Not All Data are Created Equal: Data and Pointer Prioritization for Scalable Protection Against Data-Oriented Attacks

Salman Ahmed
IBM Research [*]

Hans Liljestrand
University of Waterloo

Hani Jamjoom
IBM Research

Matthew Hicks
Virginia Tech

N. Asokan
University of Waterloo

Danfeng (Daphne) Yao
Virginia Tech

Abstract

Data-oriented attacks are becoming increasingly realistic and effective against the state-of-the-art defenses in most operating systems. These attacks manipulate memory-resident data objects (data and pointers) without changing the control flow of a program. Software and hardware-based countermeasures for protecting data and pointers suffer from performance bottlenecks due to excessive instrumentation of all data objects. In this work, we propose a Data and Pointer Prioritization (DPP) framework utilizing rule-based heuristics to identify sensitive memory objects automatically from an application and protect only those sensitive data utilizing existing countermeasures. We evaluate the correctness of our framework using the Linux Flaw Project dataset, Juliet Test Suite, and five real-world programs (used for demonstrating data-oriented attacks). Our experiments show that DPP can identify vulnerable data objects from our tested applications by prioritizing as few as only 3–4% of total data objects. Our evaluation of the SPEC CPU2017 Integer benchmark suite shows that DPP-enabled AddressSanitizer (ASan) can improve performance (in terms of throughput) by ∼1.6x and reduce run-time overhead by ∼70% compared to the default ASan while protecting all the prioritized data objects.

1 Introduction

With the advancement toward practical code pointer protection countermeasures [14,32,33,39,52] and practical Control-Flow Integrity (CFI) [26, 42, 68], we anticipate a shift towards the manipulation of sensitive memory-resident data and pointers as attack vectors. In recent research, we observe an uptick in Data-Oriented Attacks (DOAs), also known as non-control data attacks [17,30,31,36,43,54,55,65,67] even though DOAs were introduced more than a decade ago [10]. DOAs achieve their malicious goals by changing program behavior without violating the normal flow of a program as specified by its Control-Flow Graph (CFG). Conceptually, DOAs [3,10,30,36,67] can modify all kinds of data to change program behavior such as leaking sensitive information [6] or performing privilege escalations [16]. However, corruption of data pointers [14] is often desirable as it allows leaking the address space layout [23, 58], stitching gadgets in Data-Oriented Programming (DOP)-based attacks [31], and performing stack-based [10] or heap-based [60] exploitation.

To stop attackers from manipulating memory-resident data and their pointers (hereafter referred to simply as "data objects" for the sake of brevity), researchers have proposed both software and hardware-based countermeasures. Software-based countermeasures such as Data-Flow Integrity (DFI) [9], Data-Space Randomization (DSR) [4, 7, 53], and memory tagging techniques [44, 45] suffer from run-time overhead (ranging from 42% to 116% [4,7,9,44,45,53]) due to inter-procedural DFI, encryption, and masking. On the other hand, hardware-based countermeasures (*e.g.*, HDFI [63], Intel's Control-Flow Enforcement Technology (CET) [32], and ARM Pointer Authentication (PA) [52]) are efficient, but in general, limited to certain platforms. Furthermore, the overhead is non-negligible, *e.g.*, ARM PA costs on average around 19-26% [1] [24, 40] overhead for protecting data pointers.

The main reason for this run-time overhead is the huge number of data objects, on average ∼100x compared to code pointers in an application [2]. One solution for reducing this overhead is to identify the *sensitive* data objects and prioritize them for protection, rather than all. Besides reducing overhead, it is also extremely important to know what the most sensitive data objects are so that existing defense mechanisms can put extra effort to protect them. There are two approaches to identifying sensitive data. One approach is manual, and the other one is best effort semi-automatic. Prior works [30, 49, 50] have suggested the *manual* earmarking of sensitive data. However, it is time-consuming and error-prone. A few best-effort semi-automated approaches [36, 43] can

[*] A significant amount of the work was conducted while the first author was a graduate student at Virginia Tech.

[1] Evaluated using software-based simulation of PA.

[2] We obtain the numbers by counting the code pointers, and data variables and pointers from eight real-world applications

determine the criticality or sensitiveness of data. But these works require traces of data accesses, including traces for both normal and violating execution. As a result, these solutions may have scalability issues due to the need for huge and relevant execution and access traces. Besides, exercising all the violating execution paths is challenging. Furthermore, these techniques may not be application-agnostic and cannot work with existing countermeasures. Thus, there is a need for a scalable and platform- or application-agnostic automated approach for identifying and prioritizing sensitive data.

In this paper, we automate the identification and prioritization of sensitive data objects through our DPP framework. DPP uses common and widely applicable vulnerability patterns to identify and prioritize sensitive data objects. These vulnerability patterns enable DPP to prevent unknown and future DOAs. DPP is also platform- and application-agnostic and adaptable with existing countermeasures. DPP uses rule-based heuristics to identify sensitive data objects.

We address two key challenges. First, it is challenging to find a representative set of rules with comprehensive coverage since DOAs are constantly evolving. To address the challenge regarding the coverage and representativeness of rules, we extract the rules by abstracting exploits into common vulnerability patterns. These patterns are applicable to many exploits and future or unknown attacks. Second, it is also challenging to evaluate the accuracy of the rules. To the best of our knowledge, there is no ground truth dataset of sensitive data objects. Thus, to evaluate the accuracy and effectiveness of our rule-based heuristics, we rely on the datasets from the Linux Flaw Project [19], Juliet Test Suite [5, 47], and five DOA exploits against real-world programs. We manually investigate the datasets and exploits to identify vulnerable objects using knowledge from CVE description, exploits, online sources, browsing code, and code comments.

We apply our rule-based heuristics to a program's data flow graph for identifying sensitive data objects. We implemented these heuristics in LLVM as a set of analysis passes. For performance comparison, we leverage the vanilla ASan [59] and a DPP-enabled modified version to instrument all and the prioritized data objects, respectively. We used the SPEC CPU2017 benchmark suite, five real-world applications, one library, and one lightweight benchmark to evaluate the performance and execution time improvement due to DPP.

Our key contributions of this work are as follows.

- We present an adaptable and platform- or application-agnostic DPP framework for automatically identifying and prioritizing sensitive data objects. We define seven rule-based heuristics for the identification and prioritization (Section 3).

- Our evaluation shows that DPP can detect vulnerable data objects from the Linux Flaw Project [19], Juliet Test Suite [5,47], and five real-world programs by prioritizing as few as only 3–4% of total data objects (Section 5.2).

- Our performance evaluation through the SPEC CPU2017 Integer benchmark suite shows that DPP improves performance by ∼1.6x in terms of throughput and reduces run-time by ∼70% compared to ASan (Section 5.4). Our code is available at https://github.com/salmanyam/dpp-llvm

2 Background & Threat Model

In this section, we discuss different attack techniques for DOAs and existing software- and hardware-based defenses for them. We then discuss the threat model and assumptions.

2.1 Data-Oriented Attacks

The power of DOAs was not realized until recently when control-oriented attacks have become unreliable due to many practical software and hardware-assisted defenses. Chen *et al.* first demonstrated the power of data-oriented attacks in 2005 [10]. However, recently data-oriented attacks have gained momentum and researchers have presented data-oriented attacks in many widely-used applications including server applications [30,31,43] and browsers [36,54]. Besides, various automated data-oriented exploit generation tools such as FLOWSTITCH [30], BOPC [35], STEROIDS [51], and LIMBO [57] can automatically generate data-oriented attacks with a little manual effort. We refer readers to [11, 12] for the systematization of DOAs.

DOAs can be as simple as corrupting a variable. However, they can be powerful enough to aid control-data attacks in the presence of fully precise 'static' CFI [8] or achieve Turing-complete expressiveness [31]. The key to DOAs is strategic manipulations of two kinds of data: *i)* data variables or objects, and *ii)* data pointers. The overwriting of security critical data variables or objects leads to change program behavior or inadvertent data leaks.

Data pointers as attack vectors (like code pointers [1, 61]) have recently gained attackers' interest. DOP [31] used the address of some non-control data pointers to select and stitch DOP gadgets. Chen *et al.* [10] corrupted a data pointer in the ghttpd HTTP server through a stack buffer overflow to bypass security checks of input strings. COOP [56] utilized a C++ object to hijack the *virtual table pointer* of a C++ object and constructed an exploit using the virtual functions as gadgets. Heap-based exploitations such as the *House of Spirit* attack [60] on Glibc also manipulate a data pointer returned by malloc(). Besides, the corruption of data pointers can leak information through software side channels such as pointer probing [23] and timing side-channel attacks [58].

2.2 Defenses against Data-Oriented Attacks

Both software- and hardware-based mechanisms can protect a program from data-oriented attacks through pointer-based bound-checking [18,33,44], object-based bound checking [28, 59], pointer integrity [52], DFI [9, 63], and DSR [4, 7, 53]. Some defense mechanisms (*e.g.*, *YARRA* [55], *PT-Rand* [17], Orpheus [13], etc.) especially target data-oriented attacks.

On one hand, it is encouraging to have many defense mechanisms to tackle DOAs. On the other hand, performance and run-time issues with hardware extensions set a high bar for deployment. For example, software-based memory safety protection incurs run-time overhead ranging from 48% to 116% [20, 21, 44–46], DFI overhead ranging from 42% to 103% [9, 62] and pointer integrity around 20% [14]. *HardBound* [18] can lower the overhead to 9% on average for pointers in C, but with architectural support. Object-based approaches such as ASan can introduce up to 200% overhead [59]. Hardware-assisted AddressSanitizer (HWASan) [28] improves performance via a slight compromise in security. Memory Protection Extensions (MPX) can incur on average 50% overhead [48] due to loading and storing bound metadata. Due to this high run-time overhead, most of these protection mechanisms, in general, are not practical to protect memory-resident data objects. Thus, it is necessary to identify and prioritize the sensitive data objects to improve these protection mechanisms and make them practical.

2.3 Threat Model

The goal of this work is to identify and protect critical and sensitive data objects, variables, and pointers to prevent data-oriented attacks. Thus, our threat model is on par and consistent with the requirements and assumptions of existing DOAs [10, 30, 31, 35, 57]. The key requirement of DOAs is arbitrary manipulation or write capability of memory data through one or more memory vulnerabilities to change a program's execution behavior or obtain sub-attack goals such as gadget stitching or selection. Thus, in our threat model, we assume a powerful adversary who can exploit vulnerabilities to control or corrupt memory. We also assume the integrity of program code that is protected by Data Execution Prevention (DEP) or W⊕X. Protections such as Address Space Layout Randomization (ASLR) [66] or variations [27, 29, 38], full or partial Relocation Read-Only, Code-Pointer Integrity (CPI) [14, 32, 33, 39, 52], CFI [26, 42, 68], and memory protection can be present in a system to prevent control-oriented attacks. Since we rely on the existing defenses (ARM PA [52], Intel MPX [33], Softbound [44], ASan [59], etc.) for protecting sensitive or critical memory data, we assume these defenses are correct and secure. For example, the metadata used by Softbound [44] or ASan [59] is protected or cannot be manipulated. Besides, we assume that attackers have no access to higher privilege levels, or the underlying operating systems are safe and protected. For example, the kernel stores the PA keys, so we assume that attackers cannot access the keys.

3 Data and Pointer Prioritization

Data and Pointer Prioritization (DPP) is a generic framework for automatically identifying and prioritizing sensitive data objects. Both identifying the sensitive data objects and prioritizing them are challenging tasks. We start this section with the definition of sensitive data objects and summary of DPP's operations. We then elaborate each operation of DPP as well as our rule-based heuristics for prioritization.

Sensitive data object. A program can have many data objects that are possibly attacker-controlled through external manipulation. However, all attacker-controlled objects may not be equally useful for attackers. Data objects or pointers that allow attackers to change a program flow or select their chosen execution path are useful. We call such prioritized input-dependent objects, and pointers to them, *sensitive*.

Our DPP framework automatically identifies and ranks sensitive data objects. DPP has two major operations: *i)* taint–tracking to identify possibly attacker-controlled data objects, and *ii)* detection and prioritization of objects that might facilitate exploitation if corrupted by an attacker. We utilize system or library I/O functions that receive input from external sources (*e.g.*, network, file system, or user input) as initial taint sources. Since we consider reading from file system as external input, reading from any configuration files is also considered as external input. For example, Listing 1 shows the overwrite of a user's ID (*pw->pw_uid*). A malicious user can override this ID with a vulnerability in the program, to gain root privilege [10]. To capture such vulnerabilities, we consider any file system objects, including configuration files, to be outside the Trusted Computing Base (TCB) and mark any data objects read from them as tainted. In this case, we taint the user's ID, *i.e.*, *pw->pw_uid* since the value is set from a configuration file.

```
FILE ∗ getdatasock( ... ) {
  ...
  seteuid(0);
  setsockopt( ... );
  ...
  seteuid(pw−>pw_uid);
  ...
}
```

Listing 1: Overwriting user's ID to gain root privilege [10].

After identifying the taint sources, we then propagate the taint throughout the program to identify any data objects with data-dependence on possibly attacker-controlled inputs. From the set of tainted objects, we use rule-based heuristics to prioritize those that are sensitive. We can then protect these sensitive objects through memory-protection defenses such as ASan [59], Softbound [44], CETS [45], Intel MPX [33], or pointer integrity schemes such as ARM PA [52].

3.1 Rule-based Prioritization

DPP uses rule-based heuristics to automatically detect sensitive data objects that could potentially lead to vulnerabilities. The rules are data-driven and solidly based on a large number of facts extracted from existing exploits and Common Vulnerabilities and Exposuress (CVEs). We analyzed various types of data-oriented exploits demonstrated in security literature that corrupt data objects and pointers. We also carefully examined CVEs related to memory corruption vulnerabilities. We categorize the type of these corruptions into four categories: *i)* Control alteration – where the corruption of data objects and their pointers aims to alter a program behavior, *ii)* Proximity based – where pointers aim to corrupt nearby buffers, *iii)* Erroneous – where data pointers have bad casting, violate intended pointer semantics, and cause memory corruptions; or data objects have erroneous bound conditions, and *iv)* Unguarded – where pointers are allocated as unbounded.

Based on these corruptions from at least nine data-oriented exploits [10, 23, 30, 31, 58] and 20 CVE disclosures (*e.g.*, CVE-2001-0820, CVE-2006-5815, CVE-2017-9430, CVE-2018-6151, CVE-2018-10111, and CVE-2021-23017), we formulate seven heuristic rules shown in Table 1. These are further categorized on whether the rule is for Allocation-based Protection (AP) or Pointer-integrity Protection (PP). AP ensures the integrity of data objects, while PP safeguards the integrity of pointers to these objects. For instance, Rule 6 prioritizes objects that lack bounds-checking, and is therefore applicable to AP but not PP. We also categorize them into four categories based on their actions (Table 1).

Table 1: We use exploit- and vulnerability-driven heuristics to realize rules for identifying and prioritizing input-dependent data or pointers.

Rule #	Category	Short Description	Protection	Example CVE
Rule 1	Control alteration	Data objects/pointers in predicates may alter program behavior	PP/AP	CVE-2006-5815
Rule 2	Control alteration	Data pointers used in loops may alter program flow or leak sensitive information	PP/AP	CVE-2006-5815
Rule 3	Proximity-based	Data pointers that are near to data buffers	PP/AP	CVE-2002-1496
Rule 4	Proximity-based	Data objects or pointers used in vulnerable functions	AP	CVE-2021-31226
Rule 5	Erroneous	Data pointers that have been cast to different types	AP	CVE-2018-6151
Rule 6	Erroneous	Data objects that have out-of-bound access	AP	CVE-2021-21773
Rule 7	Unguarded	Pointers that have unbounded allocations	AP	CVE-2020-11612

Rule 1 prioritizes allocations and pointers that are used as predicates for conditional execution and could thus allow an attacker to manipulate the control-flow of a program. Listing 2 shows an example from the ProFTPD DOP attacks [31] where the pointers *cp* and *pbuf* are dependent on user input and are used in the condition on Line 10.

```
1  char *sreplace(char *s, ...) { //char *s is the taint source in this function
2      ... // taint flow: *s -> *src -> *pbuf and *cp
3      char *src = s, *cp, **rptr, *pbuf = NULL;
4      size_t rlen = 0, blen; cp = buf;
5      ...
6      while( *src ) {
7          //pbuf and cp pointers are set based on the user-provided *src
8      }
9      ...
10     if((cp - pbuf + 1) > blen) { // off-by-one error
11         cp = pbuf + blen - 1; ...
12     } /* Overflow Check */
13     *cp++ = *src++;
14     ...
15 }
```

Listing 2: Usage of data pointer *cp* and *pbuf* in the condition at Line 10 in ProFTPD v1.3.0 (CVE-2006-5815).

Rule 2 prioritizes objects and pointers used in a loop. Listing 3 shows an example from a DOP attack on ProFTPD v1.3.0 [31], where the manipulation of pointer *src* (*i.e.*, `for(; *src && n > 1; n--)` at Line 3 in Listing 3) can control the execution of a loop to produce DOP assignment gadgets at Line 4 (*i.e.*, `*d++ = *src++`). This loop-based heuristics covers such custom *strcpy*-type functions, but also other common patterns that utilize loops to stitch and select gadgets used in DOP. Data-manipulation within loops has also been used to leak confidential information, *e.g.*, to break ASLR [23, 58].

```
1 char *sstrncpy(char *dest, const char *src, size_t n) {
2     register char *d = dest;
3     for (; *src && n > 1; n--)
4         *d++ = *src++;
5 }
```

Listing 3: Loop manipulation for DOP [31] gadgets in ProFTPD v1.3.0 through data pointer overwrite (CVE-2006-5815).

Rule 3 detects objects that include an addressable buffer—typically an array—that is followed by a pointer. The exploitation of this pattern is attractive for two reasons. First, because the overflow and target are in the same buffer, there is no dependence on the overall memory layout of the program. And second, due to limitations of allocation-based bounds checking [44], such overflows can be performed even in the presence of common memory-protection schemes (such as the allocation-based ASan [59]). Listing 4 shows an example of such an exploitable code pattern.

```
struct mystruct_s {
    char buffer[64]; // Can overflow other fields without violating allocation
      boundary
    void (*f_ptr)();
};
```

Listing 4: Example of Rule 3, where a buffer within a structure could overwrite a sensitive pointer (*f_ptr*) within the same structure.

Rule 4 prioritizes data objects used by vulnerable library functions such as *strcpy()*, *memcpy()*, *gets()*, *strncpy()*, and *sprintf()*. Listing 5 shows a vulnerability (CVE-2017-9430) in dnstracer v1.9 caused by a vulnerable *strcpy* call at Line 4.

```
1 int main(int argc, char **argv) {
2     while ( (ch = getopt(argc, argv, "4cCoq:r:S:s:t:v")) != -1 ) {...}
3     ...
4     strcpy(argv0, argv[0]); // argv[0] depends on user input!
5     ...
```

Listing 5: Rule 4 detects the use of commonly exploited functions, such the call to *strcpy()* on Line 4.

Rule 5 prioritizes data pointers that have been cast from one type to another type. Incorrect casts can cause type-confusion attacks that allow over-reads or -writes by accessing an object of incorrect type, or incorrect function calls by invoking methods of a wrong type. For example, Out-of-Bounds (OOB) memory read (CVE-2018-6151) happens in Google Chrome version prior to 66.0.3359.117 due to a bad cast where an object cast to an unexpected type causes a bad cast.

Rule 6 detects memory dereferences that could lead to an OOB due to pointer arithmetic or incorrect indices that cannot be statically shown safe. A simple example of OOB access is to read from or write to an array beyond its allocation range. Such vulnerabilities exist in real-world applications such as in Nginx v0.6.18 – v1.20.0 where an OOB write happens due to an off-by-one error (CVE-2021-23017).

Rule 7 prioritizes objects that are allocated to a size without bounds checked. In Listing 6, Line 6 shows an example of an allocation of unbounded size in the GEGL version 0.3.32 that could lead to a Denial-of-Service (DOS) attack by exhausting all memory (CVE-2018-10111).

```
1 static gboolean render_rectangle (GeglProcessor *processor) {
2     ...
3     GeglRectangle *dr = processor->dirty_rectangles->data;
4     ...
5     guchar *buf; // Create pointer for buffer
6     buf = g_malloc (dr->width * dr->height * pxsize); // Allocate buffer
7     g_assert (buf);
8     ...
9 }
```

Listing 6: Rule 7 prioritizes unbounded allocations, such as this example from CVE-2018-10111 where an unbounded allocation in GEGL leads to DOS.

There are two main challenges to applying the rule-based heuristics for identifying sensitive memory objects. First, it is difficult to know the complete and representative set of rules. Second, DPP needs an efficient technique to track the data flow. We discuss these challenges as follows.

Ch1. Completeness and representativeness of the rules. Anticipating future attacks is difficult as attackers' capabilities constantly evolve, making it challenging to ensure the completeness of rule-based heuristics. A minor change in the attack pattern can render signature-based or end-to-end rules ineffective. To enhance rule representativeness, we extract them by identifying key exploit strategies such as manipulation of conditions and loops, positional corruptions by data pointers, vulnerable library functions, and unbounded allocations. Our identified strategies have been experimentally confirmed to work well (Section 5), but additional rules can be easily added to our set for future attacks.

Ch2. Data flow tracking. Since DPP aims to apply the rules to input-dependent data objects, it needs to track the flow of data objects and their pointers efficiently throughout a program, starting from the input sources. A crucial task is the determination of the points-to set of a pointer. Luckily, we have existing techniques to perform this heavy lifting task. In this work, we use Static Value Flow (SVF) [64] with Anderson [2] pointer analysis to perform the data flow tracking (Section 3.2). We use SVF as this graph-based implementation is efficient to traverse and query.

3.2 Data Flow Tracking

The purpose of data flow tracking is to identify all attacker-controlled data (*i.e.*, all the reachable data from external inputs) and taint them. We apply the rule-based heuristics on a tainted data flow graph of a program. To construct the data flow graph, we utilize the interprocedural SVF analysis [64]. SVF performs its analysis on the LLVM Intermediate Representation (IR) to construct the Static Value Flow Graph (SVFG). SVF first converts LLVM IR instructions into a Program Assignment Graph (PAG). A PAG has two types of nodes: *i)* Value Pointer Node (ValPN), and *ii)* Object Pointer Node (ObjPN). A ValPN represents an LLVM value that is a pointer and an ObjPN represents an abstract memory object (*i.e.*, the address-taken variable of an IR pointer). An edge in the PAG represents the constraints between nodes by capturing the `address-of`, `load`, `store`, and `copy` associations. To perform pointer analysis, SVF starts with a copy of PAG called the constraint graph. SVF solves the constraints in the graph by converting each `load` and `store` constraints to `copy` constraints. Once the constraint resolution is done, SVF then constructs the SVFG. To do so, SVF annotates the potential use of a variable at loads, potential definitions and uses of the variable at stores, inter-procedural uses and definitions at call sites, and parameter passing or return at function entries or exits, where the variable is pointed by a top-level pointer. SVF obtains the points-to set of a top-level pointer using Andersen's points-to analysis [3] [2]. Finally, SVF constructs the SVFG by converting all the address-taken variables to Single Static Assignment (SSA) form, merging multiple definitions using `phi` instructions, and connecting the definition-uses for each SSA variable. We refer readers to [64] for details.

[3] Practical considerations must be made when choosing a points-to analysis technique considering the trade-offs between speed and precision. We evaluate and discuss these trade-offs in Section 5.

3.3 Taint Analysis

Our taint analysis identifies all the taint sources from a program and propagates the taints throughout the program considering simple and complex taint propagation scenarios.

Identification of tainted sources. Vulnerable data objects or pointers must depend on external input channels (*e.g.*, network, file system, or user input) so they are susceptible for external manipulation. These correspond to standard library functions such as *read*, *recv*, *getc*, *recvmsg*, *scanf*, *fscanf*, *fread*, and *fgets*. To optimize analysis, we also identify common wrapper functions that always lead to input functions in the standard libraries. We refer to both standard library functions and common wrapper functions as *input-reading functions*. We also consider the *main* function as an input-reading because its parameters are user-controlled. For any input-reading function, we mark all non-constant input values and the return value as taint sources. If a parameter or returned value is a pointer, then the tainting process starts from the points-to set of the pointer parameter or returned value.

Propagation of tainted data. We propagate the taint through the SVFG by traversing all successors of a tainted node. However, we need to address the following limitations of the SVFG for the completeness of the taint propagation process:

1. SVFG does not taint dynamically allocated memory when the arguments passed to the allocator are tainted.

2. SVFG does not taint the underlying object when a pointer is indexed with a tainted value or a result of pointer arithmetic with tainted inputs.

3. SVFG does not track when a function (*e.g.*, *memcpy*) stores value from one parameter to another.

To address the first limitation, we simply taint the return result of an allocation function if the arguments are tainted. To address the second, we identify all the address-taken data objects and one or more index-variables or pointers used to access the data objects. If one or more index-variables or pointers are tainted, we taint the data objects and obtain the points-to sets of the newly tainted data objects. If any pointers in the obtained points-to sets are not tainted, we start the tainting process for the pointer. We address the third by identifying all function calls that store the value of their second parameter to the first parameter. Such function call instructions include *memcpy*, *memmove*, *strcpy*, *strncpy*, *strcat*, and their variations. If the second parameter is tainted, then we taint the first parameter and its points-to set. Algorithm 1 in Appendix A.1 shows the pseudocode of the tainting process in the SVFG.

As mentioned above, we apply the rule-based heuristics on a tainted data flow graph. We discuss this application of rules on a tainted graph in Section 4. We also demonstrate in Section 5 that these rules are simple, but powerful for coverage, security, and performance.

4 Implementation

We implemented DPP [4] in LLVM 12. To perform the pointer analysis and data-flow construction, we utilize the SVF [5] [64] tool. We performed our analysis on top of the LLVM bitcode. We obtained a single whole program bitcode file using the whole program LLVM tool [6]. We then apply the above mentioned rules on a tainted SVFG to identify sensitive data objects or pointers. When a rule flags a pointer, our technique also flags the data object where the pointer points to.

To discuss the implementation of the rules, we use an example program in Figure 1. Figure 2 shows the SVFG of the example program in Figure 1. The example program has three memory objects (two local and one dynamic). The corresponding memory allocation nodes in the SVFG are the hexagonal nodes ①, ⑤, and ㉗ (Figure 2). The rectangular nodes show how the values of the memory objects flow through different IR instructions.

4.1 Rule Implementation

Each rule is implemented as a separate analysis pass that is then collected to a joint analysis result. This facilitates the addition of new emerging exploitation patterns without additional changes to protection that rely on the prioritization for selectively instrumenting code.

Rules 1, 2, 4, and 5 utilize the SVF analysis to identify tainted address-taken pointers. We then extract the alias set of those pointers. For example, one of the tainted pointers in Figure 2 is node ⑤, which has an alias set consisting of nodes {5, 11, 22}. We then use the LLVM's built-in definition-use chains to create a list of all uses involving any pointer in the alias set. In the case of node ⑤, this gives us nodes {5, 6, 7, 8, 11, 12, 13, 14, 22, 25, 26}. Once we get this usage list, we apply the following techniques to apply Rule 1, 2, 4, and 5.

- To implement Rule 1, *i.e.*, usage of data objects and pointers in predicates, we check if any node from the usage list has a compare instruction implying that it can alter control-flow or data.

- To implement Rule 2, *i.e.*, usage of data pointers in loops, we check if any node from the usage list has a load/store/compare instruction and has been used in a loop's predecessor, header, and latch. We analyze loops using the LLVM *LoopAnalysis* [7].

- To implement Rule 4, *i.e.*, usage of data pointers in vulnerable functions, we check if any arguments of a vulnerable function are from the usage list.

[4] available at https://github.com/salmanyam/dpp-llvm
[5] https://github.com/svf-tools/SVF
[6] https://github.com/travitch/whole-program-llvm
[7] https://llvm.org/doxygen/classllvm_1_1LoopAnalysis.html

```c
void testcase(int ts) {
  char buf[10];

  char *s = (char *) malloc(ts * sizeof(char));
  for(int i=0; i < ts; i++) s[i] = i + ’0’;

  if (s[0]==’a’) printf("%d\n", s[ts−1]);

  fscanf(stdin, "%s", s); //’s’ marked as tainted
  memcpy(buf, s, strlen(s)); // sink
  printf("%s\n", buf);
}

int main(int argc, char **argv) {
  int size;
  scanf("%d", &size); //’size’ marked as tainted
  testcase(size);

  return 0
}
```

Figure 1: Example C program.

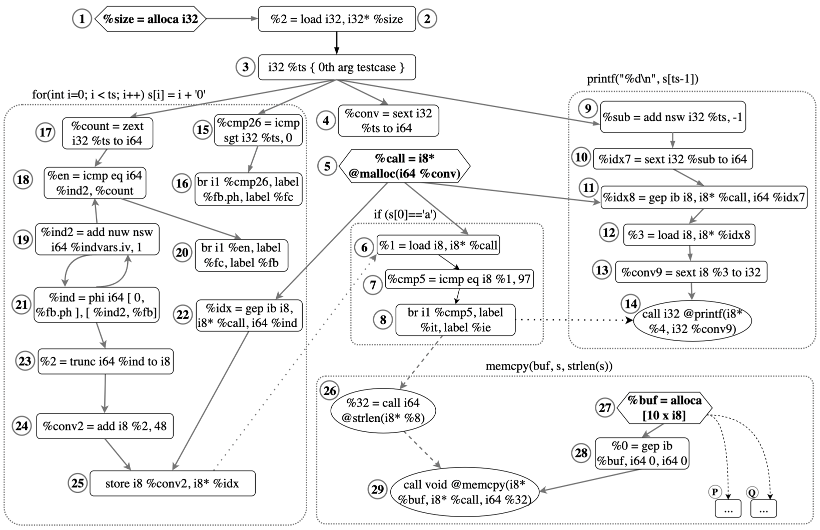
1
%size = alloca i32
%2 = load i32, i32* %size
2
3
i32 %ts { 0th arg testcase }
for(int i=0; i < ts; i++) s[i] = i + '0'
17
%count = zext i32 %ts to i64
15
%cmp26 = icmp sgt i32 %ts, 0
18
%en = icmp eq i64 %ind2, %count
16
br i1 %cmp26, label %fb.ph, label %fc
19
%ind2 = add nuw nsw i64 %indvars.iv, 1
20
br i1 %en, label %fc, label %fb
21
%ind = phi i64 [0, %fb.ph], [%ind2, %fb]
22
%idx = gep ib i8, i8* %call, i64 %ind
23
%2 = trunc i64 %ind to i8
24
%conv2 = add i8 %2, 48
25
store i8 %conv2, i8* %idx
4
%conv = sext i32 %ts to i64
5
%call = i8* @malloc(i64 %conv)
if (s[0]=='a')
6
%1 = load i8, i8* %call
7
%cmp5 = icmp eq i8 %1, 97
8
br i1 %cmp5, label %it, label %ie
printf("%d\n", s[ts-1])
9
%sub = add nsw i32 %ts, -1
10
%idx7 = sext i32 %sub to i64
11
%idx8 = gep ib i8, i8* %call, i64 %idx7
12
%3 = load i8, i8* %idx8
13
%conv9 = sext i8 %3 to i32
14
call i32 @printf(i8* %4, i32 %conv9)
memcpy(buf, s, strlen(s))
26
%32 = call i64 @strlen(i8* %8)
27
%buf = alloca [10 x i8]
28
%0 = gep ib %buf, i64 0, i64 0
29
call void @memcpy(i8* %buf, i8* %call, i64 %32)
P
...
Q
...

Figure 2: SVFG of the motivating example. *gep ib* → getelementptr inbounds.

- To implement Rule 5, *i.e.*, possibly unsafe pointer casts, we check if any node from the usage list is a `bitcast` that is used to realize casts in the LLVM IR. We filtered out trivial casting like from and to *char* *, and any non-pointer casts. We check the incompatibility of checking the size of the pointed-to source and destination types based on the data layout information in the LLVM IR.

We implement Rule 3, *i.e.*, data pointers possibly vulnerable to inter-allocation overflows, by checking all the tainted stack allocations (`alloca` instructions) in a function and all the tainted global variables for structures that contain arrays followed by pointers. When found, we mark the corresponding pointer as sensitive.

To implement Rule 6, *i.e.*, on possible OOB access, we apply three optimizations to filter out safe tainted objects or pointers from all the tainted objects and pointers. First, we only consider operands to `load`, `store`, and `call` SVF nodes that deal with memory accesses. Second, we apply LLVM's stack safety analysis [8] to filter out allocations that are free from memory access bugs. Third, we ignore nodes that we can prove statically safe using LLVM's ObjectSizeOffsetVisitor [9] class. We flag the remaining operands as sensitive.

To implement Rule 7, *i.e.*, unbounded memory allocations, we first use SVF to retrieve the SVFG nodes corresponding to dynamic allocations. We filter out the allocation nodes that are untainted or in dead functions. Once we get the allocation sites or nodes in the SVFG, we obtain the corresponding nodes in the Interprocedural Control-Flow Graph (ICFG). We perform a backward search from the obtained ICFG nodes to fetch `cmp` instructions to check if the argument used in a memory allocation function is bounded. However, we need to address three key challenges: *i)* how to deal with path explosion due to the backward search, *ii)* how to deal with loops in the ICFG, and *iii)* how to determine the right `cmp` instruction(s) that are relevant to the argument of a memory allocation function.

We address the path explosion issue by parameterizing how many paths and how far in a path to explore. We remove all the edges that create loops in the ICFG. We address the third challenge by determining if a `cmp` instruction and the argument of an allocation function have a *common ancestor* in the SVFG. The key idea is that the argument and `cmp` instruction will be the descendants of a node if the `cmp` instruction operates on the argument. We refer readers to the detailed discussion of how we implement the *common ancestor* solution in SVFG in Appendix A.2.

5 Evaluation

We evaluate DPP by answering the following questions:

1. How capable is DPP for prioritizing security critical data objects? (Section 5.1)

2. How effectively can DPP rank sensitive data objects? What is the impact of individual rules on the accuracy of the prioritization? (Section 5.2)

3. How sensitive are the leftover data objects? (Section 5.3)

4. How much performance improvement can DPP enable? How are the DPP's end results amenable for implementing a live defense? (Section 5.4)

[8] https://llvm.org/docs/StackSafetyAnalysis.html

[9] https://llvm.org/doxygen/classllvm_1_1ObjectSizeOffsetVisitor.html

Table 2: Linux Flaw Project [19].

CVE	Type	Application	ASan (default)	ASan + DPP
CVE-2006-0539	heap-buffer-overflow	fcron-3.0.0	✓	✓
CVE-2006-2362	buffer-overflow	binutils-2.15	✓	✓
CVE-2009-1759	stack-overflow	ctorrent-dnh3.3.2	✓	✓
CVE-2009-2285	heap-buffer-overflow	tiff-3.8.2	✓	✓
CVE-2010-2481	out-of-order	tiff-3.9.2	×	×
CVE-2010-2482	null-pointer-dereference	tiff-3.9.2	✓	✓
CVE-2013-4243	heap-buffer-overflow	tiff-4.0.1	✓	✓
CVE-2013-4473	stack-smashing	poppler-0.24.2	✓	✓
CVE-2013-4474	stack-buffer-overflow	poppler-0.24.2	✓	✓
CVE-2014-1912	heap-buffer-overflow	Python-3.1.5	×	×
CVE-2015-8668	heap-buffer-overflow	tiff-4.0.1	✓	✓
CVE-2016-10095	stack-buffer-overflow	tiff-4.0.7	✓	✓
CVE-2016-10271	heap-buffer-overflow	tiff-4.0.7	✓	✓
CVE-2017-12858	heap-use-after-free	libzip-1.2.0	✓	✓
CVE-2018-9138	stack-overflow	binutils-2.29	✓	✓

5.1 Security Evaluation

To evaluate the capability of DPP to prioritize sensitive data objects and their pointers, we run DPP on the Linux Flaw Project [19] and Juliet Test Suite [5,47]. We only consider the CVEs and test cases from the Linux Flaw Project and Juliet Test Suite which are related to memory errors. We then utilize ASan to protect only the prioritized data objects to see if we can detect all the memory errors. If DPP does not prioritize a sensitive data object that is related to a memory error from the Linux Flaw Project or Juliet Test Suite, then ASan does not protect that data object. Hence, we cannot detect that memory error. Table 2 and Table 3 show the results for DPP's prioritization capability for the Linux Flaw Project and Juliet Test Suite, respectively. To reproduce all the memory errors in the Linux Flaw Project (Table 2), we use Ubuntu 18.04. For each of the five Common Weakness Enumeration (CWE) categories for the Juliet Test Suite (Table 3), we select test cases that depend on external inputs.

As shown in Table 2 and Table 3, ASan with DPP can detect all the memory errors from both datasets the same as the default ASan can. However, ASan with DPP can detect memory errors by only protecting the prioritized data objects. This evaluation demonstrates the capability of DPP for prioritizing security-critical data objects.

Our key evaluation datasets (*i.e.*, the Linux Flaw Project and Juliet Test Suite) are completely distinct from our rule dataset. Our rules do not have any knowledge about the programs in the Linux Flaw Project or Juliet Test Suite. We evaluated DPP utilizing 11 unique programs from the Linux Flaw Project and 720 test cases from the Juliet Test Suite. For sanity checking in terms of accuracy and effectiveness, we also include the exploit programs (*i.e.*, the rule dataset) in some of our evaluations (*e.g.*, Table 4). Thus, the evaluation datasets are distinct from the rule dataset and our evaluation does not suffer from overfitting (more discussion on the generalization of our rules in Section 6).

Table 3: Juliet Test Suite [5,47].

Type	Total tested cases	ASan (default)	ASan + DPP
CWE121_Stack_Based_Buffer_Overflow	144	144	144
CWE122_Heap_Based_Buffer_Overflow	144	144	144
CWE124_Buffer_Underwrite	144	144	144
CWE126_Buffer_Overread	144	144	144
CWE127_Buffer_Underread	144	144	144

5.2 Prioritization Efficacy

Efficacy of any memory protection scheme is challenging to evaluate because new attacks are a priori unknown [22,65]. However, we can evaluate DPP against known vulnerabilities. This shows that the rule-based approach of DPP can be used to cover known exploitation techniques, and so, suggest that it can be extended to cover new techniques uncovered in the future. To evaluate the efficacy of DPP, *i.e.*, how well DPP can rank sensitive data objects, we identify the vulnerable data objects from the Linux Flaw Project [19], Juliet Test Suite [5,47], and five DOA exploits against real-world programs. The vulnerabilities cover both C and C++ programs and five CWE categories. We manually investigate the datasets and exploits to identify vulnerable objects using knowledge from CVE description, exploits, online sources, browsing code, and code comments. Table 8 in Appendix A.3 shows the vulnerable data objects or pointers extracted from the datasets and exploits.

Table 4 shows the programs and test cases used in our evaluation, including the number of all and prioritized data objects in these programs and test cases. The first column of Table 4 shows the names of the vulnerable data objects ($\mathrm{V_{objs}}$) [10]. The fifth column shows the percentage of prioritized data objects with respect to the number of all data objects. On average, the percentages of the prioritized data objects for the Linux Flaw Project [19] and DOA Exploit Dataset is ∼41% and ∼31%, respectively. DPP considers the rest ∼59% for the Linux Flaw Project and ∼69% for the DOA Exploit Dataset as safe, indicating they are either not manipulatable or if manipulated, unlikely to facilitate exploitation. Hence, DPP filters out these data objects. The percentage of the prioritized data objects in the Juliet Test Suite [11] [5,47] is much lower than the Linux Flaw Project [19] or DOA Exploit Dataset because the test cases in the Juliet Test Suite are much smaller than the other two datasets. The Juliet Test Suite has small programs designed to evaluate accuracy of a static analysis where the test cases are implemented by adding minor changes to a small base program. Most test cases follow a similar format, and a few include data objects that have external dependencies.

[10] For locations of these vulnerable data objects in the program, refer to Table 8 in Appendix A.3.

[11] The numbers of total data objects, prioritized data objects, and the rank of vulnerable objects for each CWE category are presented in Table 4 as the average of the total number of test cases in that CWE category.

Table 4: The number and percentage of top k prioritized objects needed for detecting vulnerable data objects (V_{obj}) from the Linux Flaw Project [19], Juliet Test Suite [5,47], and five DOA exploits.

Vulnerable data object (V_{obj})	**Program name**	**# of data objs**	**# of prio. data objs**	**% of prio. data objs**	**Rank of V_{obj} in prio. objs (kth)**	**% of top k w.r.t. prio. objs**	**% of top k w.r.t. all objs**
			Linux Flaw Project				
char *argv[]	fcron-3.0.0	110	31	28%	14	45%	13%
char sym[17]	binutils-2.15	4664	2349	50%	43	2%	1%
char path[MAXPATHLEN]	ctorrent-dnh3.3.2	1379	672	49%	8	1%	1%
TIFF* tif	tiff-3.8.2	1405	731	52%	186	25%	13%
TIFF* tif	tiff-3.9.2	1445	761	53%	37	5%	3%
TIFF* tif	tiff-4.0.1	1857	806	43%	23	3%	1%
char pathName[4096] char *destFileName	poppler-0.24.2	11539	3293	29%	19	1%	0.2%
TIFF* tif	tiff-4.0.1	1857	806	43%	23	3%	1%
zip_buffer_t *buffer	libzip-1.2.0	878	255	29%	15	6%	2%
			Average→	**∼41%**		**∼10%**	**∼4%**
			DOA Exploit Dataset				
char *ptr	ghttpd-1.4	77	14	18%	4	24%	5%
struct passwd *pw	wu-ftpd-2.6.0	590	378	64%	34	10%	6%
char *src char *cp	proftpd-1.3.0	5070	1313	26%	236 28	18% 2%	5% 1%
apr_array_header_t *log_format;	httpd-2.4.7	6754	587	9%	77	13%	1%
char *pPostData;	nullhttpd-0.5.0	100	37	37%	1	2%	1%
			Average→	**∼31%**		**∼12%**	**∼3%**
			Juliet Test Suite				
int buffer[10]	CWE121	27	3	11%	2	67%	7%
int * buffer	CWE122	28	3	11%	1	33%	4%
int buffer[10]	CWE124	27	3	11%	2	67%	7%
int buffer[10]	CWE126	27	3	11%	2	67%	7%
int buffer[10]	CWE127	27	3	11%	2	67%	7%
			Average→	**∼11%**		**∼60%**	**∼6%**

We also rank the prioritized data objects based on the number of rules they match. If ties occur in the ranking of data objects, we use the number of pointers to a data object to break the ties. This ranking allows us to understand the efficacy of DPP for prioritizing V_{objs}. To estimate the efficacy, we determine the rank of each of the V_{obj} in our dataset indicated in the first column of Table 4. The sixth (6th) column in Table 4 shows the rank (k) of a V_{obj} within the prioritized data objects of the program the V_{obj} belongs to. For example, the rank of the data object pointed by `char *ptr` in *ghttpd-1.4* is four ($k = 4$), *i.e.*, `char *ptr` is the fourth (4th) data object in the ranked and prioritized data objects. The rank four of `char *ptr` indicates that the top four prioritized data objects include the `char *ptr` V_{obj}. We compute the percentages of this top k data objects with respect to the prioritized and all data objects in a program or test case. The seventh and eighth columns of Table 4 show the top k percentages.

On average, we noticed that DPP prioritizes 31-41% of all data objects (excluding Juliet). However, the prioritized data objects are also sorted based on how many rules flag the data objects. After the sorting, we observed that all vulnerable data objects are found within the top 10-12% of the prioritized data objects. These top 10-12% of the prioritized data objects are 3-4% of all data objects. For the Juliet Test Suite [5,47] test cases, we found that the V_{objs} are in the top 60% of the prioritized data objects which are in the top 6% with respect to all data objects. The top k percentage is higher (60%) for the Juliet Test Suite compared to the other two due to a few objects in Juliet with external dependencies indicating a few numbers of objects to prioritize.

To understand the impact of individual rules on the accuracy of the prioritization, we perform an experiment using the Linux Flaw Project [19] dataset to see what rules prioritize the V_{objs}. Table 5 shows the result. As shown in the table, we noticed that Rules 1, 2, 4, and 6 have the most impact on prioritizing the V_{objs} from the Linux Flaw Project dataset. The rules capture the condition and loop manipulation, exploitation of data pointers through vulnerable library functions, and out-of-bound access. These four rules cover the three (control alteration, proximity based, and erroneous) of the four categories of the rules. The result indicates the generality and applicability of some rules over others due to the nature of the vulnerabilities that make the data objects vulnerable. The seemingly less impactful rules are interesting as they could indicate some classes of attacks are just harder to exploit. This also indicates a potential limitation of our dataset as we need

Table 5: Impact of individual rules on the accuracy of prioritization.

CVE	Data Variable	Program	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7
CVE-2006-0539	char *argv[]	fcron-3.0.0	✓	✓	×	✓	×	×	×
CVE-2006-2362	char sym[17]	binutils-2.15	✓	✓	×	✓	×	✓	×
CVE-2009-1759	char path[MAXPATHLEN]	ctorrent-dnh3.3.2	✓	✓	×	✓	×	✓	×
CVE-2009-2285	TIFF* tif	tiff-3.8.2	✓	×	×	×	×	×	×
CVE-2010-2482	TIFF* tif	tiff-3.9.2	✓	×	×	✓	×	×	×
CVE-2013-4243	TIFF *tif	tiff-4.0.1	✓	×	×	✓	×	×	×
CVE-2013-4473	char pathName[4096]	poppler-0.24.2	✓	✓	×	✓	×	✓	×
CVE-2013-4474	char *destFileName	poppler-0.24.2	✓	✓	×	✓	×	✓	×
CVE-2015-8668	TIFF *tif	tiff-4.0.1	✓	×	×	✓	×	×	×
CVE-2017-12858	zip_buffer_t *buffer	libzip-1.2.0	✓	✓	×	✓	×	✓	×

a broader benchmark to realize the usefulness of all rules.

5.3 Sensitiveness of Leftover Data Objects

To evaluate the data objects left unprioritized by DPP are indeed insensitive, we performed our evaluation through case studies. We manually analyzed *ghttpd-1.4* and *nullhttpd-0.5.0*. We choose these two applications because these applications are small compared to other applications. We performed the analysis in two steps: *i)* analyzing all untainted data objects, and *ii)* analyzing all tainted but unprioritized data objects. Our analysis shows that the rest 63 data objects (out of 77) in *ghttpd-1.4* are untainted either because they are constant or derived from another constant. These data objects are used to store file names, server names, document roots, directory headers, mime types, string formats, dates, and time. We observed similar scenarios in *nullhttpd-0.5.0*. A total of 56 out of 100 data objects were untainted due to their use in storing local/gm time and constant. Seven data objects were tainted due to their use in *fopen* function. However, we found that these objects are also derived from constants as these were just file names and had no uses that satisfy our rules. It is important to note that DPP marks the content of a file as sensitive even if the file name object is left untainted or unflagged.

5.4 Performance Evaluation

We use ASan [12] [59] for evaluating the performance impact of our prioritization. ASan detects memory-related errors by instrumenting data objects by inserting extra code and enforcing the instrumented code through a runtime library. It can detect various memory-related errors such as out-of-bounds access, use-after-free, use-after-return, use-after-scope, and double-free, and invalid-free. Though ASan is primarily used as a sanitizer to catch bugs during testing rather than as a practical mitigation practice due to its high performance or memory overhead and security issues with shadow memory protection, we chose ASan over the other live defenses due to its compatibility and stability, and because it still can give us a good estimation of the comparative performance improvement due to our prioritization. To evaluate such performance improvement, we use the SPEC CPU2017 benchmark as well as five real-world applications, one library, and one lightweight benchmark. We compare the throughput, run-time, and benchmark specific metric (*e.g.*, CPU Index) of the benchmark or applications with vanilla ASan and DPP-enabled ASan. To implement the DPP-enabled ASan, we modify the vanilla ASan to enable the option to instrument only the prioritized data objects.

Performance improvement of the SPEC CPU2017 benchmark due to DPP. To evaluate the performance improvement through the SPEC CPU2017 benchmark, we utilize the Integer benchmark suite. The Integer benchmark suite contains 10 benchmark programs. Out of these 10 benchmarks, one benchmark (*648.exchange2_s*) is written using Fortran and DPP has no impact on the benchmark. Thus, we discarded *648.exchange2_s* and conducted our experiments on the remaining nine benchmarks.

We prepared three versions of each of the benchmark programs: *i)* a *baseline* version without any instrumentation, *ii)* an *asan* version instrumented with the default ASan, and *iii)* an *asan+dpp* version instrumented with the DPP-enabled ASan. We ran each version of the benchmark multiple times and measured the throughput. We used Ubuntu 18.04 with 16 CPUs and 64 Gigabytes of memory. We used four CPU threads to run each benchmark. Figure 3 shows the percentage of throughput of each benchmark with *asan* and *asan+dpp* with respect to the *baseline* benchmark. That means we normalize the throughput of *asan* and *asan+dpp* versions of the benchmarks with respect to the throughput of the *baseline* benchmarks and compute the percentages.

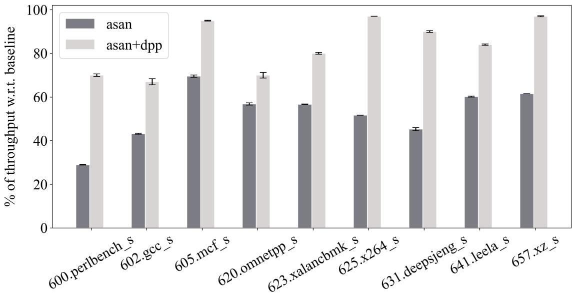

Figure 3: Throughput (% with respect to the baseline) of SPEC CPU2017 benchmarks fully instrumented with ASan (*asan*) and with DPP-prioritized ASan (*asan+dpp*).

[12] https://clang.llvm.org/docs/AddressSanitizer.html

As can be seen from Figure 3, the percentage of throughput is more than 80% in five out of nine benchmarks with the DPP-enabled ASan (*i.e.*, *asan+dpp*) versions. Three of them have throughput close to the *baseline* throughput. On average, the throughput for benchmarks with *asan+dpp* is ∼83%. On the other hand, the average throughput for benchmarks with *asan* versions is ∼53%, indicating ∼1.6x performance improvement by DPP-enabled ASan compared to the default ASan. This result also indicates that ASan reduces the throughput by ∼47% on average compared to the baseline, where the reduction is only around ∼17% with DPP-enabled ASan. This result implies that DPP-enabled ASan incurs 30% less overhead than what the default ASan does. This improvement of 30% overhead reduction by DPP-enabled ASan is ∼64% with respect to the overhead of the default ASan.

Impact of DPP on workload run-time. To observe DPP's impact on workload run-time, we also measure the total workload run-time for three versions (*i.e.*, *baseline*, *asan*, and *asan+dpp*) of each benchmark in SPEC CPU2017. Figure 4 shows the cumulative workload run-time for the *baseline*, *asan*, and *asan+dpp*. As can be seen from the figure, the cumulative workload run-time for the default ASan is almost 2*x* (1.9*x* to be exact) with respect to the *baseline*. The run-time is reduced to only ∼ 1.2*x* when we incorporate DPP in ASan, which reduces the run-time overhead by around 70% compared to the default ASan.

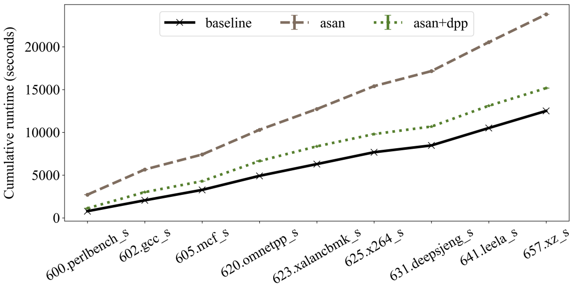

Figure 4: Impact of DPP on reducing the workload run-time overhead of the SPEC CPU2017 benchmark suite.

Performance improvement of real-world applications. Additionally, we measured DPP's performance improvement in five real-world applications including a browser, one browser library, and one lightweight benchmark: nginx, httpd, lighttpd, postgres, midori, gtk library, and nbench. The *midori* browser uses the *gtk* library for creating graphical user interfaces. We compared the throughput and run-time of the *asan* and *asan+dpp* versions of each application, as well as the CPU Index for nbench, using benchmark workloads generated by wrk [35] for the web servers and sysbench [49] for postgres. We used the Lite Brite [13] benchmark for the browser and libgtk's own benchmark for the gtk library. We chose the Midori browser as our focus for evaluation due to the fast and straightforward compilation process of DPP-enabled ASan. Incorporating DPP-enabled ASan compilation is exceptionally slow for standard commercial browsers like Chromium and Firefox, primarily due to their large size and high complexity, which pose challenges for the points-to analysis. Developing a feasible points-to analysis (*e.g.*, utilizing the latest type-based analysis [41]) and an optimized DPP framework for such large projects remains an important future work.

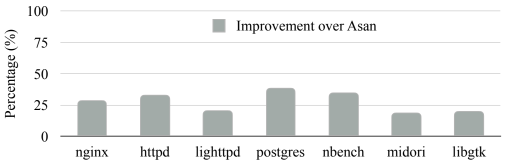

Figure 5: DPP's performance improvement over asan for real-world applications.

Figure 5 shows the percentage of performance improvement of the applications, benchmark, and library over the ASan. We found that DPP-enabled ASan improves the performance by 28% on average in this experiment compared to the ASan. Besides, we observe DPP-enabled ASan incurs ∼19% less overhead than the default ASan (∼44%) for the web servers, database, and *nbench*. With respect to ASan, this overhead reduction is ∼43% by the DPP-enabled ASan.

The overhead reduction in this scenario (*i.e.*, with the real-world applications or libraries) is also slightly less than what we observe in the SPEC CPU2017 benchmark suite, likely because the type of these applications is different from the CPU2017 benchmark. Besides the CPU-intensive tasks, these applications also have network-intensive operations, whereas most operations in the CPU2017 benchmark suite are CPU-intensive. However, we also observe that only 3-4 CPUs out of eight CPUs are saturated with 100% CPU utilization due to mostly being network-intensive applications and limitations of the benchmarking tools. Network latency is not also significant due to the benchmark tools running for 60 seconds in the same machine where the applications are running. This is a limitation of our setup and the benchmarking tools. Nonetheless, this setup gives us a useful measurement to compare our technique with the default ASan.

In addition, to understand how the individual rules impact performance, we measure the throughput of *nginx* by applying a single rule at a time. Figure 6 shows the throughput ratio of different rules after normalizing a throughput with the lowest one. According to the result, the ratios are very similar (and within 1.0∼1.1). The reason is that the numbers of prioritized data objects in various rules are close to each other and are very few compared to the total number of data objects in *nginx*. However, due to the difference in the number of prioritized data objects in different rules, we still can observe some differences (though not significant) in the throughput ratios across the rules. We show this impact of individual rules in

[13] https://testdrive-archive.azurewebsites.net/Performance/LiteBrite/

detail using the SPEC CPU2017 benchmark in Figure 9 in Appendix A.4. It is important to mention that multiple rules enable us the opportunity to identify top-ranked data objects.

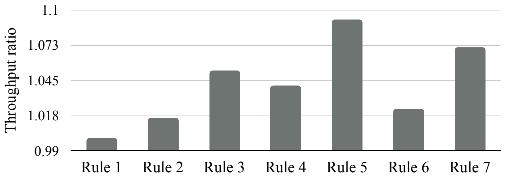

Figure 6: Impact of individual rules on performance of nginx.

Impact of various analysis approaches on DPP. To understand how the choice of the various points-to analyses impacts our approach, we prioritize vulnerable data objects from Linux Flaw Project using three points-to-analysis choices: Andersen, Steensgaard, and Flow Sensitive. We compare the accuracy and time for each choice. Table 6 in Appendix A.4 shows the result. Steensgaard is the fastest of the three because it requires less than 47% time than Andersen. However, DPP with Steensgaard fails to prioritize the most vulnerable objects (misses seven out of ten vulnerable objects). That means, Steensgaard increases false negatives due to imprecise points-to-analysis. Flow Sensitive, on the other hand, is the slowest as it requires 78% more time than Andersen but manages to prioritize all cases in programs (except where it did not fail due to memory issues marked by a dash in Table 6 in Appendix A.4). Andersen seems a balanced option between the two options. It is neither the fastest nor the most accurate, but it is reasonable for a reliable points-to analysis. Note that we use Andersen's points-to analysis in our previous results.

Cost of data flow analysis and rules. We provide the costs incurred by three key parts of DPP: (*i*) the pointer and data flow analysis (*i.e.*, SVF), (*ii*) taint analysis, and (*iii*) individual rules. For an average-sized program (∼60kloc) in our evaluation (*e.g.*, proftpd), SVF takes ∼23 seconds whereas our analysis (taint and individual rules) takes ∼72 seconds. However, this cost also depends on how many pointers are in a program. For example, SVF and the rest analysis take around 2 and 4 seconds, respectively, to complete for a reasonably large program (httpd, ∼260kloc). We provide the detailed results in Table 7 in Appendix A.4.

Summary of Key Takeaways

❶ *More than 95% of data objects in a real-world program do not need protection*. DPP identifies potentially sensitive data objects by identifying and prioritizing top 3–4% data objects from real-world applications.

❷ *Common and widely applicable rules are simple, but powerful for coverage*. We extracted the common and widely applicable rules from existing exploits. These rules identify all the vulnerable objects from the Linux Flaw Project or Juliet Test Suite with zero knowledge about the data objects from the datasets.

❸ *DPP is tunable in the security, usability, and performance dimensions*. DPP enables the trade-offs between accuracy and performance. We can make DPP tunable in the security (false negative), usability (false positive), and performance dimensions. DPP achieves ∼1.6x performance improvement compared to the default ASan. We can also tune the performance based on the level of security needed.

6 Discussion

Our DPP is a generic framework intended for any data-oriented defense mechanisms. Our specific prototype builds on ASan but does not have to. We chose ASan over the other live defenses due to its compatibility and stability, and because it gives us a good estimation of the efficacy of our prioritization. However, despite its benefits, DPP still faces an average overhead of around 20% (when overhead from SPEC CPU2017 benchmark and real-world applications are considered), while protecting 100% of the prioritized data objects. This level of overhead may not be practical for deploying DPP as a live defense. Nevertheless, DPP does offer the flexibility to selectively protect data objects from the prioritized list. In fact, as demonstrated in Table 4, the minimum number of data objects requiring protection can be as low as 3-4% of the total data objects. This newfound capability presents an opportunity to significantly reduce DPP's overhead to a practical range when performance and security do not compete with each other.

Our approach differs from Clang Static Analyzer [14] , which focuses on detecting bugs, while we prioritize instrumentation for input-dependent and potentially flawed code. Therefore, DPP complements static analyzers, and using both tools can reduce code errors and prioritize runtime instrumentation. Our approach also offers flexibility in balancing performance and security by adjusting the number of protected data objects.

The efficacy of a rule depends on its ability to detect a wide variety of attacks. To achieve the coverage and generalization of rules, we extract the rules by breaking down exploits and identifying key exploit strategies such as manipulation of conditions and loops through data pointers, utilizing the position of a data pointer (*e.g.*, adjacent to a data buffer), utilizing vulnerable library functions, finding unbounded allocations and so on. Exploits may use a strategy or a combination of strategies from our identified strategies to construct their attacks. Our identified strategies are our best effort rules that tend to work well, as we experimentally confirmed in our evaluation (Section 5). However, new rules may be added to our rule set in the future to cover additional classes of attacks.

We evaluated DPP's security guarantees using empirical evidence instead of theoretical guarantees commonly found in system security literature. We tested DPP using programs containing eight types of memory issues and found that our

[14] https://clang.llvm.org/docs/ClangStaticAnalyzer.html

rules detected all these errors in the Linux Flaw Project and Juliet Test Suite datasets (Tables 2 and 3), despite having no prior knowledge of these datasets.

Our prioritization approach has some limitations. For example, our approach may generate false positives in some conditions. An input-dependent object might be assumed correct due to the existence of bounds checks, however because the bounds check is itself not validated for correctness this might lead to a false positive. Such incomplete or wrong bound-conditions are a common source of many real-world vulnerabilities (*e.g.*, CVE-2006-5815 and CVE-2021-23017). More extensive analysis, for instance, utilizing symbolic execution [37] could be employed to address this and, in general, improve accuracy of analysis.

Currently, our rules do not capture the complete end-to-end logic for temporal vulnerabilities like use-after-free, double-free, and invalid-free. This is due to the complexity involved, which requires logging and analyzing the temporal usage of pointers, making our rules more complex. However, our rules still prioritize and protect objects or pointers vulnerable to temporal vulnerabilities through ASan or other methods, as long as they match the rules. Nevertheless, our approach may miss sensitive objects if we overlook sensitive variables (apart from pointers) followed by a buffer and if the list of vulnerable library functions is incomplete. Researching the identification of simple and sensitive primitive data variables would be valuable, and a broader benchmark is needed to fully assess the effectiveness of our rules. Additionally, integrating DPP's outcomes into live defense, such as ARM PA, can be achieved by selectively enabling PA instructions for DPP-related pointers through LLVM transformation and backend passes. Yet, further work is necessary to address integration challenges and evaluate the approach's robustness.

7 Additional Related Work

Researchers have designed techniques to protect sensitive data. Palit *et al.* designed a compiler-level defense that protects critical data [49, 50]. However, they manually annotate the sensitive data. Similarly, FlowStitch [30] performed the automation of data-oriented attacks using predefined critical data. Our work automates the identification and prioritization of sensitive data. A few best-effort and semi-automated techniques [36, 43] also determined the critical data. For example, Jia *et al.* [36] determined the decision-making data by recording the execution of two traces with normal execution and violated execution, and observing the data that get modified and change executions. Access-driven trace data [43] are also useful to determine and understand the critical data and their structures. However, these solutions have limited scalability due to the need for huge and relevant execution and access traces. On the other hand, Pathfinder [54] can automatically navigate to sensitive data from a leaked data pointer. However, it does not indicate how to determine or label sensitive data. Researchers also proposed metrics to identify and prioritize significant software weaknesses [25].

Object- and pointer-based countermeasures have become popular. Object-based protection mechanisms such as ASan [59] and HWASan [28] protect objects by storing metadata in a *shadow memory* area and marking red-zones with blocks of poisoned memory between adjacent objects. An access to these red-zones indicates an overflow and terminates the programs at runtime. *PointGuard* [15] and ARM PA [52] can protect the integrity of pointers. To improve security and performance of ARM PA, researchers have also proposed the secured [40] and efficient [34] use of PA. PARTS [40] used a type-based modifier to prevent pointer reuse attacks. PACTight [34] selectively protects pointers that are reachable from code pointers. Data space randomization or diversification [4, 7, 53] is another defense direction for protecting sensitive data. For example, CoDaRR [53] could be an alternative and orthogonal approach to DPP.

8 Conclusion

In this paper, we proposed an automatic prioritization framework for identifying and protecting sensitive memory-resident data objects to prevent data-oriented attacks. The overall results suggest that the simple rule-based heuristics are effective. Our exploit and vulnerability-driven rule-based heuristics give the flexibility to add new rules when necessary. Our experimental evaluations using the Linux Flaw Project [19], Juliet Test Suite [5, 47], and real-world programs showed the successful identification of vulnerable data objects. DPP improves performance by $\sim$1.6x and reduces run-time overhead by 70% compared to ASan. We presented that not all data objects in an application require protection and we can prioritize them. This proposed prioritization scheme is new and makes our approach different from the conventional protection paradigm.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their support and valuable feedback for this work. This work has been supported by the Office of Naval Research under Grant N00014-22-1-2057, Virginia Commonwealth Cyber Initiative (CCI), the Natural Sciences and Engineering Research Council of Canada (RGPIN-2020-04744), and Intel Labs via the Private-AI consortium.

References

[1] Salman Ahmed, Ya Xiao, Kevin Z Snow, Gang Tan, Fabian Monrose, and Danfeng Yao. Methodologies for quantifying (re-) randomization security and timing under jit-rop. In *Proceedings of the 2020 ACM SIGSAC*

Conference on Computer and Communications Security, pages 1803–1820, 2020.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[3] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *2007 IEEE Symposium on Security and Privacy (SP'07)*, pages 246–251. IEEE, 2007.

[4] Sandeep Bhatkar and R. Sekar. Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer, 2008.

[5] Paul E. Black. A software assurance reference dataset: Thousands of programs with known bugs. *Journal of Research of the National Institute of Standards and Technology*, 123:123005, April 2018.

[6] The Heartbleed Bug. http://heartbleed.com. Accessed June 01, 2023.

[7] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. Data randomization. Technical report, TR-2008-120, Microsoft Research, 2008.

[8] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium*, pages 161–176, 2015.

[9] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[10] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, volume 5, 2005.

[11] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–36, 2021.

[12] Long Cheng, Hans Liljestrand, Md Salman Ahmed, Thomas Nyman, Trent Jaeger, N Asokan, and Danfeng Daphne Yao. Exploitation techniques and defenses for data-oriented attacks. In *2019 IEEE Secure Development, SecDev 2019*, pages 114–128, 2019.

[13] Long Cheng, Ke Tian, and Danfeng Yao. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 315–326, 2017.

[14] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuardTM: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[15] Stanley Crispin Cowan, Seth Richard Arnold, Steven Michael Beattie, and Perry Michael Wagle. Pointguard: method and system for protecting programs against pointer corruption attacks, July 6 2010. US Patent 7,752,459.

[16] Daniel Moghimi. Subverting without EIP. https://moghimi.org/blog/subverting-without-eip.html, 2014. Accessed January 06, 2021.

[17] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *NDSS*, 2017.

[18] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[19] Dongliang Mu. Linux Flaw Project. https://github.com/mudongliang/LinuxFlaw. Accessed June 01, 2023.

[20] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142, 2016.

[21] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *NDSS*, 2017.

[22] Thomas F. Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 2018.

[23] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy*, pages 781–796. IEEE, 2015.

[24] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054, 2021.

[25] Carlos Cardoso Galhardo, Peter Mell, Irena Bojanova, and Assane Gueye. Measurements of the most significant software security weaknesses. In *Annual Computer Security Applications Conference*, pages 154–164, 2020.

[26] Masoud Ghaffarinia and Kevin W. Hamlen. Binary control-flow trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1009–1022, 2019.

[27] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, pages 475–490, 2012.

[28] Hardware-assisted AddressSanitizer. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html. Accessed June 01, 2023.

[29] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.

[30] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, 2015.

[31] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[32] Intel. Control-flow enforcement technology preview. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html. Accessed June 01, 2023.

[33] Introduction to Intel® Memory Protection Extensions. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html, 2013. Accessed March 24, 2020.

[34] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Tightly seal your sensitive pointers with pactight. *arXiv preprint arXiv:2203.15121*, 2022.

[35] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882, 2018.

[36] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, and Zhenkai Liang. "the web/local" boundary is fuzzy: A security study of chrome's process-based sandboxing. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 791–804, 2016.

[37] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[38] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477. IEEE, 2018.

[39] Volodymyr Kuznetsov, László Szekeres, and Mathias Payer. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 147–163, Broomfield, CO, USA, 2014.

[40] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. Pac it up: Towards pointer integrity using arm pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[41] Kangjie Lu. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1610–1624. IEEE Computer Society, 2022.

[42] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[43] Micah Morton, Jan Werner, Panagiotis Kintis, Kevin Snow, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Security risks in asynchronous web servers: When performance optimizations amplify the impact of data-oriented attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 167–182. IEEE, 2018.

[44] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[45] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. In *Proceedings of the 2010 international symposium on Memory management*, pages 31–40, 2010.

[46] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[47] NIST. Software Assurance Reference Dataset. https://samate.nist.gov/SARD/test-suites/112. Accessed June 01, 2023.

[48] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. *SIGMETRICS Perform. Eval. Rev.*, 46(1):111–112, June 2018.

[49] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. DynPTA: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1919–1937, May 2021.

[50] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. Mitigating data leakage by protecting memory-resident sensitive data. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 598–611, 2019.

[51] J. Pewny, P. Koppe, and T. Holz. STEROIDS for DOPed Applications: A Compiler for Automated Data-Oriented Programming. In *IEEE European Symposium on Security and Privacy (Euro S&P)*, pages 111–126, 2019.

[52] Qualcomm Technologies Inc. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf. Accessed June 01, 2023.

[53] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. CoDaRR: Continuous data space randomization against data-only attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 494–505, 2020.

[54] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z. Snow, and Michalis Polychronakis. Revisiting browser security in the modern era: New data-only attacks and defenses. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 366–381. IEEE, 2017.

[55] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[56] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[57] Edward J Schwartz, Cory F Cohen, Jeffrey S Gennari, and Stephanie M Schwartz. A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1801, 2020.

[58] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 54–65, 2014.

[59] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[60] Shellphish. Educational Heap Exploitation: how2heap. https://github.com/shellphish/how2heap. Accessed June 01, 2023.

[61] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[62] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[63] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[64] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[65] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[66] PaX Team. PaX address space layout randomization (ASLR). 2003.

[67] Jidong Xiao, Hai Huang, and Haining Wang. Kernel data attack is a realistic security threat. In *International Conference on Security and Privacy in Communication Systems*, pages 135–154. Springer, 2015.

[68] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*, pages 337–352, 2013.

A Appendix

A.1 Algorithm

Algorithm 1 performs the tainting process in SVFG, involving taint source identification and propagation initiation (lines 3–8), tainted data propagation (lines 9–12), data tainting via index variables or pointers (lines 13–17), propagation from second to the first argument (lines 18–24), and SVFG traversal using *UpdateTaintList* function (lines 25–32).

A.2 Common Ancestor Solution

To find the common ancestor, we obtain all the SVF nodes that are backwardly *reachable* from an allocation's argument node. Figure 7 shows a partial SVFG of the example program in Figure 1. Node ④ in Figure 7 is the argument of the memory allocation SVF node (*i.e.*, Node ⑤) in Figure 2. The *reachable* nodes from the allocation's argument node (*i.e.*, node ❹ in Figure 7) are nodes ❷, ❸, ❹', and ❹. We then take each `cmp` node from the search paths that we have already obtained from the ICFG (*e.g.*, 11 → 10 → 7 → 6 → 5 → 4 → 3 → 2 → 1 from Figure 8), obtain its corresponding node(s) from the SVFG, and traverse backwardly from the corresponding node(s). If the backward traversal encounters any node from the *reachable* nodes, that means the `cmp` node has a common ancestor with the argument node of an allocation function, hence this `cmp` node is related to the argument of the allocation function. In Figure 7, the common ancestor of node ❹ and node ⓯ is node ❸.

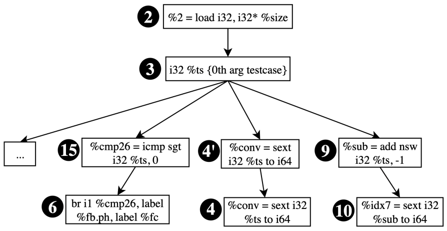
2
%2 = load i32, i32* %size
3
i32 %ts {0th arg testcase}
...
15
%cmp26 = icmp sgt i32 %ts, 0
4'
%conv = sext i32 %ts to i64
9
%sub = add nsw i32 %ts, -1
6
br i1 %cmp26, label %fb.ph, label %fc
4
%conv = sext i32 %ts to i64
10
%idx7 = sext i32 %sub to i64

Figure 7: Partial SVFG. Common ancestor of node 4 and node 15 is node 3.

Algorithm 1: Identifying taint sources starting from all call sites and propagating taints through the SVFG

```
Function PerformTaintAnalysis(SVFG):
1       InputFuncs ← a set of all input reading functions
2       TaintedNodes ← {}
        /* tainting arguments of input reading functions */
3       foreach Callee callee : getCallSites() do
4           if InputFuncs.contain(callee) then
5               foreach Argument arg : callee.getArguments() do
6                   UpdateTaintList(arg, TaintedNodes)
                    /* taints points-to set if arg is a pointer */
7                   foreach Object target : points-to-set(arg) do
8                       UpdateTaintList(target, TaintedNodes)

        /* propagating the taints starting from a memory object
           node to all reachable nodes in a SVFG */
9       foreach AddressTakenNode node : SVFG.nodes() do
10          hasTaintedArg = node.hasAnyTaintedArg()
11          if hasTaintedArg is true then
12              UpdateTaintList(node, TaintedNodes)

        /* tainting an object when the object is accessed with a
           tainted pointer or index */
13      foreach GEP node : SVFG.nodes() do
14          isTainted = node.checkForTaintedArg(startIndex=1)
15          if isTainted is true then
16              foreach Object target : points-to-set(node.getOperand(0))
                  do
17                  UpdateTaintList(target, TaintedNodes)

        /* tainting first arg when a tainted value is copied
           from the second param */
18      foreach CallInst node : SVFG.nodes() do
19          isCopyFunc = is2ndArgCopiedTo1stArg(node)
20          if isCopyFunc is true then
21              operand2 = node.getOperand(1)
22              UpdateTaintList(operand2, TaintedNodes)
23              foreach Object target : points-to-set(node.getOperand(0))
                  do
24                  UpdateTaintList(target, TaintedNodes)

    /* propagating taints to all successor nodes from a node */
    Function UpdateTaintList(arg, TaintedNodes):
25      worklist ← {}
26      worklist.insert(arg)
27      while worklist not empty() do
28          node = worklist.pop()
29          TaintedNodes.insert(node)
30          foreach Successor successor: node.successors() do
31              if successor not visited then
32                  worklist.insert(successor)
```

A.3 Locations of Vulnerable Data Objects

Table 8 shows the locations of vulnerable data objects or their pointers in various programs including the line numbers.

A.4 Additional Results

This section provides additional results regarding impact of individual rules on performance of the SPEC CPU2017 benchmark (Figure 9), impact of different choices of points-to analyses on DPP (Table 6), and cost of different analyses (Table 7).

Table 6: Different choices of points-to analyses impacting DPP.

CVE	Andersen		Steensgaard			Flow Sensitive		
	SVF (seconds)	Detection	SVF (seconds)	Detection	Run-time reduction w.r.t. Andersen	SVF (seconds)	Detection	Run-time increased w.r.t. Andersen
CVE-2006-0539	0.05	✓	0.04	✓	20%	0.06	✓	20%
CVE-2006-2362	66.19	✓	6.49	X	90%	–	–	–
CVE-2009-1759	2.42	✓	1.87	✓	23%	8.14	✓	236%
CVE-2009-2285	3.02	✓	1.59	X	47%	4.40	✓	46%
CVE-2010-2482	3.25	✓	1.33	X	59%	4.64	✓	43%
CVE-2013-4243	4.08	✓	2.30	X	44%	6.43	✓	58%
CVE-2013-4473	402.94	✓	191.52	X	52%	–	–	–
CVE-2013-4474	402.94	✓	191.52	X	52%	–	–	–
CVE-2015-8668	4.08	✓	2.30	X	44%	6.43	✓	58%
CVE-2017-12858	0.88	✓	0.50	✓	43%	1.64	✓	86%
					47.4%			**78%**

Table 7: Cost of data flow, taint and rule analyses in seconds.

Program	**Size (kloc)**	**SVF**	**Taint**	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**	**Rule 7**
fcron-3.0.0	7.9	0.05	0	0	0	0	0	0	0.01	0
ctorrent-dnh3.3.2	11.1	2.42	0.22	0.26	1.64	0	0.36	0.12	13.63	0
tiff-4.0.1	64.3	4.08	0.5	0.35	2.7	0.01	0.12	0.08	19.98	0
poppler-0.24.2	170.2	402.94	41.5	34.64	865.15	0.08	56.98	30.02	9334.37	10214.56
libzip-1.2.0	18.6	0.88	0.05	0.06	0.29	0	0.04	0.03	1.77	0.1
ghttpd-1.4	0.6	0.04	0	0	0	0	0	0	0	0
wu-ftpd-2.6.0	19.4	0.98	0.07	0.11	0.51	0	0.16	0.03	0.37	4.16
proftpd-1.3.0	59.6	23.27	4.1	1.04	8.65	0.01	2.76	0.86	32.8	21.34
httpd-2.4.7	251	2.26	0.32	0.16	1.01	0.04	0.06	0.01	2.49	0.02
nullhttpd-0.5.0	1.8	0.07	0	0	0	0	0	0	0	0
midori (browser)	89.3	1.72	0.03	0	0.01	0.01	0	0	0.07	0.01
libgtk (browser library)	443.8	88.49	0.76	0	0.15	7.26	0.07	0	1.28	0.11

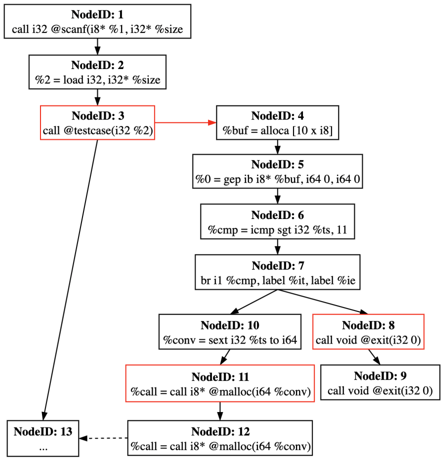
NodeID: 1
call i32 @scanf(i8* %1, i32* %size
NodeID: 2
%2 = load i32, i32* %size
NodeID: 3
call @testcase(i32 %2)
NodeID: 4
%buf = alloca [10 x i8]
NodeID: 5
%0 = gep ib i8* %buf, i64 0, i64 0
NodeID: 6
%cmp = icmp sgt i32 %ts, 11
NodeID: 7
br i1 %cmp, label %it, label %ie
NodeID: 10
%conv = sext i32 %ts to i64
NodeID: 8
call void @exit(i32 0)
NodeID: 11
%call = call i8* @malloc(i64 %conv)
NodeID: 9
call void @exit(i32 0)
NodeID: 13
...
NodeID: 12
%call = call i8* @malloc(i64 %conv)

Figure 8: ICFG for *testcase* function in Figure 1.

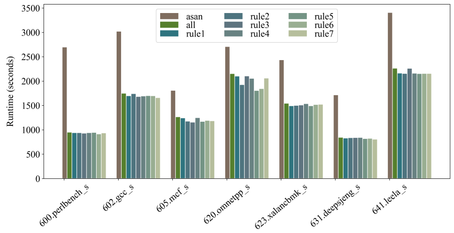

Figure 9: Impact of individual rules on performance of the SPEC CPU2017 benchmark.

Table 8: Vulnerable data objects and their pointers in various programs. The * in some source file names and function names indicate that parts of the file name or function name have been truncated for space.

Vuln. data object	**Application**	**Source**	**Line #**
char *argv[]	fcron-3.0.0	convert-fcrontab.c	152
char sym[17]	binutils-2.15	bfd/tekhex.c	394
char path[MAXPATHLEN]	ctorrent-dnh3.3.2	btfiles.cpp	454
TIFF* tif	tiff-3.8.2	tif_read.c	245
TIFF* tif	tiff-3.9.2	libtiff/tif_open.c	154
TIFF *tif	tiff-4.0.1	libtiff/tif_open.c	86
char pathName[4096]	poppler-0.24.2	utils/pdfseparate.cc	48
char *destFileName	poppler-0.24.2	utils/pdfseparate.cc	47
TIFF *tif	tiff-4.0.1	libtiff/tif_open.c	86
zip_buffer_t *buffer	libzip-1.2.0	lib/zip_buffer.c	168
char *ptr	ghttpd-1.4	protocol.c	62
struct passwd *pw	wu-ftpd-2.6.0	ftpd.c	264
char *src	proftpd-1.3.0	src/support.c	631
char *cp	proftpd-1.3.0	src/support.c	631
apr_array_header_t *log_format	httpd-2.4.7	server/log.c	1209
char *pPostData;	nullhttpd-0.5.0	src/http.c	92
int buffer[10]	CWE121*/s01	*_fgets_01.c	44
int * buffer	CWE122*/s01	*_fscanf_01.cpp	34
int buffer[10]	CWE126*/s01	*_fgets_01.c	43
int buffer[10]	CWE124*/s01	*_listen_socket_01.c	120
int buffer[10]	CWE127*/s01	*_fscanf_01.c	30