Exploiting the Trust Between Boundaries: Discovering Memory Corruptions in Printers via Driver-Assisted Testing

Xiaoyu He, Erick Bauman[†], Feng Li[*], Lei Yu, Linyu Li, Bingchang Liu, Aihua Piao, Kevin W. Hamlen[†], Wei Huo, and Wei Zou
Institute of Information Engineering, Chinese Academy of Sciences, China
[†]University of Texas at Dallas, USA
{hexiaoyu, lifeng, yulei, lilinyu, liubingchang, piaoaihua}@iie.ac.cn, {erick.bauman, hamlen}@utdallas.edu

Abstract

TrustScope is a new, a practical approach to identifying vulnerabilities in printer firmware without actually touching the firmware. By exploiting the trust between the firmware and the device drivers, TrustScope analyzes driver software to identify the driver endpoints that output the page description language (PDL) code to be sent to the printer, extracts key constraints for this output, generates new inputs violating these constraints, and fuzzes the printer firmware with malicious PDL code composed with these inputs yet conforming to the grammar of the PDL accepted by the printer. To accommodate the black-box nature of printers, printer behavior is observed strictly externally, allowing TrustScope to detect more vulnerabilities than only those that produce crashes. A variety of key optimizations, such as fuzzing without consuming paper and ink, and offline test case generation, make printer vulnerability detection feasible and practical.

An implementation of TrustScope tested with 8 different printers reveals at least one test case causing anomalous behavior in every printer tested. For most printers it finds multiple vulnerabilities, 6 of which have been assigned CVE numbers, including buffer overflow and information disclosure.

CCS Concepts: • **Security and privacy** → *Embedded systems security.*

Keywords: printer, security, vulnerability

ACM Reference Format:
Xiaoyu He, Erick Bauman, Feng Li, Lei Yu, Linyu Li, Bingchang Liu, Aihua Piao, Kevin W. Hamlen, Wei Huo, and Wei Zou. 2020. Exploiting the Trust Between Boundaries: Discovering Memory Corruptions in Printers via Driver-Assisted Testing. In *Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '20), June 16, 2020, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3372799.3394363

1 Introduction

In recent years, the number of IoT devices has increased rapidly. However, many IoT devices were not designed with security in mind, and IoT security disasters appear regularly. Printers are long-standing members of the IoT device family and have played important roles in our daily life. Unfortunately, as with many other IoT devices, the security of printers tends to be overlooked, and we have witnessed a number of printer attacks in the past several years. In 2013, a vulnerability in HP printers was discovered that allowed an unauthenticated, remote attacker to obtain sensitive information [4]. In 2017, "Stackoverflowin" boasted to have hijacked hundreds of thousands of printers across the Internet and commanded them to emit pages of ASCII art [29]. In 2018, there was a printer hack globally, urging people to subscribe to PewDiePie [11].

Digital printers typically utilize (1) a *page description language* (PDL) to describe the layout of a document, which describes the appearance of a printed page at a higher level than an actual bitmap output [32], and (2) a *PDL interpreter* that comprises the core of the parsing and rendering components in printer devices. Many manufacturers have implemented their own PDLs, such as Adobe's PostScript, HP's PCL, Brother's Brother Type 3 Metalanguage, Canon's CaPSL, Xerox's XES, and so on [30]. PDLs are very complicated and can be used in complex graphics drawing, image rendering,

[*] Feng Li is the Corresponding author. The authors from the Institute of Information Engineering, Chinese Academy of Sciences, are also affiliated with Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing Key Laboratory of Network Security and Protection Technology, and School of Cyber Security, University of Chinese Academy of Sciences.

LCTES '20, June 16, 2020, London, United Kingdom
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7094-3/20/06...$15.00
https://doi.org/10.1145/3372799.3394363

and font processing. For example, PostScript is a stack-based, Turing-complete language that can be used to implement mathematical operations, XML parsing, and even web services, in addition to the functions mentioned above [7]. Due to the complexity of PDLs, vulnerabilities often slip into their interpreters [6, 7, 22].

The most direct approach to identifying vulnerabilities in a printer's PDL interpreter is to analyze the printer firmware in the device. We therefore attempted to acquire firmware for the top five vendors with the largest market share in 2018—*viz.*, Canon, HP, Brother, Epson, and Kyocera—which together comprise 73.4% of the market [28]. Table 1 lists the results of our search, concluding that only firmware from the top two vendors is available, and only in encrypted form. Direct firmware analysis [10, 12] is hence not a viable option.

Fuzzing is an alternative approach, since it affords quick testing of many inputs. However, black-box fuzzing is ineffective when the grammar of the input language is complex and opaque to the fuzzer, as in the case of PDLs, which are vendor-customized. This makes generation-based fuzzers like Peach [20] or Sulley [19] infeasible. While PRET [16] has exploited using grammar-based fuzzing for printer vulnerability discovery, it is mainly targeted at web vulnerabilities, such as print job manipulation and information leakage, and requires experts to develop complex grammatical rules for each PDL, resulting in limited scalability.

Nevertheless, attempts to fuzz printers yield an interesting insight: *The PDL code sent to the printers is generated by printer drivers, which usually contain many security checks to guard the PDL code that they pass to the printer.* However, attackers generating PDL code on a machine under their control can easily bypass these checks by modifying or replacing the driver to send any PDL code they wish. Therefore, a secure printer must re-check the same constraints in their PDL interpreters (or parsers) within the printer firmware to avoid being compromised. If printers do not comprehensively replicate the constraint checks in the firmware, then this presents an opportunity for attacks. The constraints contained within printer driver software therefore reveal a basis for effective fuzzing. Reverse-engineering and violating these constraints to generate malicious PDL code allows us to fuzz printers without acquiring their firmware.

Based on the above insight, we present TrustScope, a fully-automated fuzzing tool capable of identifying vulnerabilities in printers. At a high level, it exploits the trust between the printer drivers and printer firmwares (in printers). The key idea is that printer firmware might not perform the security checks (e.g., integer overflow) again, since printer drivers have already done so while generating the PDL code, and printer firmwares often trust the printer drivers. Therefore, by leveraging the printer drivers to generate syntactically-consistent but semantically-incorrect (and potentially malicious) PDL code using symbolic execution, TrustScope can automatically identify vulnerabilities inside printer firmware by feeding malicious PDL code to printers without accessing the firmware code.

Table 1. The firmware availability of top 5 printer vendors

Brand	Market share	Available?	Unencrypted?
Canon	24.1%	✓	✗
HP	21.4%	✓	✗
Brother	11.3%	✗	-
Epson	9.8%	✗	-
Kyocera	6.8%	✗	-

In short, we make the following contributions.

- **Novel Framework.** We present TrustScope, the first framework to identify vulnerabilities in printer firmware based on symbolic execution of the printer driver's code, by exploiting the trust between printer driver and firmware.
- **Efficient Techniques.** We develop a set of enabling techniques, including PDL output function identification, interprocedural path identification and constraint collection for symbolic execution, efficient mutations for malicious PDL code generation, and printer state inference.
- **Implementation and Evaluation.** We have developed a prototype of TrustScope atop the Windows platform, and tested it with 8 printers from 5 vendors. We have found at least one test case causing anomalous behavior in every printer we tested, and we have obtained 6 new CVEs.

2 Background

2.1 Working Flow of Printers and Assumptions

Figure 1 illustrates a typical flow of data from an application to a printer. As justified by Table 1, we assume printers are black boxes whose firmware code is not directly accessible. However, when a user prints a document, the data typically passes through several layers before finally reaching the printer. In particular, when a user chooses to print a document in some applications, it first passes the document to the printer driver. The driver's rendering modules generate PDL code, which is then sent to the printer. Next, the PDL code is parsed by the corresponding interpreter within the printer firmware, and the result is finally printed onto paper.

According to Figure 1, we can view printer drivers as PDL code generators, responsible for processing the application data into a PDL form interpretable by the printer. We suspect many printers today blindly trust the drivers installed in users' computers. For example, a printer may assume the input PDL code generated by its device driver is well-formed, and therefore completely trusts the PDL code it receives. In other words, we suspect that the input sanitization in a driver's PDL generator is not equal to the input sanitization in its corresponding interpreter in the printer firmware. Based

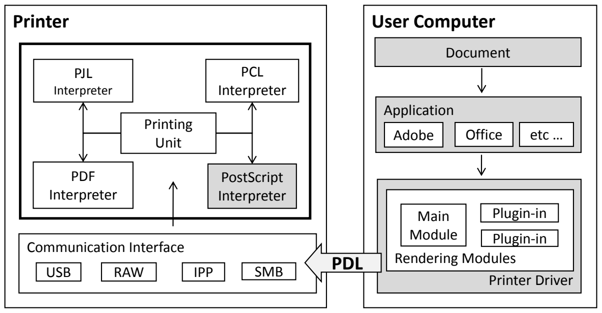

Figure 1. Data flow when submitting a document to print

on these conjectures, our approach is to try to find these inconsistencies and trigger bugs in the printers.

2.2 Key Observations

Our objective is to convert printer drivers into semi-valid PDL code generators. In contrast to black-box PDL fuzzers, this can ensure that the generated PDL code is as valid as possible to pass most of the initial sanitization checks in the PDL interpreter, which are mostly syntactic. On the other hand, we leverage some critical elements in drivers and mutate them in order to trigger inconsistencies (invalid semantics). In particular, there are two kinds of critical constraints that can help generate inconsistent PDL code: predicate constraints and data dependence constraints. Both types of constraints must be satisfied by legitimate PDL code.

Predicate constraints in PDL code. While generating PDL operations "op a,b", the driver checks the legitimacy of operands a and b to ensure that both are within certain ranges, and then filters out illegal ones. Some printers might assume that the driver has already filtered out illegal inputs and therefore trust the incoming data. Omitting its own checks in the firmware makes the printer vulnerable.

Figure 2 exemplifies the potential for predicate constraint mismatches between printer drivers and firmware with a code fragment decompiled from the Windows PSCRIPT5.DLL device driver. Function StringCchPrintfA produces the PostScript dup instruction, which is passed operand v8. Prior to calling this function, the driver performs a predicate constraint check on v8 to ensure it is within a valid range because drivers typically do not trust data coming from applications. However, the printer's PostScript interpreter (in the printer firmware) might not recheck this constraint on the generated dup when it executes the PostScript.

Data dependence constraints in PDL code. In addition to the predicate constraints (e.g., v8 $\geq$ v5), there are also data dependence constraints (e.g., $a = b$) that play critical roles in our testing. Specifically, when the driver generates instructions "op1 a,b" and "op2 c,d", operands a, b, c, and d could have data dependencies (e.g., $c = b$). Since most PDLs

```
if ( v8 >= v5 || !*v3 )
  break;
StringCchPrintfA(pszDest, 0x100ui64, "dup %d /",
                    (unsigned int)v8);
```

Figure 2. Decompiled code from the PSCRIPT5.dll rendering module

```
1 v129 = v153;
2 StringCchPrintfA(&pszDest, 0x100ui64,
3   "\n%%%IncludeResource: font %s", v153);
4 ...
5 StringCchPrintfA(&pszDest, 0x100ui64,
6   "/%s true /%s hfRedefFont", *(_QWORD*)(v3+56), v129);
```

Figure 3. Decompiled code from PSCRIPT5.dll showing a data dependence constraint

are stack-based interpreted languages, changing the context of an instruction (e.g., causing c , b) might cause serious damage to the parsing stack, such as a stack overflow.

Figure 3 shows an example of a data dependence constraint, where variable v153 is used at line 3 and variable v129 is used at line 6 to print PDL code via function StringCchPrintfA. However, there is a data dependence at line 1. If we ask the solver to break this constraint (i.e., v129 , v153) and produce new PDL code with the newly solved value, then the printer firmware could reach an inconsistent state, thereby exposing bugs.

Differences with Conventional Constraints. Prior approaches (e.g., T-Fuzz [21] or TaintScope [31]) also try to break constraints for vulnerability discovery. However, these constraints are fundamentally different from PDL constraints in two ways: (i) A conventional constraint is a check on program input. In contrast, a PDL context constraint is a check on program output. (ii) Traditional approaches negate constraints to find new paths to increase coverage (e.g., by bypassing a checksum) and hope to find vulnerable code. In contrast, in this paper we try to break PDL context constraints in order to destroy the PDL context and generate syntactically correct but semantically incorrect PDL code.

While our approach can extend to any PDL, we focus specifically on the PostScript interpreter for the printers we test due to its complexity and prevalence, and we implement our framework on the Windows platform.

3 Design

An overview of TrustScope is shown in Figure 4. There are five key components in TrustScope, corresponding with the main challenges in building TrustScope: (1) PDL code generation function identification, (2) PDL context constraints

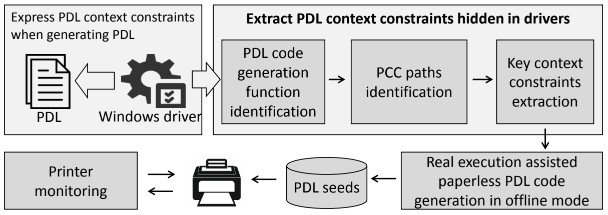
Express PDL context constraints when generating PDL
PDL
Windows driver
Extract PDL context constraints hidden in drivers
PDL code generation function identification
PCC paths identification
Key context constraints extraction
Real execution assisted paperless PDL code generation in offline mode
PDL seeds
Printer monitoring

Figure 4. An overview of TrustScope

(PCC) paths identification, (3) key context constraints extraction, (4) real execution assisted paperless PDL code generation in offline mode, and (5) printer monitoring. In the first component, we rely on static binary analysis to find the rendering modules within drivers and all of the print functions that generate PDL code within the driver. In the second component, we use static analysis and symbolic execution to identify interprocedural feasible paths. In the third and fourth component, we collect and negate key context constraints on output function variables and thereby bypass context constraints in order to generate malicious mutated PDL code. Finally, we pass our malicious inputs to printers and monitor for anomalous printer behavior in the fifth component, allowing us to detect whether we have generated an input that reveals a bug in the printer. In this section, we present detailed design for each component.

3.1 PDL Code Generation Function Identification

Printer drivers contain rendering modules that generate the PDL code used by printers. While we are specifically interested in the output functions that are the final endpoints at which actual PDL code is generated, we must first find the module within the driver that contains them.

3.1.1 Finding rendering modules. While most printer drivers use Microsoft's `PSCRIPT5.DLL` provided to generate PostScript, they also use other DLLs as plug-ins to provide custom rendering functions and modify/inject PostScript [15]. Therefore, we must find all rendering components that end up generating PostScript output.

Our analysis reveals that the generated PostScript from a driver consistently correlates to format strings within its rendering modules—a consistent pattern that is useful for identifying rendering modules. As shown in Figure 5, the generated PostScript from a driver has a clear association with the format strings contained in the rendering module to generate the PostScript. Therefore, we can print a set of sample documents and compare strings within the generated PostScript to the format strings within driver modules. Using this knowledge, we therefore use dynamic analysis to invoke the driver, print a set of sample documents, and record the corresponding PostScript. These documents contain multiple types of data to print, in order to cover as many output functions as possible in the rendering module.

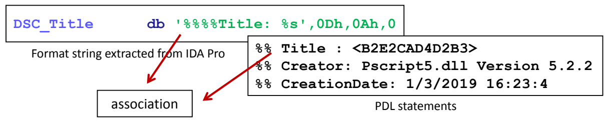
```
DSC_Title      db '%%%%Title: %s',0Dh,0Ah,0
```
```
%% Title : <B2E2CAD4D2B3>
%% Creator: Pscript5.dll Version 5.2.2
%% CreationDate: 1/3/2019 16:23:4
```

Figure 5. Identification of rendering modules

```
lea r8, aDupD ; "dup %d /"
lea rcx, [rsp+168h+Dest] ; Dest
mov edx, 100h
call sub_18003C960
```

Figure 6. Finding output function `sub_18003C960`

Next, we must compare the strings in the generated PostScript with the format strings contained within the driver modules. Such content is in natural language. An intuitive way to compare the strings would be to simply calculate the edit distance between the two: namely, between the format string that contains specifiers, and the string that eventually is output by the driver. However, this approach does not work because the printed string, which is generated by replacing the format specifiers in the format string with their actual arguments, often has significant changes (i.e., the length of the string changes and the actual arguments replace the format specifiers), as illustrated by the example in Figure 5.

We therefore adopt a fuzzy string comparison approach to match them by modifying Fuzzy Wuzzy [24], a fuzzy string matching library useful for applications such as natural language processing. Specifically, we first parse the format string and then split the format string and matching characters based on the parsing result. Then, we match each string in the set obtained by the splitting, and sum the matching values of each segment according to their respective lengths. This approach works well and eventually locates the rendering modules using the matched format strings.

3.1.2 Finding output functions. After identifying rendering modules, we must then find the output functions within those modules that generate the actual PostScript instructions. There are many common methods for identifying output functions. These often require complex data flow analysis, which makes these methods ineffective for complex programs, especially the closed-source Windows drivers that we are facing. Fortunately, through analysis, we found that the output functions in the drivers have a special structure. Generally, an output function can be defined as a function $f(\ldots, fmtstr, arg_k, arg_{k+1}, \ldots)$, and the function f replaces format string specifiers in *fmtstr* with actual arguments $(arg_k, arg_{k+1}, \ldots)$, to form the PostScript instructions and send them to the printer.

Therefore, to identify f, we could to look at the parameters to check whether any parameter to a function refers to one of the format strings in rendering modules. If so, this function is a candidate of f. An example of such a function is `sub_18003C960` as shown in Figure 6, which takes a format string specifier as an argument. However, we notice the arguments are not obvious and we have to perform symbolic execution to identify them (e.g., `lea r8, aDupD`) and set up the execution context (e.g., prepare for the arguments). Also, there could be multiple functions that access the format strings, and we should include all of them.

To identify all functions matching f, we first disassemble the device driver code, extract the control flow graphs (CFGs) of each rendering module, and retrieve the format strings we used when identifying that rendering module. After obtaining the CFG, we statically identify potentially reachable paths starting from the entry points and ending at the exit points of a function, and then perform under-constrained symbolic execution to attempt to reach each of the function calls and resolve their parameters.

For all the function calls that can be reached by symbolic execution, we then check whether any resolved parameter to a function refers to one of the format strings we previously retrieved in a callee function. If such a string is being passed to the function, we assume that the function will directly lead to generation of PostScript output, and therefore we mark it as an output function. When we check whether a callee function is an output function, the caller code needs to be included as well.

3.2 PCC Paths Identification

In order to collect the context constraints on the generated PostScript, we must identify the paths that include context constraints (PCC paths). Through our analysis, we found that PCC paths are the paths that can reach the calling points of output functions. However, identifying these paths is not a straightforward process, as attempting to directly use symbolic execution to find reachable paths inevitably leads to a path explosion. Previous efforts to avoid this path explosion include trying to filter out obviously unreachable paths with static analysis, and then performing symbolic execution on a smaller set of paths. Unfortunately, the number of paths obtained from static analysis is usually very large, still resulting in an unacceptably long analysis time. Trying to further filter the results of the static analysis without additional information could result in accidentally removing valid paths. Another potential solution is to statically find a set of potential reachable paths, and then follow these paths with concolic execution. However, this also takes too long.

Our approach. We propose a new approach for identifying interprocedural reachable paths with improved efficiency and a higher recall rate relative to simple static filtering, symbolic execution, or full program concolic execution. Figure 7 summarizes our approach, consisting of the following steps:

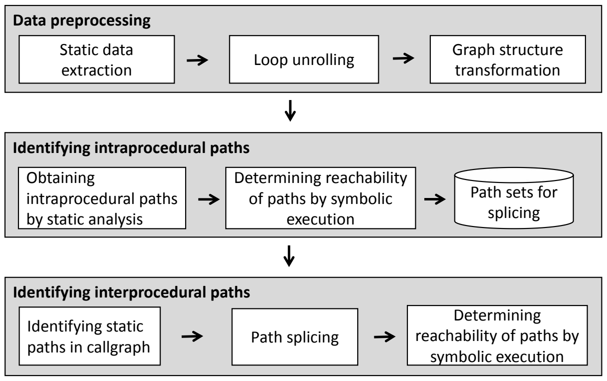
Data preprocessing
Static data extraction
Loop unrolling
Graph structure transformation
Identifying intraprocedural paths
Obtaining intraprocedural paths by static analysis
Determining reachability of paths by symbolic execution
Path sets for splicing
Identifying interprocedural paths
Identifying static paths in callgraph
Path splicing
Determining reachability of paths by symbolic execution

Figure 7. Workflow of path identification

- We provide static paths for the under-constrained symbolic execution to follow, preventing path explosion. We chose to extract the path from IDA Pro because of its superior stability for large programs.
- We reduce the burden of judging the reachability of long interprocedural paths by checking reachability of each single intraprocedural path in advance. In particular, we first extract the coarse-grained paths, then splice the candidate intraprocedural paths to get the fine-grained paths, and check the reachability to obtain the final interprocedural paths set.
- There are some differences between the static path we must analyze and the actual trace. In order to enable symbolic execution along the static path, we customize a path search plugin to solve the corner cases.

3.2.1 Data preprocessing.

Prior to our analysis, we first perform three preprocessing steps: (i) We extract callgraphs and control flow graphs from the rendering modules, which serve as the basis for our analysis. (ii) We then modify these graphs to unroll loops in the path. In order to prevent symbolic execution from getting stuck, we need to ensure that paths are finite, so we unroll each loop only once by removing back edges on both graphs. (iii) We finally must convert the paths from the form our disassembler (IDA Pro) provides into a form that works for symbolic execution, as basic blocks are not generated for function calls, and the connection between caller and callee is hidden. We ensure with our changes that symbolic execution will work correctly.

3.2.2 Identifying intraprocedural paths.

Our first objective is to extract reachable paths within each function. We start by statically analyzing each function and extracting static paths from each function in the rendering modules. The paths start from the function entry points and end at call sites. After obtaining our set of static paths, we need to perform local under-constrained symbolic execution on these paths to determine reachability for calls within these functions. In contrast to UC-KLEE, we are performing symbolic execution on a set of static paths rather than full static analysis of the entire function. Even with careful pruning of paths within functions, UC-KLEE is still susceptible to a path or memory explosion within complicated functions, while our approach avoids this explosion by only following a set of paths. Our approach is similar in concept to concolic execution, as concolic execution performs symbolic execution on concrete program paths, whereas we are performing under-constrained symbolic execution on static paths.

3.2.3 Identifying interprocedural paths.

Limiting the analysis to intraprocedural analysis is inaccurate, as it ignores whole-program flow. Therefore, our next objective is to extend our analysis to identify interprocedural paths.

Obtaining interprocedural paths. As with the intraprocedural analysis, we start with static analysis, in this case to search for possible paths to output functions. This follows the call graph of a module, giving a static path at the function level. In order to splice these paths in order, we must first extract static interprocedural paths from the call graph.

Path splicing. We replace each edge in the static interprocedural paths with the corresponding path fragments in the CFG from our intraprocedural analysis. Each edge corresponds to a set of reachable paths K_{nm} between function n and function m in the CFGs within individual functions. We select among paths reachable between pairs of functions and splice them together into full paths.

Reachability checking. After obtaining interprocedural paths by splicing, we use symbolic execution to check the reachability of the spliced paths.

3.3 Key Context Constraints Extraction

As discussed in §2, printer drivers have context constraints when generating PDL code, which is important to the PDL context. After obtaining paths to output functions in the previous section, the next step is to extract these context constraints. The context constraints are required so they can then be broken to generate deformed PDL samples.

Obstacle 1. A straightforward idea to break the PDL constraints is to negate constraints directly. However, there are many paths in the program and negating one of the path constraints may result in getting another feasible path. The new path corresponds to another feasible interval for the PDL instruction arguments. Our goal is to find an infeasible interval for the PDL instruction arguments. In other words, we need to find an infeasible path targeting an output function in order to get a value that cannot be generated under normal circumstances. The overhead of searching through every possible path would be very high.

Algorithm 1: Extracting key context constraints

```
Input: output function: $F$
Output: key context constraints set: $Q_3$
$Q_3 \leftarrow []$
$Y \leftarrow []$
$globaldoms \leftarrow \emptyset$
$domfuns \leftarrow$ identifyDominators($callgraph$, $F$)
foreach $domfun$ in $domfuns$ do
    $callpoints \leftarrow$ getCallpointsToOutput($domfun$, $F$)
    $rawcfgdom \leftarrow []$
    foreach $callpoint$ in $callpoints$ do
        $rawcfgdom[callpoint] \leftarrow$ identifyDomnators($domfun$, $callpoint$)
    end
    foreach $cfgdom$ in intersectionForAllSet($rawcfgdom$) do
        $globaldoms$.add($cfgdom$)
    end
end
$path \leftarrow$ searchPathToFun($F$)
foreach $block$ in symExeIterateBasicBlock(path) do
    if $block$ in $globaldoms$ then
        foreach $con \leftarrow$ extractCons($block$) do
            foreach $k$ in iterateAST($con$) do
                if isSymbolic(k) then
                    $Y[k]$.add($con$)
                end
            end
        end
    end
    if $block.addr == F.startAddr$ then
        foreach $m$ in getArgs(F) do
            foreach $j$ in iterateAST(m) do
                if isSymbolic(k) and $k$ in $Y$ then
                    $Q_3[F][k] \leftarrow Y[k]$
                end
            end
        end
    end
end
return $Q_3$
```

Our observation. Experimental analysis reveals that the context constraints added at dominator nodes of output functions are very important. After negating these checks, we could get the mutated argument set immediately. Since these checks exist in every path to the output function, we only need to analyze one of these paths, which is very effective and imposes very low overhead. We call these *key context constraints* and formally define the key context constraint set as $C = X \cap Y$, where

$$X = \{x \mid x \text{ is the constraint of variable } a \in A\}$$
$$Y = \{y \mid y \text{ is the constraint added at } d \in D\}$$
$$O = \{o \mid o \text{ is the output function}\}$$
$$A = \{a \mid a \text{ is the argument of function } o \in O\}$$
$$D = \{d \mid d \text{ is the dominator node of function } o \in O\}$$

Our approach tends to be much faster than conventional path search analysis because it requires $O(nh)$ time, where n is the number of dominating nodes per path and each basic block has h constraints. In contrast, conventional path search requires $O(kmh)$ time, where k is the number of paths to output function f, and m is the number of basic blocks per path. Typically $n \ll km$, speeding the analysis significantly.

Obstacle 2. To identify the dominator nodes, we must first construct a global CFG and traverse it to identify global dominator nodes. However, for large programs, the global CFGs of complex programs are usually very large. Doing actual analysis directly on the graphs is often very inefficient.

Our method. Therefore, we present a new method to narrow the recognition range of the global dominator nodes to the dominance function. In this paper, we name dominator nodes in the function call graph as dominator functions and dominator nodes in the global CFG as global dominator nodes. For the target node t, the global dominator function set is $\mathsf{DomG}(t)$, the global dominator node set for the target is $\mathsf{DomF}(t)$, and the set of blocks of $\mathsf{DomF}(t)$ is $\mathsf{DomFB}(t)$. If there is a node $x \in \mathsf{DomG}(t)$, with $x < \mathsf{DomFB}(t)$, then there is no path from x to t, so $\mathsf{DomG}(t) \subset \mathsf{DomFB}(t)$. Therefore, we can identify the dominator function first, and then identify the global dominator nodes in every dominator function, thus avoiding the construction of the global CFG.

In the following, we describe the specific technical details of how to extract the key context constraints in drivers. The algorithm is shown in algorithm 1.

3.3.1 Identifying dominator nodes. First, we statically identify the dominator node set α of output functions in the interprocedural control flow graph. The node set of interprocedural paths ending at an output function is β. The dominator nodes that we seek are $\alpha \bigcap \beta$.

3.3.2 Constructing the constraints map. After identifying dominator nodes, we collect constraints. First, we perform symbolic execution on the previously identified interprocedural reachable paths. In this process, we collect constraints added in dominator nodes. For each constraint, we iterate its abstract syntax tree (AST) and check whether each variable in the tree is a symbolic value. If it is a symbolic value k, we store it in the constraints map Y with key k.

3.3.3 Collecting key context constraints. When the symbolic execution reaches an output function, we parse the format string argument of the function and locate another function parameter m according to the format string. If the parameter is a symbolic value, then we traverse the AST of argument m and detect whether each node j on the tree is in the constraints map Y. If it exists, $Y[j]$ is the constraint set we want to collect.

3.4 Real Execution Assisted Paperless PDL Code Generation

After obtaining the context constraints, the next step is to generate test cases. Since PDLs have complex grammatical structures, it is almost impossible to directly generate samples using symbolic execution. Therefore, we use the actual execution of the printing program to assist the symbolic execution to generate test cases. In order to improve the speed of analysis, we generate test cases in offline mode; and to make the testing more practical, we propose a new method to evaluate printers without consuming paper or ink.

3.4.1 Obtaining malicious arguments. The algorithm to obtain malicious arguments is shown in algorithm 2. First, we get the output function f at output site ∂. Next, we get a path that ends at function f, and then we iterate every *arg* of function f and get the related key context constraints C. Next, we iterate through each constraint $c \in C$ and remove c from the constraints set of argument *arg*. Next, we negate c, add that to the constraints set, and pass the constraints set containing the negated constraint to the solver to find a solution. Finally, we get the mutated value n of the parameter *arg* and add n to the mutated set Q.

Algorithm 2: Obtaining malicious arguments

```
Input: output site : $g$
Output: argument mutated set: $Q$
$Q \leftarrow \{\}$
$f \leftarrow$ getOutPutFunc($g$)
$p \leftarrow$ getPathEndAtCallsite($f$)
$args \leftarrow$ getArgsOfFunc($f$)
foreach $arg \in args$ do
    $cur\_constraints \leftarrow$ extractConstraints($p$)
    $C \leftarrow$ getKeyContextConstraints($f$, $arg$)
    foreach $c \in C$ do
        $cur\_constraints$.remove($c$)
        $new\_con \leftarrow \neg c$
        $cur\_constraints$.add($new\_con$)
        $n \leftarrow$ SMTSolver.solve($cur\_constraints$)
        $Q[g][arg] \leftarrow n$
    end
end
return $Q$
```

3.4.2 Real Execution Assisted PDL Code Synthesize. After getting the mutated parameter set, the next step is to synthesize PDL code with that parameter set. Specifically, we implement a tool to make different applications print data in multiple formats. During the printing, we hook the output functions and replace the parameters. The process can be divided into the following steps.

Step 1: Rendering module selection based on the proportion of output functions. The rendering modules include the main engine and the plug-ins in the printer driver. In our implementation, we select a module to be analyzed from the main engine and plug-ins weighted by its proportion of output functions.

Step 2: Selectively hooking to maintain PDL context with low overhead. We cannot simply hook all the output functions in modules and mutate all their arguments, as unfortunately this results in both high overhead and poor results. The result likely will not pass basic lexical and grammar checks, let alone discover the bugs we are looking for. Therefore, our configurable mutation rate allows us to only select part of output functions for parameter replacement, producing mutations that are less likely to violate the language grammar.

Step 3: Type identification and argument positioning. We use the function calling convention to determine the parameters of the function, and then extract the format string parameter *fmt*. Then, we parse *fmt* according to the C format string specification, and determine the number and type of parameters. Our analysis of *fmt* lets us sequentially locate and determine the type of the parameters following the format string parameter.

Step 4: Type based replacement. The types of the function parameters that we want to replace may vary widely. We must handle each type of argument differently. For example, we must replace integer variables directly. In contrast, we need to replace the content pointed to by pointers. Strings have the additional constraint of needing to end with the delimiter '`\0`'. Since we identify the position and type of arguments in Step 3, we can easily modify each parameter according to its type.

3.4.3 Offline generation.
Testing of printers is usually done online. In this case, we need to ensure our computer is connected to printers which decreases the efficiency. However a technique called Spool Printing [23] allows print jobs transferred from a computer to be temporarily stored, and then prints them after they are transferred. This shortens printing time as it maximizes printer efficiency. By utilizing this mechanism, we can get test cases directly from the spool directory and generate test cases without connecting to the printer. In this way, we configured 60 virtual machines on OpenStack for parallel test case generation, greatly increasing the speed.

3.4.4 Testing without paper and ink consumption.
If we do not process the tested data and let the printer print the data on paper as usual, it will waste resources such as ink and paper, slow the speed of testing, and reduce the life of the printers. For more efficient and less wasteful testing, we have to achieve printing without consuming paper and ink. In the PostScript language, we found that the `showpage` command is responsible for printing the rendered data onto the paper. Therefore, we can remove `showpage` related commands to achieve only parsing, not printing, and we implement this by statically patching rendering modules or replacing the memory data.

3.5 Printer Monitoring
When a printer receives mutated PDL code, we need to know whether it triggers any abnormal behaviors. Based on our analysis, we found that printer exceptions can be classified into the following four levels:

1. interpreter errors (which do not cause much of a problem and are not the focus of our attention),
2. errors that put the directly related printer service in an abnormal state,
3. errors that put other unrelated printer services in abnormal states, and
4. errors that result in the printer being entirely unresponsive.

Prior approaches for testing IoT devices only detect limited exception levels. For instance, IoTFUZZER [3] only detects fourth level exceptions. In order to achieve effective and comprehensive monitoring of printers at different granularities, we have designed three monitoring strategies based on printer characteristics. In the following, we describe how we monitor exceptions from the second to the fourth level, respectively.

3.5.1 Monitoring based on printing-related services.
Printers usually have their own exception handling mechanisms, and not every exception can cause printers to crash. However, there are some printer services that are closely related to the interpreter. Therefore, for this level exceptions, we indirectly watch for parser exceptions by monitoring critical printing services of printers, especially the printing service of 9100 [17].

3.5.2 Monitoring based on bidirectional communication of printers.
When printing a document, users often need to know the printer's status and settings in real time. This is accomplished via bidirectional communication between the user's computer and the printer [14].

We therefore use this channel to get the status data [13]. After that, we analyze the data to see whether some programs or devices are affected by an abnormal interpreter state, such as memory or hard disk.

3.5.3 Monitoring based on ICMP packets.
For level four exceptions, we have implemented a detection method based on ICMP heartbeat packets. Using this method, we can monitor the state of the malicious PDL code causing the printer to crash and not connect.

4 Implementation
Here we present the prototype implementation of TrustScope and describe its details. At a high level, it is around 8,000 lines of code in total including Python, JavaScript, and C++. It also integrates several open source projects. For static analysis and symbolic execution, we used the IDA Pro plug-in Sark [1] to extract call graphs and CFGs, and idalink [34] to connect to IDA Pro for automatic operation with Python. Since IDA Pro and its related libraries lack good support in Python 3, we used an older version (7.8.9.26) of the symbolic execution tool angr [26], which supports Python 2. Angr was modified to meet our requirements, and many lines of Python were written for path splicing and under-constrained symbolic execution along the static paths. For malicious PDL code generation,

Table 2. Printer data type and tested applications

Application	Data types
viso 2013	vsdx
office 2013	pptx
firefox	svg, html
mspaint	png, bmp, jpg, tif, gif

we used the cross-platform analysis tool Frida [18] for API hooking to realize real-time replacement of parameters.

5 Evaluation

To evaluate TrustScope, we collected 8 IoT printers from 4 vendors including HP, Samsung, Lenovo, OKI. The data to be parsed by a printer is usually divided into three types: graphics, images, and text. We used 4 applications to print data in 9 different file formats. The detailed description is shown in Table 2, and the 9 file types listed provide a range of inputs for all three categories. In the following, we describe how each component of TrustScope performed.

5.1 Rendering Module and Output Identification

The result for identifying rendering modules is shown in column 2 in Table 3. All printers all have two rendering modules, one of which is the same for each one: `PSCRIPT5.dll`.

Having identified the rendering modules of the tested printers, we then identify the output functions in these modules. The statistics of how many output functions we identified in the tested drivers are shown in column 5 in Table 3. The intermediate result of how many functions call an output function is presented in column 4 in Table 3.

5.2 PCC Paths Identification

The most time consuming part of TrustScope is the feasible path identification, which aims to identify feasible paths from the sources that introduce the variables used in PDL, and the sinks (i.e., the printf instructions) that produce the PDL. It is a two step process (intraprocedural and then interprocedural with path splicing). TrustScope's performance statistics are shown between the 6th and 10th columns in Table 3. We can see that for each driver, TrustScope is able to identify a large number of feasible paths.

5.3 Key Context Constraints Extraction

When the feasible path identification is finished, TrustScope collects the key context constraints. The number of collected key context constraints for each module is presented in the 11th column of Table 3. We can see there are dozens of context constraints (including predicate constraints and data dependence constraints) collected for each printer.

5.4 Vulnerabilities Discovery and Case Studies

After collecting the constraints, we then used the constraint solver to solve the negated constraints and generate the new PDL code. The total number of test cases generated is presented in the last column of Table 3. The time spent on finding intraprocedural paths and interprocedural paths for each module are shown in columns 7 and 10 in Table 3.

Next, we sent the generated test cases to test how the printers would react. Encouragingly, TrustScope found vulnerabilities in all 8 tested printers, resulting in 6 new CVEs. The results are shown in Table 4.

HP OfficeJet Pro 8210. We found three test cases that caused the printer to crash and display "`ERROR CODE`" along with a system error number and an exception address on its display panel. We used `gdb` to view the instruction of the exception location, and determined that one is a memory write exception. By reverse-engineering the firmware code, we found that this exception occurs in a loop that copies data to a buffer.

Samsung CLP-680 Series. We found a pair of test cases that, when submitted to the printer sequentially, cause the printer panel to display "`internal error`", along with a red light flashing to indicate an error. To further investigate this vulnerability, we connected to the printer's serial port. From the startup log, we found that the system enters into a serious error state.

6 Future Work

In this paper, we have proposed many new methods. Any tool needing to perform format string vulnerability analysis of embedded devices can directly apply our method for identifying output functions. Current symbolic execution tools face the difficulty of extending to actual applications. We can use our proposed method to assist in the analysis of related procedures. Many program analysis methods rely on control flow or data flow graphs which are particularly large and difficult to analyze for real programs. Therefore, our method described in §3.3 could be used to narrow the scope of the analysis and improve its applicability. Our monitoring method can also be extended to other embedded device monitoring, such as service-related monitoring.

7 Related work

Printer security. Sibert et al. [27] discovered the danger of I/O operation in the PostScript file. Costin [5, 8] presented the possible attack surfaces of printers, without discussing any systematic and automated approaches to analyze these security issues. Müller et al. [16] implemented the PRET framework, which targets web vulnerabilities and relies on expert experience to write complex syntax rules. In contrast, our approach utilized the semantic information in printer

Table 3. Statistics of how each component of TrustScope performs.

Idx	Printer model	Rendering modules	Output functions		Intraprocedural paths		Interprocedural paths			#Context constraints	#Test cases
			# Callsites	Sum	# Feasible	T(h)	# Spliced	# Feasible	T(h)		
P1	Samsung CLP-680 Series	PSCRIPT5.dll	715	59	9343	16.7	6808	965	29.90	151	10946
		smc680rd.dll	47	7	6954	22.57	1700	349	42.81		
P2	Samsung ML-371x Series	PSCRIPT5.dll	715	59	9343	16.7	6808	965	29.90	111	48316
		sml371rd.dll	29	5	3033	16.83	585	194	2.81		
P3	OKI C711	PSCRIPT5.dll	715	59	9343	16.7	6808	965	29.90	122	35717
		Okb p03S.dll	30	7	494	1.34	265	144	1.10		
P4	Lenovo LJ4010DN	PSCRIPT5.dll	715	59	9343	16.7	6808	965	29.90	119	26722
		lenp1ypr.dll	1768	195	62792	58.94	19847	141	81.53		
P5	HP Color LaserJet MFP M277dw	PSCRIPT5.dll	715	59	9343	16.7	6808	965	29.90	133	69377
P6	HP LaserJet M1536dnf MFP										
P7	HP OfficeJet Pro 8210	hpcsr215.dll	839	54	21622	32.08	11633	462	62.84		
P8	HP PageWide Pro 477dw MFP										

Table 4. Summary of discovered vulnerabilities.

Issue	Printer model	Status	Descriptions
1	Samsung CLP-680 Series	CVE-2019-6335	Denial of service
2	Samsung ML-371x Series	confirmed	Not be assigned a CVE for ending of service
3	Lenovo LJ4010DN	CVE-2020-8329	Denial of service
4	Lenovo LJ4010DN	CVE-2020-8330	Denial of service
5	HP OfficeJet Pro 8210	CVE-2019-6337	Buffer Overflow Disclosure of Information
6	HP OfficeJet Pro 8210	CVE-2019-10627	Buffer Overflow Disclosure of Information
7	HP OfficeJet Pro 8210	CVE-2019-16240	Buffer Overflow Disclosure of Information
8	HP Color LaserJet MFP M277dw	CVE-2019-6337	duplicate 5
9	HP PageWide Pro 477dw MFP	CVE-2019-6337	duplicate 5
10	HP PageWide Pro 477dw MFP	CVE-2019-10627	duplicate 6
11	HP PageWide Pro 477dw MFP	CVE-2019-16240	duplicate 7
12	OKI C711	confirmed	confirmed
13-20	HP LaserJet M1536dnf MFP	reported	Not analyzed for ending of service

drivers to find memory-related vulnerabilities in printer interpreters.

IoT security. Chen et al. [2] implemented the firmware simulation framework FIRMADYNE, which can perform a full-system simulation of Linux-based firmware to detect vulnerabilities. Shoshitaishvili et al. [25] developed Firmalice, which is used to discover authentication bypassing vulnerabilities in firmware. Zaddach et al. developed the FIE [9] to discover bugs of MSP430 firmware. Jonas et al. [33] developed Avatar to combine firmware emulation with real program execution. However, these methods rely on firmware code, but usually there is no firmware code of printers and related PDL documentation. In contrast, our approach can be used to test printers without accessing firmware code. IoTFUZZER [3] mutated communication packets from mobile applications to IoT devices to discover vulnerabilities in firmware. And it can only detect whether the device is connected based on the heartbeat packets, and cannot detect vulnerabilities that cannot cause IoT devices to crash. In contrast, our tool is extracts and violates the context constraints to generate malformed PDL, exposing vulnerabilities in printer firmware due to missing security re-checks.

8 Conclusion

We have presented TrustScope, a framework that identifies memory corruptions in printers without accessing printer firmware or PDL documents. In order to achieve this, we introduce new techniques to identify PDL output functions and important semantic paths, extract the key constraints hidden in drivers, and use real execution to assist the symbolic execution in synthesizing the PDL code. We also use characteristics of printers to improve printer monitoring. We have tested TrustScope with 8 printers, and multiple vulnerabilities were discovered in all of the printers.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work is supported in part by National Natural Science Foundation of China under Grant No. U1836209, 61802394; National Key Research and Development Program of China under Grant No. 2016QY071405; and Strategic Priority Research Program of CAS under Grant No. XDC02040100. In addition, the authors from University of Texas at Dallas are supported by U.S. Awards ONR N00014-17-1-2995, DARPA FA8750-19-C-006, and NSF-IUCRC from Lockheed Martin.

References

[1] Tamir Bahar. Sark documentation. https://sark.readthedocs.io/en/latest, 2015.

[2] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *Proceedings of the 23rd Annual Network & Distributed System Security Symposium (NDSS)*, 2016.

[3] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *Proceedings of the 25th Annual Network & Distributed System Security Symposium (NDSS)*, 2018.

[4] Cisco. HP printers remote unauthorized file access information disclosure vulnerability. https://tools.cisco.com/security/center/viewAlert.x?alertId=29111, May 2013.

[5] Andrei Costin. Hacking printers: For fun and profit. Hack.lu Security Conference, 2010. http://archive.hack.lu/2010/Costin-HackingPrintersForFunAndProfit-slides.pdf.

[6] Andrei Costin. Hacking MFPs. 28th Chaos Communication Congress (CCC), 2011. https://fahrplan.events.ccc.de/congress/2011/Fahrplan/track/Hacking/4871.en.html.

[7] Andrei Costin. PostScript: Danger ahead?! HITB Security Conference, 2012. https://www.slideshare.net/phdays/postscript-danger-ahead.

[8] Andrei Costin. Andrei Costin papers. http://andreicostin.com/papers, 2016.

[9] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the 22nd USENIX Security Symposium*, pages 463–478, 2013.

[10] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 206–215, 2008.

[11] Patricia Hernandez. Someone hacked printers worldwide, urging people to subscribe to PewDiePie. November 2018. https://www.theverge.com/2018/11/30/18119576/pewdiepie-printer-hack-t-series-youtube.

[12] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering (TSE)*, 2019.

[13] Microsoft. Bidirectional communication error codes, 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/print/bidi-error-codes.

[14] Microsoft. Bidirectional communication schema reference. https://docs.microsoft.com/windows-hardware/drivers/print/bidi-communications-schema-reference, 2017.

[15] Microsoft. Introduction to rendering plug-ins, 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/print/introduction-to-rendering-plug-ins.

[16] Jens Müller, Vladislav Mladenov, Juraj Somorovsky, and Jörg Schwenk. Sok: Exploiting network printers. In *Proceedings of the 38th IEEE Symposium on Security & Privacy (S&P)*, pages 213–230, 2017.

[17] OKI Europe. What is port 9100? https://okiprinting-en-gb.custhelp.com/app/answers/detail/a_id/334/~/what-is-port-9100%3F, October 2019.

[18] oleavr. Frida – a world-class dynamic instrumentation framework. https://www.frida.re, 2019.

[19] OpenRCE. sulley: A pure-Python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley, 2019.

[20] Peach Tech. Peach fuzzer: Discover unknown vulnerabilities. https://www.peach.tech, 2020.

[21] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by program transformation. In *Proceedings of the 39th IEEE Symposium on Security & Privacy (S&P)*, pages 697–710, 2018.

[22] Redrain and Min Zheng. A ghost from PostScript. Ruxcon Security Conference, 2017. https://ruxcon.org.au/assets/2017/slides/hong-ps-and-gs-ruxcon2017.pdf.

[23] Ricoh. Spool printing. http://support.ricoh.com/bb_v1oi/pub_e/oi_view/0001038/0001038577/view/software/unv/0065.htm, 2009.

[24] SeatGeek. fuzzywuzzy: Fuzzy string matching in Python. https://github.com/seatgeek/fuzzywuzzy, 2020.

[25] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice – automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium (NDSS)*, 2015.

[26] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P)*, 2016.

[27] W. Olin Sibert. Malicious data and computer security. In *Proceedings of the 19th National Information Systems Security Conference*, 1996.

[28] Statista. Printer share by vendor worldwide from 2015 to 2019. https://www.statista.com/statistics/541347/worldwide-printer-market-vendor-shares.

[29] Iain Thomson. Hacker: I made 160,000 printers spew out ASCII art around the world. *The Register*, February 2017. https://www.theregister.co.uk/2017/02/06/hacker_160000_printers.

[30] Undocumented Printing Wiki. Page description languages [undocumented printing]. http://www.undocprint.org/formats/$page_d$escription$_l$anguages.

[31] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31st IEEE Symposium on Security & Privacy (S&P)*, pages 497–512, 2010.

[32] Wikipedia. Page description language. https://en.wikipedia.org/wiki/$Page_d$escription$_l$anguage.

[33] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 21st Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[34] zardus. Some glue facilitating remote use of IDA (the Interactive DisAssembler) Python API. https://github.com/zardus/idalink, 2018.