This is an XPS file parsing error message: <Insert error message>

Based on the following XPS official documentation: <Insert XPS official documentation content>

Please attempt to repair the following erroneous XPS document by addressing the following:
(1) Fix basic syntax issues, such as improper formats for colors and matrices.
(2) Correct spelling errors in tags and attributes.
(3) Tackle complex structural problems, such as mismatched XML tags and incorrect hierarchies.
(4) Resolve attribute errors, including missing required attributes, duplicate attributes, and invalid values.
(5) Address logical errors, such as inconsistencies between attribute values and missing dependencies.
(6) If there are runtime errors such as missing resource files, report these errors without attempting to fix them.

Output the corrected document format.

Figure 7: Prompt for Finetuning

"Patching" Fuzzer Runtime. A well-formed XPS file may still contain invalid content, such as a link to a file that does not exist. To also satisfy requirement (3) in Section 3.2, we will further patch these runtime errors and provide the fuzzing environment with correct runtime values. The underlying cause of potential runtime errors often lies in the dependency between rendering XPS files and the access to external resources. For instance, an XPS file can employ a special font from a `.ttf` file defined in the `FontUri` or load an PNG image from a `ImageSource`. Based on the information provided by the error message, we update the relationship file (`.rels` file) to include the relationship for the new resource. Specifically, we add a new `Relationship` element, specifying the ID and type of the new resource, and place the resource in the corresponding folder. It is important to note that this patching process is not a one-time task. As the printing workflow progresses, different runtime checks are incrementally performed by individual printing modules, requiring continuous updates.

3.5 Implementation Details

We have implemented PrintXPSurge in Python and C. To achieve this, we incorporated several existing tools including TinyInst, IDA Python, Process Monitor[44], Detours, and WinDbg[46].

Constraint Generation. To construct these policies, we manually interpret XPS specifications and derive the constraints with respect to the XPS elements and their dependencies. Note that, XPS specs may not exhaustively cover all possible formats for attribute values. Thus, to address *Content Format* constraints, we instead collect and reuse the content values from existing well-formed XPS files. Our LTL constraint solving is performed using an off-the-shelf model checker NuSMV [49]. Particularly, we first extract relations among attributes from XML files, and convert these relations to identifiers and transitions in NuSMV models, and then use NuSMV to verify the constructed models against manually defined LTL constraints in order to determine consistency/violations.

Our LTL specification generation requires manual effort, but once created, XPS file generation and mutation are fully automated. We implemented the specs in 416 LOC, completed by one person in 40 hours, and achieving high coverage of specifications is not essential, as we leverage an LLM to repair ill-formed inputs.

Progressive Runtime State Reconstruction. By hooking into the XPS API[35], which is a native Windows API allowing programs to create, read, edit, save, and print XPS files and is supported on Windows 7 and later versions for use by both user-mode programs and XPSDrv printer drivers, we were able to record the trace information of printer calls, capturing the entire sequence of API calls. In the XPS driver configuration file, the names of all filters for the XPS driver are recorded. For different filters, we monitor by hooking each filter's specific interface to confirm when these filters are called. For functions, we try to cover all tags and resources when generating XPS documents to test as many different functions as possible. We then employed traditional static reverse engineering analysis methods to reconstruct function prototypes. This involved combining static analysis provided by IDA Pro[20] with specific information obtained from dynamic execution tracing. Additionally, we used Process Monitor, a Windows API monitoring program, to observe sensitive behaviors such as DLL loading, file reading and editing, and registry key value modifications.

We determine the time window for obtaining a snapshot by monitoring signature function calls. For example, to identify the starting point of XPS filters, we rely on our knowledge that each filter object must be loaded and initialized by a filter pipeline manager immediately before the filter execution. Specifically, the filter pipeline manager loads each filter, implemented as a dynamically linked library (DLL), by invoking the exported function `DllGetClassObject()` defined in the library. It then initializes the filter object by triggering the callback function `InitializeFilter()` that implements the `IPrintPipelineFilter()` interface in any XPS filters. Therefore, by dynamically tracing and observing the invocation of these functions at runtime, we can confirm that the next filter module has successfully started. At this point, we can take a snapshot using tools such as WinDbg [46] to capture the current memory and CPU states. XPS files are fixed by directly modifying the memory snapshot at the locations where these files reside.

Bug Localization. In the fuzzing process, sanitizers like PageHeap[37] and AppVerifier[36] are used to monitor the heap memory state of all running processes and issue warnings when memory corruption is detected at runtime. By streamlining the testing framework generation with a workflow-level approach, we inadvertently introduce some ambiguity. Specifically, when a crash occurs, it is not immediately clear whether the XPS driver or another component in the pipeline is responsible. This ambiguity can lead to the premature exposure of superficial vulnerabilities, hindering the progression of subsequent testing processes. In practice, the repeated occurrence of these shallow defects can disrupt or halt extensive printing workflows, further impeding the exploration of deeper components within the pipeline. At this stage, our objective is to eliminate redundant vulnerabilities and ensure the continuity of normal printing operations. First, to accurately pinpoint the source of the detected vulnerabilities, we perform bug localization by backtracking through the call stack. To achieve this, we employed TinyInst[16] to instrument and collect coverage information for our test targets. TinyInst is a lightweight dynamic instrumentation library. Unlike similar tools such as DynamoRIO[19] and Pin[27], TinyInst is designed to be more lightweight, easier to understand, and more convenient for developers to customize. By leveraging TinyInst, we were able to efficiently gather detailed coverage information, which is crucial for our analysis and testing purposes. In particular, we attached TinyInst to the printer process `printfilterpipelinesvc.exe` and instrumented the corresponding graphic DLLs to record the paths traversed during the parsing of an XPS file. After identifying the root cause of the duplicate vulnerabilities, we manually patch the issue to prevent subsequent crashes from originating from the same source.

PoC Construct. We conducted an in-depth analysis of the XPS files that caused crashes and performed validation based on this analysis, which introduced some manual work. To confirm exploitability, manual work was required—simple cases took approximately 30 minutes, while complex cases could take up to a day. However, identifying the inputs that triggered crashes and reproducing these crashes was fully automated. To assess the severity of these bugs, we constructed both local printer environments and network remote printer environments. We crafted XPS files that caused crashes by arranging objects and resource files to perform heap feng shui. These crafted XPS files were then sent via print requests to both local machines and remote machines in a network environment to evaluate whether these vulnerabilities could be exploited for Remote Code Execution (RCE), Local Privilege Escalation (LPE), or information disclosure.

4 Evaluation

We have identified 10 presentative XPS printer drivers on Windows 10 version 10.0.19041.1806 for our evaluation. These printer drivers include official Microsoft XPS printer driver components [43], specifically Microsoft PCL6 Class, PS Class and OpenXPS Class Drivers, as well as drivers from major printer vendors such as Canon, HP, Kyocera, etc. We have also chosen the state-of-the-art coverage-guided fuzzers, WinAFL [15] and Winnie [25], on the Windows platform, along with the open-source generative XML fuzz tools, xmlfuzzer [29], and untidy [57], for comparative analysis. The experiments were conducted on a machine equipped with 32GB of RAM and an Intel i7-13700K processor, running the operating systems configured with their default settings for assessment purposes. Our evaluation aimed to answer the following research questions:

RQ1. How effective is PrintXPSurge in discovering of vulnerabilities, particularly compared to existing tools?

RQ2. What is the extent of potential harm caused by these vulnerabilities, and what specific damages can they inflict?

Table 2: List of Drivers Tested with PrintXPSurge

Brand	Driver Name	Filter Name	Version	#Unique Crashes	#Bugs
Microsoft	Microsoft PCL6 Class Driver	MSxpsPCL6.dll	10.0.19041.1806	58	21
	Microsoft PS Class Driver	MSxpsPS.dll	10.0.19041.1806	60	17
	Microsoft OpenXPS Class Driver	MSxpsPS.dll	10.0.19041.1806	0	0
Canon	Generic PCL6 V4 Printer Driver	cnnv4_flayout.dll cnnv4_faqua.dll	2.1	37	16
	Generic UFR II V4 Printer Driver	cnnv4_flayout.dll cnnv4_faqua.dll	2.1	16	13
HP	Smart Universal Printing Driver	hpxpspdlconverter.dll	4.01.2.2972	76	20
Kyocera	Kx v4 Printer Driver	kyv4.dll MSxpsPCL6.dll	6.1.1603	27	14
TOSHIBA	TOSHIBA V4 Printer	eSc6m.dll	10.70.3989.43	1	1
Dell	Dell Printer S5830dn XPS v4	DKAESKF*.DLL (13) *	03142016	0	0
Ricoh	PCL6 V4 Driver for Universal Print	r4600fxl.dll	4.26.0.0	1	0

* The driver contains a total of 13 Filter DLLs named with the prefix DKAESKF.

4.1 RQ1: How effective is PrintXPSurge in discovering of vulnerabilities?

We selected V4 architecture XPS printer drivers from common brands available on the market, ensuring that all drivers were updated to their latest versions at the time of our testing. During our tests, we identified zero-day bugs in seven of these drivers, all of which were discovered exclusively by PrintXPSurge and could not be detected by other tools. Table 2 presents our test results.

Evaluation of XPS File Generation Correctness. In our evaluation, we tested the correctness of XPS files generated using tools such as PrintXPSurge, untidy, xmlfuzzer, and WinAFL. To assess the effectiveness of PrintXPSurge in generating XPS files, we employed several evaluation criteria: We used Python's XML syntax parsing module to verify the correctness of the XML format and applied LTL rules to verify the correctness of the XPS format. To verify the correctness of the generated XPS files, we wrote additional LTL rules. Since it is very difficult to cover all possible rules completely, we only wrote a subset of LTL rules that were not used during generation to validate the effectiveness of the LLM in repairing XPS files. Considering the inherent non-determinism associated with the LLM, we account for variability by conducting five experimental iterations. This approach balances computational cost and performance while enabling the computation of an arithmetic mean across multiple measurements. Table 3 presents a comparative analysis of the tools, with columns two and three detailing the outcomes for the two specified evaluation criteria, respectively.

Through manual analysis and debugging, we identified significant shortcomings in WinAFL when applied to fuzzing highly structured files like XML. Firstly, the test cases generated by its mutation strategy often lack syntactic correctness. This deficiency results in the generated test cases being unable to pass the format checks within the module, hindering the execution of subsequent parsing processes. Additionally, WinAFL's limitation to mutating existing corpora prevents the generation of new XML tags. Even if the test cases pass the format check, achieving meaningful coverage growth during the fuzzing process proves challenging. Notably, the generation module used by Winnie is based on WinAFL.

As for the generative XML fuzz tools, xmlfuzzer and untidy, the test cases they produce can successfully pass the format checks within the module. However, these tools face limitations in generating hierarchical logical structures in the XPS format. This limitation, too, leads to interruptions in the printing process. xmlfuzzer primarily focuses on the syntactic structure of XML rather than the semantic content of the documents. While it is capable of generating documents that conform to XML syntax, these documents may be semantically incorrect or invalid.

Table 3: Tools and Their Correctness Evaluation Criteria

Tool	XML	Valid XPS
WinAFL/Winnie	4.48%	0.89%
xmlfuzzer	84.50%	0.00%
untidy	2.02%	0.78%
PrintXPSurge(without LLM Fix)	**100.00%**	**31.70%**
PrintXPSurge	**100.00%**	**75.60%**

Evaluation of XPS Printer Driver Vulnerability Discovery. While maintaining the consistency of XPS generation correctness, we evaluated the effectiveness of PrintXPSurge in exploring vulnerabilities. We selected two drivers from the suite, namely the Microsoft PCL6 Class Driver and the Microsoft PS Class Driver, for comparison experiments with other tools. These drivers are officially provided by Microsoft for the conversion of files into PCL and PS, respectively. The majority of third-party drivers largely rely on calling the filters of these two drivers to execute their final processing tasks.

The fuzz testing cycle is 24 hours. To ensure fairness in the experiments comparing the vulnerability detection capabilities of different tools, we did not provide any valid XPS files as seeds. As shown in Table 4, the prevalence of insecure XPS-related functions in printers is notably high.

During the 24-hour testing period, PrintXPSurge triggered 111 and 117 crashes in the `MSxpsPCL6` and `MSxpsPS` modules respectively, with unique crashes tallying at 58 for the former and 60 for the latter, while WinAFL, Winnie, xmlfuzzer, and untidy did not trigger any crashes. Among these crashes, we ascertained 38 exploitable vulnerabilities through manual analysis and debugging.

Duplicate Bug and False Positive. Due to vulnerabilities in certain shallow regions of the XPS driver, a significant number of crashes may occur at these same locations. As illustrated in Table 4, out of the 111 and 117 crashes identified in MSxpsPCL6 and MSxpsPS, only 58 and 60 are unique crashes, respectively, with the

Table 4: Comparison of Crashes, Bugs Found by Various Tools

Driver	Tool	#Crashes	#Unique Crashes	#Bugs[1]
MSxpsPCL6	WinAFL	0	0	0
	Winnie	0	0	0
	xmlfuzzer	0	0	0
	untidy	0	0	0
	PrintXPSurge	111	58	21
MSxpsPS	WinAFL	0	0	0
	Winnie	0	0	0
	xmlfuzzer	0	0	0
	untidy	0	0	0
	PrintXPSurge	117	60	17

[1] Bugs represent exploitable memory corruption vulnerability.

proportion of duplicate crashes approaching 50%. Among these, there are five false positives, all of which are null pointer dereferences. The classification of these bugs as false positives, rather than CVEs, is justified by their lack of exploitability. Microsoft has declined to acknowledge these issues as security threats and refuses to implement any fixes. Additionally, we investigated why the Microsoft OpenXPS Class Driver, listed in Table 2, experienced zero crashes despite using the same version of the filter as other drivers. We discovered that the Microsoft OpenXPS Class Driver does not perform XPS parsing; instead, it retains the original format in its output.

4.2 RQ2: Exploiting Vulnerabilities

We manually analyzed the exploitability of the bugs found in Table 2 and reported them to the respective vendors for verification. Table 5 provides detailed information on 17 CVEs that we identified in both official Microsoft and third-party modules, which have been confirmed by the vendors. Each row in the table represents a unique CVE and provides relevant information such as the module name, vulnerability function, vulnerability type, impact, and status.

Of the 17 confirmed CVEs, 15 vulnerabilities have been identified as exploitable for remote code execution, while the remaining 2 vulnerabilities can be exploited for information disclosure. This highlights the significant risks associated with XPS printers and underscores the necessity of implementing robust security measures to mitigate these threats. In this section, we will examine two specific vulnerability cases discovered by PrintXPSurge to analyze their underlying causes and discuss appropriate mitigation strategies.

CaseStudy I: CVE-2023-24925. This case demonstrates the presence of three deep vulnerabilities exclusively discovered by PrintXPSurge within the same function. These bugs cannot be revealed if the fuzzing process is not able to successfully pass the input validation and printing pipeline. Specifically, it identified two Use After Free (UAF) vulnerabilities and a Function Pointer Hijacking vulnerability in `MSxpsPS.dll`. These vulnerabilities were reported to Microsoft in 2023 and assigned CVE-2023-24925. The root cause of these vulnerabilities, which occurs during the parsing of an fpage document, is located in the `MSxpsPS!xamlStOptimize::optimize+0x2f1` function.

Figure 8 provides a simplified representation of the code. This code snippet contains three vulnerabilities: two UAF vulnerabilities and an instance of Function Pointer Hijacking.

(1) The first instance of the UAF vulnerability occurs within the if statement. When the value of `*(_DWORD *)(v21 + 8)` equals 29, there is a use of previously deallocated memory. This occurs because the memory block is freed but not set to NULL or subjected to any other appropriate handling.
(2) The second UAF vulnerability occurs within the if statement block, specifically in the `*(struct xamlNode **)(v21 + 104)` statement. Similarly, previously deallocated memory is being accessed.
(3) The third vulnerability involves Function Pointer Hijacking. In the code segment `(*(__int64 (__fastcall **)(__int64))(*(_QWORD *)v21 + 216i64))(v21)`, by controlling the value of `*(_QWORD *)v21`, it is possible to control the function pointer `(*v21+216)`, thereby manipulating the program's execution flow (Instruction Pointer).

```
// (*v21) UAF here
if ( *(_DWORD *)(v21 + 8) == 29 )
{
    // UAF here too
    v22 = *(struct xamlNode **)(v21 + 104);
    if ( v22 )
    {
    // UAF here too
    xamlVisualBrush::getVbToVpTfm( (xamlVisualBrush
    ↪ *)v21, (struct cmnMatrix_t *)v35);
    // Controlling function pointer
    v23 = (*(__int64 (__fastcall
    ↪ **)(__int64))(*(_QWORD *)v21 + 216i64))(v21);
    ...
```

Figure 8: Simplified Code Snippet of CVE-2023-24925

These vulnerabilities could potentially be exploited by malicious attackers to execute arbitrary code, manipulate the behavior of programs, and possibly lead to system crashes or data breaches. Methods to remediate these vulnerabilities include timely memory deallocation and nullifying reference pointers, as well as strengthening input validation mechanisms to prevent attacks such as function pointer hijacking.

CaseStudy II: CVE-2023-24927. Another typical vulnerability reported by PrintXPSurge is CVE-2023-24927, a type confusion vulnerability found in `MSxpsPCL6.dll`. A type confusion vulnerability occurs when a program allocates a piece of memory for one type of object but later accesses it as a different type. This can lead to unpredictable behavior, including crashes and security breaches.

This vulnerability occurs when a user attempts to print a local document using the Microsoft PCL6 Class Driver printer. The issue arises with the `</Path.Stroke>` tag, which always expects a correct `Brush` object. However, if a correct `Brush` object is not provided, it leads to type confusion. In this example, the `Path` element is being drawn using a `Brush` object of type `xamlSolidColorBrush`.

PrintXPSurge successfully generated and fixed the correct syntax format `<Path.Stroke> </Path.Stroke>` and provided a `Brush` object, but it did not construct the correct virtual function table (vtable) for it, as this was beyond its capabilities. In this particular scenario, the provision of an incorrect `Brush` object being provided and an attempt to invoke the virtual function table on the object result in type confusion, ultimately leading to program crash. When the `poc.xps` is printed using the Microsoft PCL6 Class Driver printer with PageHeap enabled, the type confusion occurs at the following

Table 5: Details of Vulnerabilities Discovered by PrintXPSurge in Microsoft Official and Third-Party Modules.

No.	Driver Name	Vulnerable Function	Vulnerability Types	Impact	Status
1	MSxpsPS	nnXXXXXXew	Out-of-bounds Read	Information Disclosure	CVE-2023-32085
2	MSxpsPS	xrXXXXXXze	Out-of-bounds Write	Remote Code Execution	CVE-2023-24928
3	MSxpsPS	xaXXXXXXze	Use After Free	Remote Code Execution	CVE-2023-24925
4	MSxpsPS	PCXXXXXXx3	Out-of-bounds Write&Read	Remote Code Execution	Merged
5	MSxpsPS	xaXXXXXXnt	Out-of-bounds Write&Read	Remote Code Execution	Merged
6	MSxpsPS	DiXXXXXX8	Out-of-bounds Write	Remote Code Execution	Merged
7	MSxpsPS	xaXXXXXXke	Out-of-bounds Write	Remote Code Execution	Merged
8	MSxpsPS	xaXXXXXXsh	Use After Free	Remote Code Execution	Merged
9	MSxpsPS	xrXXXXXXHi	Out-of-bounds Write&Read	Remote Code Execution	Merged
10	MSxpsPS	xrXXXXXXha	Out-of-bounds Write&Read	Remote Code Execution	Merged
11	MSxpsPS	xaXXXXXXXX	Out-of-bounds Write&Read	Denial of Service	Confirmed
12	MSxpsPS	xaXXXXXXXXsh	Out-of-bounds Read	Denial of Service	Confirmed
13	MSxpsPS	xaXXXXXXXXnd	Out-of-bounds Read	Denial of Service	Confirmed
14	MSxpsPS	diXXXXXXXXER	Out-of-bounds Read	Information Disclosure	Confirmed
15	MSxpsPS	xaXXXXXXXXll	Null Pointer Read	Denial of Service	Confirmed
16	MSxpsPS	xaXXXXXXXXll	Out-of-bounds Read	Denial of Service	Confirmed
17	MSxpsPS	xaXXXXXXXXRE	Out-of-bounds Read	Denial of Service	Confirmed
18	MSxpsPCL6	DiXXXXXXXXe8	Out-of-bounds Write	Remote Code Execution	CVE-2023-24887
19	MSxpsPCL6	xaXXXXXXXXent	Out-of-bounds Write&Read	Remote Code Execution	CVE-2023-24885
20	MSxpsPCL6	xrXXXXXXXXHi	Out-of-bounds Write	Remote Code Execution	CVE-2023-24926
21	MSxpsPCL6	xaXXXXXXXX	Type confusion	Remote Code Execution	CVE-2023-24927
22	MSxpsPCL6	xrXXXXXXXXse	Out-of-bounds Write	Remote Code Execution	CVE-2023-24884
23	MSxpsPCL6	xrXXXXXXXXAlpha	Out-of-bounds Write&Read	Remote Code Execution	CVE-2023-24929
24	MSxpsPCL6	xaXXXXXXXX	Out-of-bounds Write	Remote Code Execution	CVE-2023-28243
25	MSxpsPCL6	cxXXXXXXXXta	Out-of-bounds Read	Information Disclosure	CVE-2023-24883
26	MSxpsPCL6	xaXXXXXXXXush	Use After Free	Remote Code Execution	CVE-2023-24886
27	MSxpsPCL6	cmXXXXXXXXrt	Out-of-bounds Write	Remote Code Execution	CVE-2023-24924
28	MSxpsPCL6	DiXXXXXXXXHi	Out-of-bounds Write	Remote Code Execution	Merged
29	MSxpsPCL6	PCXXXXXXXX3	Out-of-bounds Write&Read	Remote Code Execution	Merged
30	MSxpsPCL6	xaXXXXXXXXmize	Use After Free	Remote Code Execution	Merged
31	MSxpsPCL6	nnXXXXXXXXew	Out-of-bounds Read	Information Disclosure	Merged
32	MSxpsPCL6	DiXXXXXXXX4	Out-of-bounds Write	Denial of Service	Confirmed
33	MSxpsPCL6	xaXXXXXXXX	Out-of-bounds Write&Read	Denial of Service	Confirmed
34	MSxpsPCL6	DiXXXXXXXX	Out-of-bounds Read	Denial of Service	Confirmed
35	MSxpsPCL6	diXXXXXXXXHER	Out-of-bounds Read	Information Disclosure	Confirmed
36	MSxpsPCL6	xaXXXXXXXXRE	Out-of-bounds Read	Denial of Service	Confirmed
37	MSxpsPCL6	xaXXXXXXXXush	Out-of-bounds Read	Denial of Service	Confirmed
38	MSxpsPCL6	xaXXXXXXXXtand	Out-of-bounds Read	Denial of Service	Confirmed
39	HP SUPD	-	Out-of-bounds Write	Remote Code Execution	CVE-2024-9419
40	Canon PCL6/UFR II	-	Out-of-bounds Write	Remote Code Execution	CVE-2025-0234
41	Canon PCL6/UFR II	-	Out-of-bounds Write	Remote Code Execution	CVE-2025-0235
42	Canon PCL6/UFR II	-	Out-of-bounds Write	Remote Code Execution	CVE-2025-0236

Merged: This bug has been merged by Microsoft with other bugs because its cause is similar to other bugs.
Confirmed: This bug has been confirmed through our manual analysis.

address in Figure 9. Therefore, in order to prevent errors, it is essential to ensure that a correct `Brush` object is provided when using the `<Path.Stroke>` tag to fill the boundaries of the path.

```
;MSxpsPCL6!xamlPath::draw+0x769:
;00007ffe`75fb1db9 488b01
mov rax,qword ptr [rcx]
;={MSxpsPCL6!xamlPath::`vftable` (00007ffe`76225570)}
;0:010> t
;MSxpsPCL6!xamlPath::draw+0x76c:
;00007ffe`75fb1dbc 488b80c8010000
mov rax,qword ptr [rax+1C8h]
;={MSxpsPCL6!xamlNode::Track::`RTTI Complete Object Locator' (00007ffe`76260cb8)}
```

Figure 9: Incorrect Vtable Parsing Leading to CVE-2023-24927

4.3 Ethics and Responsible Disclosure

Upon identifying vulnerabilities, we promptly reported them to the Microsoft Security Response Center (MSRC) with proof-of-concept (PoC) exploits, diligently following up until the issues were resolved and patches were released. Upon identification of these vulnerabilities, we promptly communicated our findings to Microsoft. Subsequently, Microsoft implemented the necessary resolutions to address these issues by mid-2023. After these initial patches were released, we reported a second batch of vulnerabilities, which Microsoft fixed in July 2023. Additionally, third-party driver vulnerabilities, such as those from HP, were reported in June 2024 and fixed by the end of October. Microsoft is now focusing on addressing the printer attack surface and is gradually rolling out new mitigation mechanisms, as detailed in Subsection 5.2.

5 Discussion

5.1 Generality of PrintXPSurge

Using PrintXPSurge in Other XPS Parsers. The XPS file format is commonly used in the Windows environment and is typically created using the "Microsoft XPS Document Writer." On other platforms, XPS parsing components can also be developed and integrated to utilize XPS files or perform related parsing operations. To research the security of these parsing libraries across different platforms, it is crucial to implement their core functionalities, such as input and output mechanisms, within a robust testing framework. For example, XPS Viewer, a classic software provided by Microsoft on the Windows platform, supports diverse parsing methods for XPS files. It is composed of both an application and a library file, with clearly defined input and output functions. The .NET platform offers libraries for parsing XPS files, with WPF serving as a subset of .NET and functioning as the UI framework for applications. Within this framework, numerous applications possess the capability to preview and display XPS files. Therefore, applications such as Microsoft Exchange and Visual Studio may also be susceptible to similar attacks when previewing XPS files. Unlike the printer components on the Windows platform, open-source software on platforms like Linux provides abundant documentation and accessible source code, facilitating the identification and modification of variables. For instance, libgxps [12] is an open-source library based on GObject, specifically designed for handling and rendering XPS files. Leveraging this information, our solution can be seamlessly applied to other platforms.

Reusing the LLM repair tech in other document types. This technology exhibits significant versatility, making it feasible to apply large language model (LLM) repair techniques to fuzz testing of other document parsers. However, as encountered with XPS files, this process does present certain challenges. We address these blackbox issues by leveraging feedback mechanisms and supplementing with ancillary documentation to enhance the model's comprehension of the process. Our approach combines general methodologies with specialized treatments, extending beyond mere LLM repair.

The core advantage of LLM repair technology lies in its flexibility and adaptability. In the Windows printing architecture, XPS format printing is predominant, and most Windows print jobs involve some form of XPS conversion[39]. When considering other input methods, such as using LLMs to repair PDF files that often contain binary data, it is necessary to utilize stream converters within PDFs to enable traditional file transfer programs to recognize the binary nature of the files. Although the Windows development guidelines recommend using debugging information[33], the practical feasibility of this must be evaluated on a case-by-case basis. If the error logs provide detailed and sufficient information, they can be very useful. When error logs are effective, due to their data-driven nature, LLMs can be fine-tuned and trained to accommodate various document formats.

Furthermore, this technology can be extended beyond printing architectures to other frameworks, such as Office applications (e.g., `EXCEL.EXE`). When applied to Office-related formats, OpenXML (OOXML), introduced by Microsoft in Office 2007 as a new document format, offers extensive documentation [42] to elucidate its structure. Similar to XPS, manually identifying and establishing constraints for such formats is exceedingly challenging. Our preliminary reverse engineering of Microsoft Office binary files indicates that there are dedicated logging components capable of providing rich logging and diagnostic capabilities.

5.2 Windows Protected Print Mode

Our research contributes indirectly to the development of a new security mechanism for Windows printers, termed Windows Protected Print Mode (WPP). Historically, the Windows printing system has been a favored target for adversaries. The spooler service operates with elevated privileges and is required to load code from the network, which exposes it to significant risks. A substantial number of vulnerabilities related to XPS rendering were reported to Microsoft, highlighting inherent design flaws within the XPS printing framework.

In WPP, XPS rendering processes will execute with user-level permissions instead of system-level, effectively mitigating a large swath of vulnerabilities associated with XPS rendering. Furthermore, under WPP, printers are restricted to loading only Microsoft-signed binary files, while third-party code is contained within an AppContainer, providing an additional layer of defense. Microsoft is also planning to phase out support for third-party driver services within Windows [40]. Starting in 2025, the company will no longer accept new drivers from printer manufacturers, and beginning in 2027, Microsoft will cease the distribution of updates for third-party drivers. However, it is anticipated that the most significant impact will be on the Windows Update channel, as users will still be able to manually install packages from vendor websites after 2027, suggesting that these drivers may continue to pose challenges for a prolonged period.

An analysis [39] conducted by Microsoft on historical MSRC cases pertaining to Windows Print has been carried out to assess the effectiveness of these changes. The Windows Print Protection mode has mitigated over half of the vulnerabilities. However, as this new mechanism is still in the testing phase, it has not entirely supplanted the older printer model.

6 Related Work

Windows Platform Fuzz: There have been several research studies on fuzz testing for the Windows platform. Winnie [25] is an end-to-end solution for fuzzing Windows applications, synthesizing lightweight harnesses to bypass GUI code and directly invoke functions. WinFuzz [54] is notable for its speed and effectiveness, enabling rapid in-memory looping by tracking and resetting program state changes. Digtool [23] is a kernel vulnerability detection framework operating on binary code and built on a virtualization monitor. Choi et al. [6] developed a tool for automated API fuzz testing, evaluated on Windows systems. Forrester et al. [11] used random input testing on Windows NT applications with simple black-box tests.

Printer Security Research: In the realm of printer research, there exist several investigations. Cui et al. [8] provide a comprehensive case study on the vulnerability of HP-RFU (Remote Firmware Update) LaserJet printer firmware. This particular vulnerability allows for the arbitrary injection of malicious software into the printer's firmware through standard printed documents.

7 Conclusion

In this paper, we present PrintXPSurge, the first tool specifically designed to detect vulnerabilities in Windows printer XPS components. PrintXPSurge uses dynamic binary analysis to examine printer binaries and identify insecure interfaces. It also employs large language models to repair XPS files for fuzz testing and uses snapshot techniques to construct the printer's runtime state. Our experiments reveal significant security risks in the Windows printer XPS component, uncovering a total of 13 CVEs.

Acknowledgment

This work was partly supported by the National NSF of China (grants No.62302181) , the Key R&D Program of Hubei Province (2023BAB079, 2023BAB017), Huawei Research Fund, and HUSTCSE-FiberHome Joint Research Center for Network Security. Mu Zhang was supported in part by NSF OAC-2115167 and Battelle Energy Alliance, LLC Contract No. 313367. Additionally, this research was supported by the Hong Kong Polytechnic University Grant (No. 1-ZVG0) and the Hong Kong RGC Grant for the Theme-based Research Scheme Project (No. T43-513/23-N).

References

[1] 2021. CVE-2021-34527. https://cve.mitre.org/cgi-bin/cvename.cginame=CVE-2021-34527.

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.

[3] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[4] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19).*

[5] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *2021 IEEE Symposium on Security and Privacy (SP).*

[6] YoungHan Choi, HyoungChun Kim, and DoHoon Lee. 2008. An Empirical Study for Security of Windows DLL Files Using Automated API Fuzz Testing. In *2008 10th International Conference on Advanced Communication Technology* (2008-02), Vol. 2. 1473–1475. doi:10.1109/ICACT.2008.4494042

[7] CISA. 2021. PrintNightmare, Critical Windows Print Spooler Vulnerability. https://www.cisa.gov/news-events/alerts/2021/06/30/printnightmare-critical-windows-print-spooler-vulnerability.

[8] Ang Cui, Michael Costello, and Salvatore Stolfo. 2013. When firmware modifications attack: A case study of embedded exploitation. In *Network and Distributed System Security Symposium (NDSS 2013).*

[9] Saif El-Sherei. 2017. Demystifying Kernel Exploitation by Abusing GDI Objects. In *DEF CON 25.*

[10] Nicolas Falliere, Liam O Murchu, and Eric Chien. 2011. W32.Stuxnet Dossier. https://www.symantec.com/content/en/us/enterprise/media/security_response-/whitepapers/w32_stuxnet_dossier.pdf.

[11] Justin E. Forrester and Barton P. Miller. 2000. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4* (Seattle, Washington) *(WSS'00).* USENIX Association, USA, 6.

[12] GNOME. 2010. libgxps. https://github.com/GNOME/libgxps

[13] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[14] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).*

[15] Google Project Zero. 2016. WinAFL. https://github.com/googleprojectzero/winafl.

[16] Google Project Zero. 2020. TinyInst. https://github.com/googleprojectzero/TinyInst.

[17] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller. 2018. Sample-Free Learning of Input Grammars for Comprehensive Software Fuzzing. arXiv:1810.08289 [cs.SE]

[18] Yeming Gu, Hui Shu, Rongkuan Ma, Lin Yan, and Lei Zhu. 2022. spotFuzzer: Static Instrument and Fuzzing Windows COTs. arXiv:2201.07938 [cs.SE]

[19] Marc Heuse. 2018. AFL-DynamoRIO. https://github.com/vanhauser-thc/afl-dynamorio.

[20] HexRays. 2024. IDA Pro. https://www.hex-rays.com.

[21] Honggfuzz 2010. Honggfuzz. https://github.com/google/honggfuzz.

[22] Linghan Huang, Peizhou Zhao, Huaming Chen, and Lei Ma. 2024. Large Language Models Based Fuzzing Techniques: A Survey. arXiv:2402.00350 [cs.SE] https://arxiv.org/abs/2402.00350

[23] Pan Jianfeng, Yan Guanglu, and Fan Xiaocao. 2017. Digtool: a virtualization-based framework for detecting kernel vulnerabilities. In *Proceedings of the 26th USENIX Conference on Security Symposium.* 149–165.

[24] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, and Quan Zhang. 2024. When Fuzzing Meets LLMs: Challenges and Opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE 2024).*

[25] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021).*

[26] Zhiniang Peng Lewis Lee, XueFeng Li. 2021. Diving Into Spooler: Discovering LPE and RCE Vulnerabilities in Windows Printer. In *Black Hat USA 2021.*

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[28] Xiaoyue Ma, Lannan Luo, and Qiang Zeng. 2024. From One Thousand Pages of Specification to Unveiling Hidden Bugs: Large Language Model Assisted Fuzzing of Matter IoT Devices. In *33rd USENIX Security Symposium (USENIX Security 24).*

[29] Alexander Markov. 2024. xmlfuzzer. https://komar.in/en/code/xmlfuzzer.

[30] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS 2024).*

[31] Microsoft. 2007. XPS Specification and Reference Guide. https://www.ecma-international.org/wp-content/uploads/XPS-Standard.pdf.

[32] Microsoft. 2010. Microsoft Security Bulletin MS10-061 - Critical. https://learn.microsoft.com/en-us/security-updates/securitybulletins/2010/ms10-061.

[33] Microsoft. 2021. Debugging Printer Driver Components. https://learn.microsoft.com/en-us/windows-hardware/drivers/print/debugging-printer-driver-components.

[34] Microsoft. 2021. Print Spooler. https://learn.microsoft.com/en-us/windows/win32/printdocs/print-spooler.

[35] Microsoft. 2021. XPS Document API. https://learn.microsoft.com/en-us/windows/win32/printdocs/documents-xps.

[36] Microsoft. 2022. Application Verifier - Overview. https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/application-verifier.

[37] Microsoft. 2022. GFlags and PageHeap - Windows drivers. https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap.

[38] Microsoft. 2022. Windows Hypervisor Platform API Definitions. https://learn.microsoft.com/en-us/virtualization/api/hypervisor-platform/hypervisor-platform.

[39] Microsoft. 2023. A new, modern, and secure print experience from Windows. https://techcommunity.microsoft.com/t5/security-compliance-and-identity/a-new-modern-and-secure-print-experience-from-windows/ba-p/4002645.

[40] Microsoft. 2023. End of servicing plan for third-party printer drivers on Windows. https://learn.microsoft.com/en-us/windows-hardware/drivers/print/end-of-servicing-plan-for-third-party-printer-drivers-on-windows.

[41] Microsoft. 2023. Introduction to Point and Print. https://learn.microsoft.com/en-us/windows-hardware/drivers/print/introduction-to-point-and-print.

[42] Microsoft. 2023. Office Open XML file formats. https://ecma-international.org/publications-and-standards/standards/ecma-376/.

[43] Microsoft. 2023. Standard XPS Filters. https://learn.microsoft.com/en-us/windows-hardware/drivers/print/standard-xps-filters.

[44] Microsoft. 2024. Process Monitor. https://learn.microsoft.com/en-us/sysinternals/downloads/procmon.

[45] Microsoft. 2024. V4 printer driver. https://learn.microsoft.com/en-us/windows-hardware/drivers/print/v4-printer-driver.

[46] Microsoft. 2024. WinDbg. http://www.windbg.org.

[47] Microsoft. 2024. Windows Presentation Foundation documentation. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/.

[48] Microsoft. 2024. XPS Printer Driver (XPSDrv). https://learn.microsoft.com/en-us/windows-hardware/drivers/print/xpsdrv-printer-driver.

[49] NuSMV: a new symbolic model checker 2024. NuSMV: a new symbolic model checker. https://nusmv.fbk.eu/.

[50] Peach Fuzzer 2024. Peach Fuzzer. https://peachtech.gitlab.io/peach-fuzzer-community/.

[51] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *23rd USENIX Security Symposium (USENIX Security 14).*

[52] Morten Schenk. 2017. Taking Windows 10 Kernel Exploitation to the Next Level – Leveraging Write-What-Where Vulnerabilities in Creators Update. In *Black Hat USA 2017.*

[53] Alexander Sotirov. 2007. Heap Feng Shui in JavaScript. In *Black Hat Europe 2007.*

[54] Leo Stone, Rishi Ranjan, Stefan Nagy, and Matthew Hicks. 2023. No linux, no problem: fast and correct windows binary fuzzing via target-embedded snapshotting. In *32nd USENIX Security Symposium (USENIX Security 23).* 4913–4929.

[55] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley Professional.

[56] Peleg Hadar Tomer Bar. 2020. A Decade After Stuxnet's Printer Vulnerability: Printing is Still the Stairway to Heaven. In *Black Hat USA 2020.*

[57] untidy - XML Fuzzer 2024. untidy - XML Fuzzer. https://github.com/kbandla/python-untidy.

[58] World Wide Web Consortium (W3C). 2012. W3C XML Schema Definition Language (XSD). https://www.w3.org/TR/xmlschema11-1/.

[59] Jincheng Wang, Le Yu, and Xiapu Luo. 2024. LLMIF: Augmented Large Language Model for Fuzzing IoT Devices . In *2024 IEEE Symposium on Security and Privacy (SP).*

[60] Sally Junsong Wang, Jianan Yao, Kexin Pei, Hidedaki Takahashi, and Junfeng Yang. 2024. Detecting Buggy Contracts via Smart Testing. arXiv:2409.04597 [cs.SE] https://arxiv.org/abs/2409.04597

[61] Feifan Wu, Zhengxiong Luo, Yanyang Zhao, Qingpeng Du, Junze Yu, Ruikang Peng, Heyuan Shi, and Yu Jiang. 2024. Logos: Log Guided Fuzzing for Protocol Implementations. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024).*

[62] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024).

[63] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2024. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. arXiv:2401.00563 [cs.CR] https://arxiv.org/abs/2401.00563

Appendix A Windows Printing Architecture

In Windows, printing involves various components and services. To improve compatibility and development of printing applications and drivers, Microsoft has established a standardized printer model since Windows 1.0, continuously updating it to the latest version, V4[45].

The Windows printing architecture includes a print spooler program[34] and a series of print drivers. The spooler is essential for managing and coordinating printer drivers, primarily by retrieving and loading the appropriate ones. Print drivers consist of multiple printer graphics DLLs, which are part of the User Mode Print Driver (UMPD). These DLLs assist in the graphic rendering of print jobs and transmit the rendered data stream to the backend print program.

The main difference between the XPS print spooler and the GDI print spooler is the replacement of the GDI graphics interface with the XPS graphics interface. In Windows, developers can build graphic applications using either the GDI graphics interface or the WPF (Windows Presentation Foundation[47]) development framework. WPF, introduced in Windows Vista, is a newer user interface development framework that offers more visually appealing user interfaces. WPF applications exclusively support the XPS print interface. In other words, most new applications developed since Windows 7 are built on WPF and utilize the XPS print interface. Under the XPS framework, applications do not directly invoke drivers; instead, print data is generated through the operating system's XPS service to produce a universal XPS file intended for printing. Figure 10 provides an overview of the printing architecture.

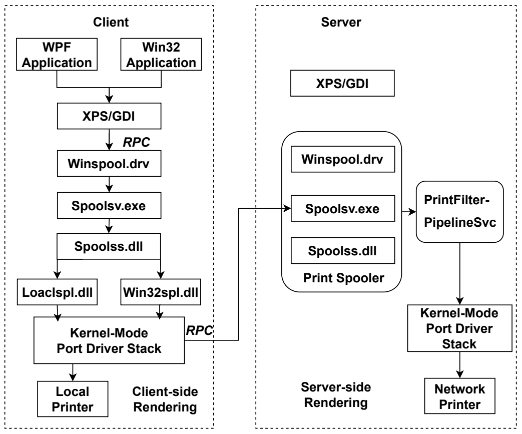

Figure 10: The overview of Windows Printing Architecture

The lifecycle of a print job is as follows: 1) Win32 or WPF applications initiate print tasks with the XPSDrv driver. Win32 apps use the GDI Print API to create XPS spool files via a Conversion Render Module, while WPF apps use the WPF print API for direct creation. These XPS files are placed in the XPS Spool. 2) An RPC call is made by XPS to `Winspool.drv` (the client-side printer pool providing RPC stubs). For instance, the `StartDocPrinter` function forwards calls to the print server (`Spoolsv.exe`). 3) The print server forwards the job to the Print Router (`spoolss.dll`), which assigns it to the appropriate print provider. Local Print Provider (`localspl.dll`) handles directly connected printers, while networked printers are managed by providers like `Win32spl.dll` or `Inetpp.dll`. 4) The User Mode Printer Driver loads, and the XPS file enters the Filter Pipeline. `PrintFilterPipelineSvc.exe` invokes driver filters to render the XPS. 5) The final output file (XPS or Raw) is sent to the printer via a Port Monitor, which facilitates communication between the User Mode Print Pool and Kernel Mode port drivers.

Appendix B XPS Format

The XML Paper Specification (XPS) is an electronic document format created using XML (Extensible Markup Language) and the Open Packaging Conventions (OPC). Commonly used in WPF development for browsing and printing, XPS enhances the efficiency of creating, sharing, printing, viewing, and archiving electronic documents within Windows. Its structure relies on XML, utilizing elements, attributes, and namespaces to define the document's content and layout.

Similar to PDF, XPS also supports resource files that may appear in the document, such as images, fonts, and colors. These resources are defined using XML statements. XPS documents have both a specific physical organization and a logical structure. According to the specification, all files must be placed in specific directory hierarchies. The logical structure is defined using XML to specify the attributes and hierarchical relationships of elements. A typical logical structure of an XPS document includes:

- The `FixedDocumentSequence.fdseq` file, which defines the order of all fixed documents contained in the document. It serves as the entry point of the entire XPS document and contains references to specific documents. It includes multiple `FixedDocumentReference` elements, with each element referencing a fixed document.
- The `FixedDocument.fdoc` file, which defines the structure and content of a fixed document. It includes multiple Page elements, with each element representing a page.
- The `FixedPage.fpage` file, which contains rich content related to the page rendering. `FixedPage` includes all the visual element content for a page, and each page has a fixed size and orientation. The layout of visual elements on the page is determined by fixed page markup, which applies precise typography and layout for graphics and text. The content of the page is described using a set of powerful yet simple visual primitives. Each section of the FixedPage uses `Path` and `Glyphs` elements (with various brush elements) and `Canvas` grouping elements to specify the page content within the `FixedPage` element. `ImageBrush` and `Glyphs` elements (or their child or descendant elements) can reference image or font parts using URIs. They should use relative URIs to reference these parts. Figure 11 shows a simplified example of a `fpage` file following XML rule.

```
<FixedPage Width="331" Height="633">
  <Canvas>
    <Path Data="M100,100 L300,100" Stroke="#FF0000"
    ↪  StrokeThickness="2"/>
    <Glyphs UnicodeString="Hello World"
    ↪  FontUri="/Resources/Fonts/Arial.ttf"
    ↪  FontRenderingEmSize="12" Fill="#000000"
    ↪  OriginX="100" OriginY="120"/>
    <ImageBrush
    ↪  ImageSource="/Resources/Images/image.jpg"
    ↪  Stretch="Uniform" ViewportUnits="Absolute"/>
  </Canvas>
</FixedPage>
```

Figure 11: Example fpage file with Glyphs and ImageBrush.