Not All Data are Created Equal:

Data and Pointer Prioritization for Scalable Protection Against Data-Oriented Attacks

Salman Ahmed[1], Hans Liljestrand[2], Hani Jamjoom[1], Matthew Hicks[3], N. Asokan[2], Danfeng (Daphne) Yao[3]

[1] IBM Research

[2] University of Waterloo

[3] Virginia Tech

IBM Research

UNIVERSITY OF WATERLOO

VIRGINIA TECH

Control-oriented attacks have become unreliable!

Control-oriented attacks

- Code injections
- Ret2libc
- ROP
- JOP
- COP
- COOP
- AOCR
- …

Defenses

- Stack canaries
- ASLR
- Code diversification
- CPI
- Control-Flow Integrity (CFI)
- …
- And many latest work (MLTA, TyPM) to make CFI sound and practical

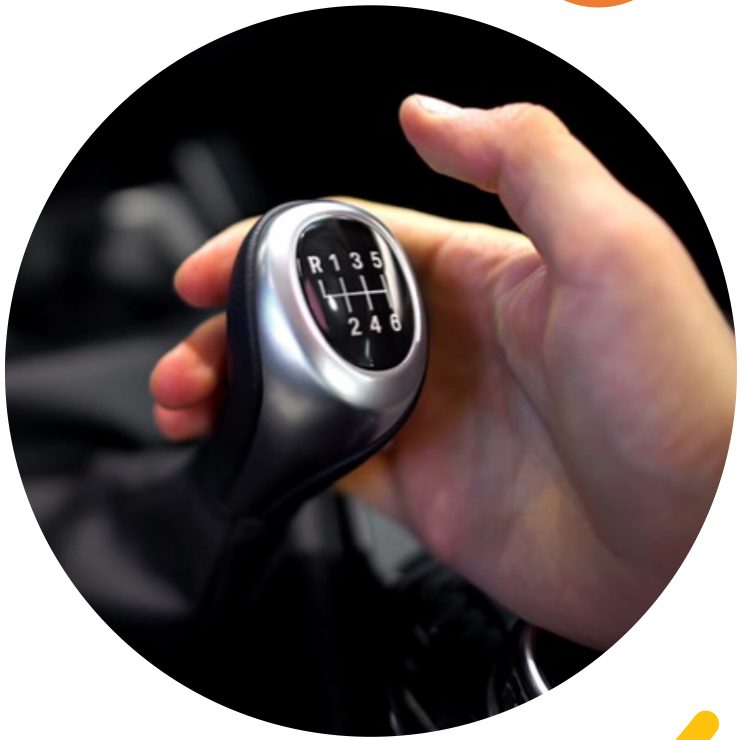
Thus, recently we have seen an uptick to data-oriented attacks (e.g., DOP, BOP, …)

Why is the shift?

- No violation of the normal flow of a program (i.e., CFI won't work)

- Expressiveness (DOP)

- Apparently, no practical defense mechanisms



Why are the existing defenses impractical?

Manipulation of data object/pointer is key attack strategy for data-oriented attacks.

It takes high overhead for data integrity!

- Data-Flow Integrity (DFI),
- Data-Space Randomization (DSR) and
- memory tagging techniques

} **42% to 116%**

- ARM Pointer Authentication } **19% - 26%**

Due to huge number of data objects/pointers, on average **~100x** compared to code pointers!

Data Pointer Prioritization (DPP)

Fact:

It takes high overhead for existing defenses to prevent data-oriented attacks through data integrity!

Observation:

Not all data objects or their pointers are vulnerable or equally sensitive.

Idea:

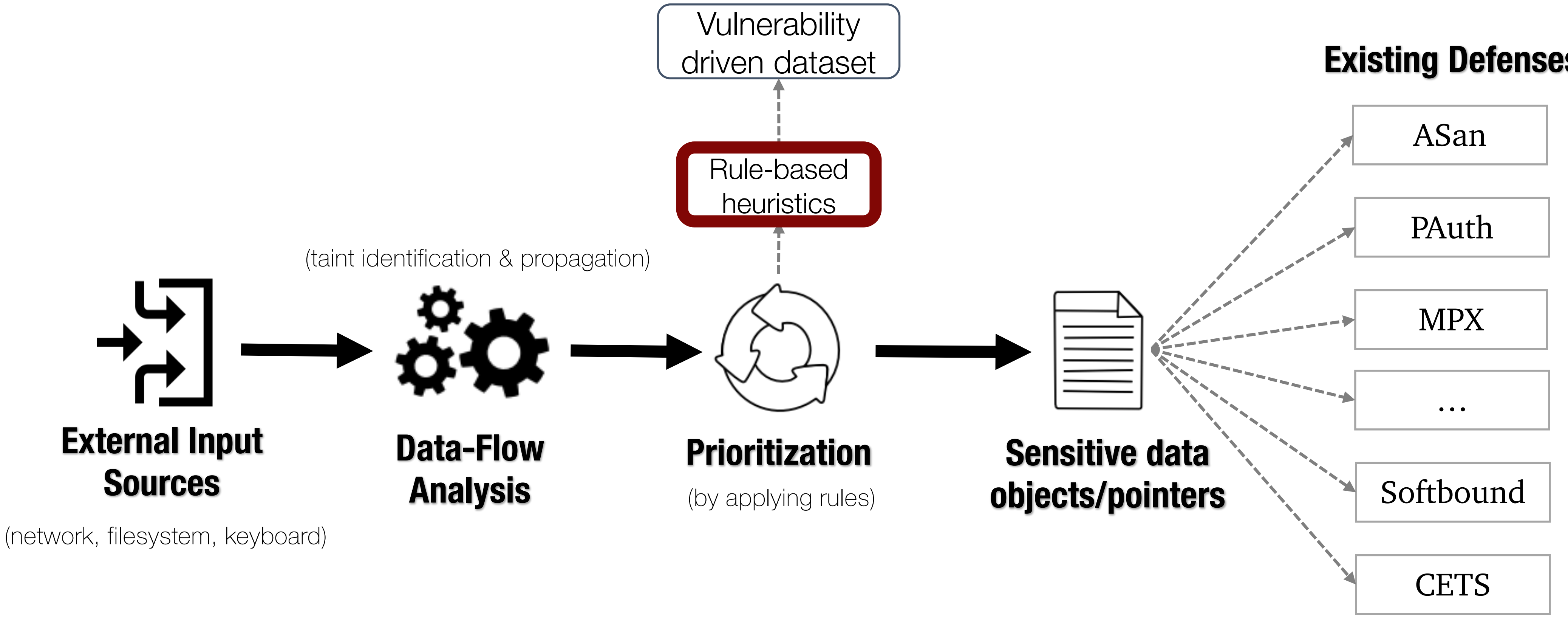
We can filter out the insensitive data/pointers and protect only the sensitive ones through prioritization.

Goal: A framework that automatically identifies and prioritizes sensitive data objects/pointers.

- generic,
- adaptable, and
- capable of being integrated with existing defenses[1].

[1] ARM pointer authentication, Intel MPX, Hardware-assisted memory tagging, etc.

DPP Framework



Challenges

- **How to obtain representative set of rules with comprehensive coverage?**
  - Breaking down advanced exploits
    - generic rules, common components increase coverage
- **How to evaluate the accuracy of the rules?**
  - Manually constructed ground truths from existing datasets
    - Juliet Test Suite, Linux Flaw Project, and data-oriented exploits

DPP Rules

We extracted 7 rules in four categories.

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

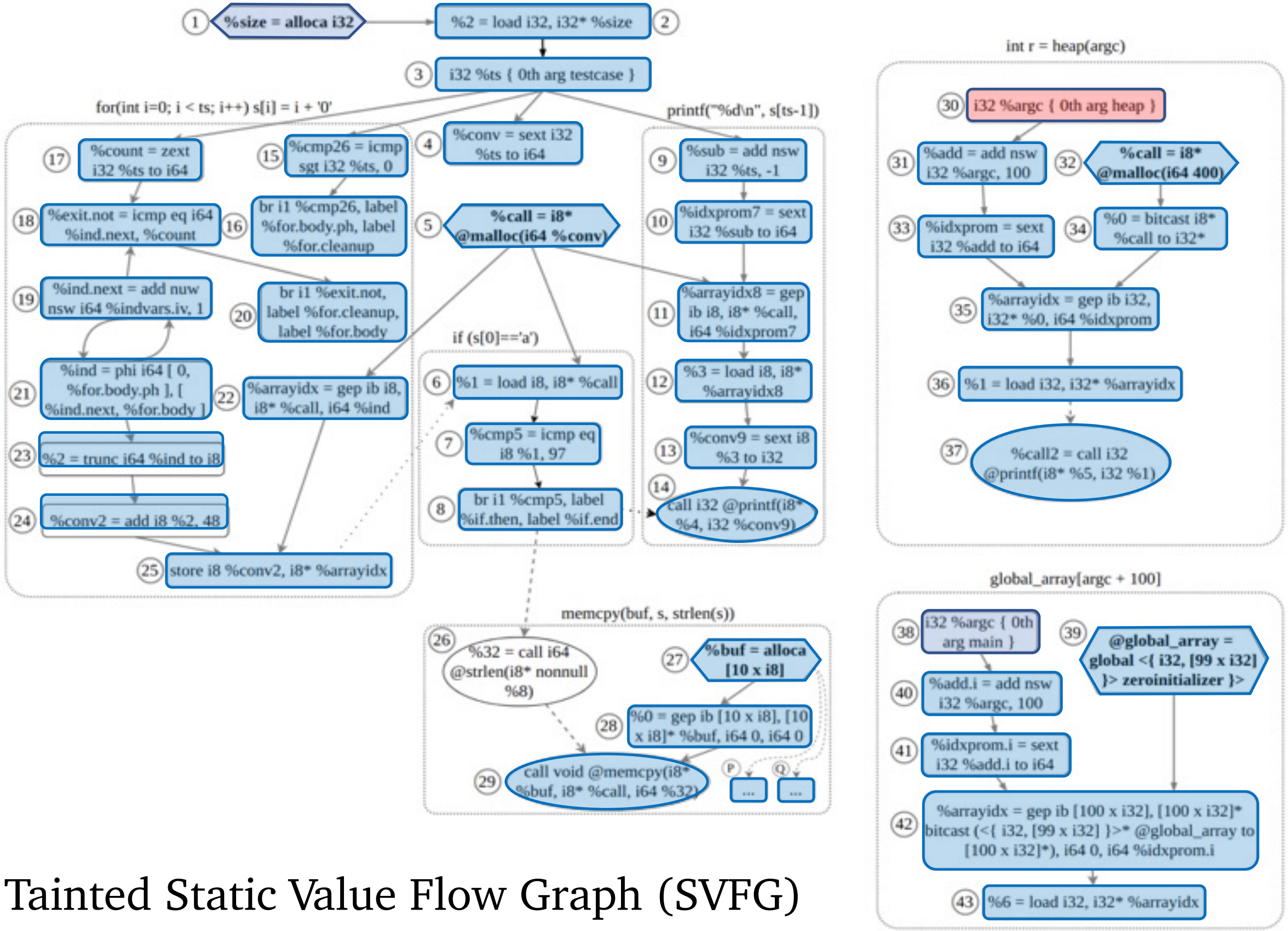
Construction of Data-Flow Graph

- We use SVF tool[1] to construct the data/value flow graph for a program.
- SVF constructs a static data/value flow graph (SVFG) on top of LLVM IR.
- We addressed three missing dependencies in SVFG.

[1]https://github.com/SVF-tools/SVF

Tainted Data Flow Graph

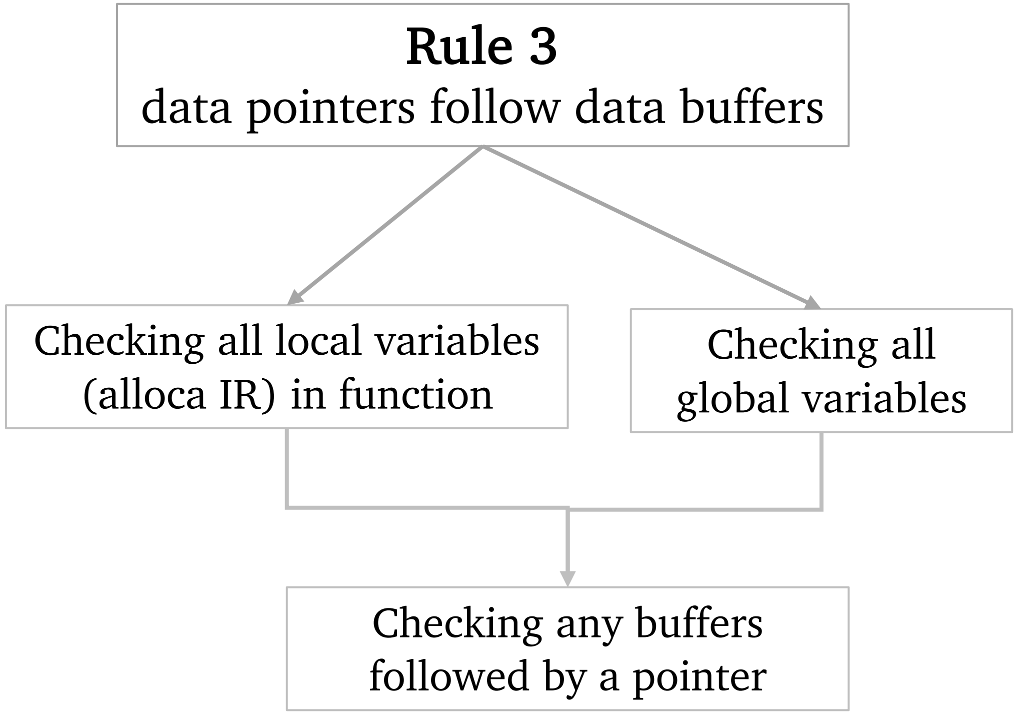
```c
#define MAX_SIZE 11
int global_array[100] = {−1};

void testcase(int ts) {
    char buf[10];
    //if (ts > MAX_SIZE) exit(0);
    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]);
    memcpy(buf, s, strlen(s)); // sink
    printf("%s\n", buf);
}
int heap (int argc) {
    int *array = (int *)malloc(sizeof(int) *100);
    int res = array[argc + 100];  // overread
    free(array);
    return res;
}
int main(int argc, char **argv){
    int size;
    scanf("%d", &size); // 'size' marked as tainted
    testcase(size);
    int r = heap(argc);
    printf("%d", r);
    return global_array[argc + 100]; //overread
}
```

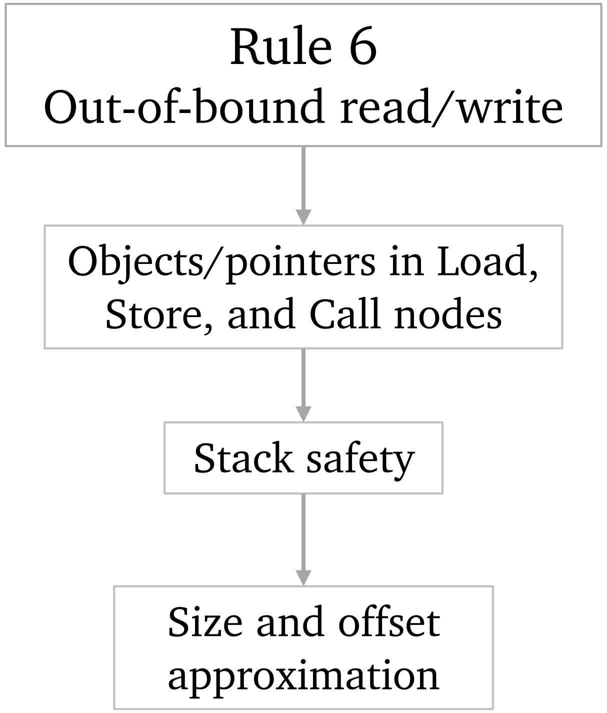


Tainted Static Value Flow Graph (SVFG)

Implementation of DPP Rules

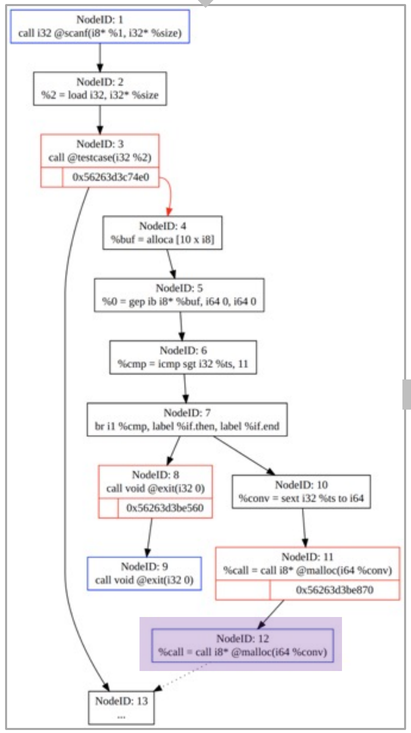
- Each rule is an LLVM analysis pass (LLVM 12)
- Rule 1, 2, 4, and 5: alias analysis, loop analysis, data layout information





Implementation of Rule 7

Dynamic memory allocation SVF nodes



Allocation node in CFG

Backward search in multiple paths for the mapped ICFG node for cmp instructions

- Path explosion → Solution: Parameterized
- Dealing with loops → Solution: removed loops
- Relevant cmp instructions → Solution: common ancestor

Details in the paper

Evaluation

1) How capable and effective is DPP for prioritizing and ranking security critical data?

2) How much performance improvement can DPP enable?

Utilized Address Sanitizer (ASan) for the evaluation

Setup

- baseline: no instrumentation
- asan: instrumented all data objects
- asan+dpp: instrumented only prioritized data objects

Security Evaluation

ASan with DPP can detect all the memory errors Linux Flaw Project and Juliet Test Suite datasets the same as the default ASan can.

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	✓	✓

Linux Flaw Project

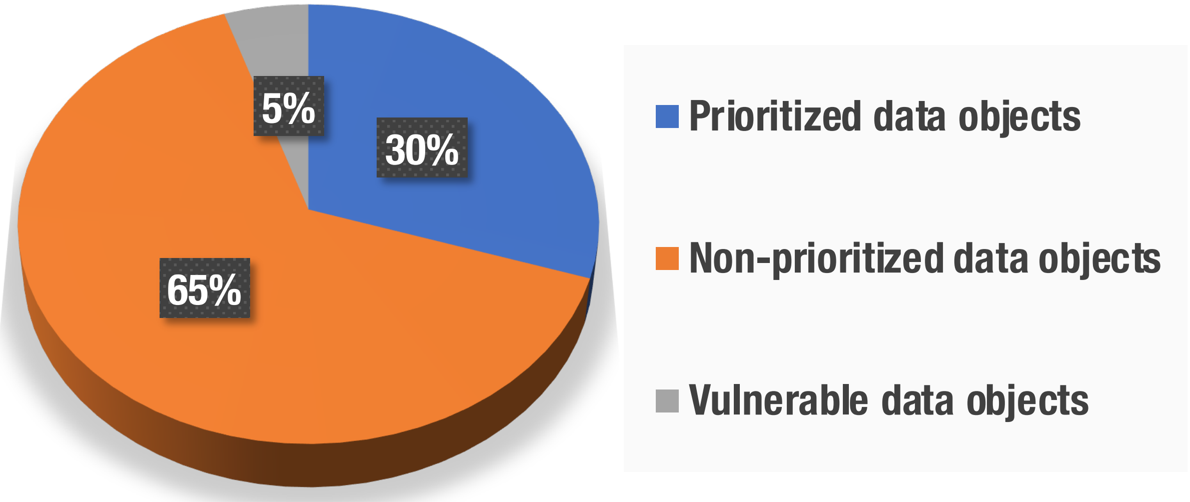
Type	Total tested cases	ASan (default)	ASan + DPP
CWE121_Stack_Based_Buffer_Overflow	144	144	144
CWE122_Heap_Based_Buffer_Overflow	144	144	144
CWE124_Buffer_Underwrite	144	144	144
CWE126_Buffer_Overread	144	144	144
CWE127_Buffer_Underread	144	144	144

Juliet Test Suite

Prioritization Efficacy

More than 95% of data objects in a real-world program do not need protection.

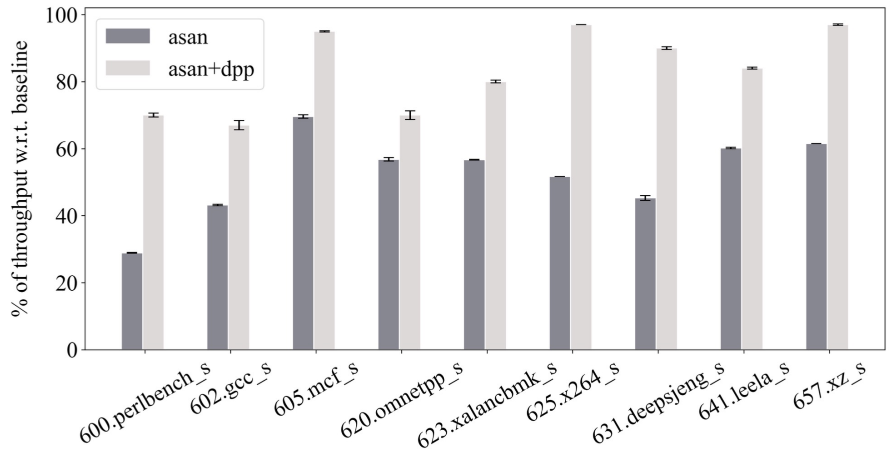
- 16 vulnerable data objects from 13 applications
- DPP identifies potentially sensitive data objects by prioritizing top 3–4% data objects from real-world applications.



Performance Evaluation

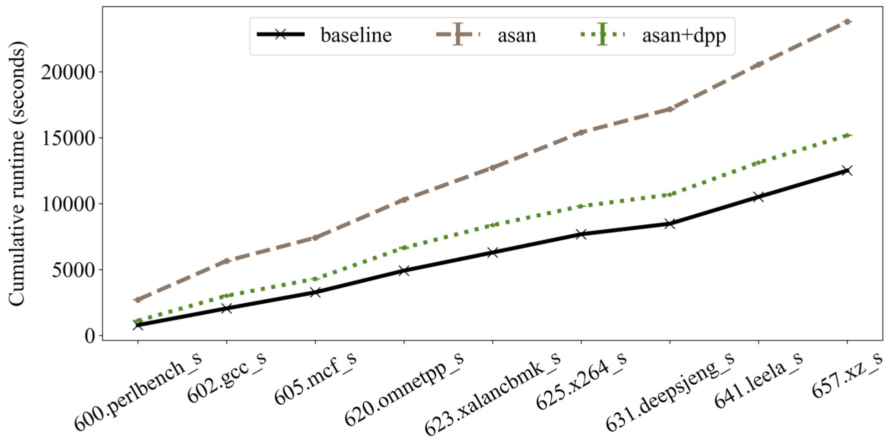
DPP improves performance by ~1.6x

- Using SPEC CPU 2017 integer benchmark



DPP reduces run-time overhead by 70% compared to ASan.

- Using SPEC CPU 2017 integer benchmark



Limitations and Discussions

- Our current prototype is NOT a live defense.
- Our approach may miss sensitive objects if we overlook sensitive variables (apart from pointers)
- A broader benchmark is needed to fully assess the effectiveness of our rules.

Conclusion

- We proposed an automatic prioritization framework for identifying and ranking sensitive memory-resident data to prevent data-oriented attacks.

- Simple rule-based heuristics are effective.

- Our proposed prioritization scheme is new and different from the conventional protection paradigm.

Not All Data are Created Equal:

Data and Pointer Prioritization for Scalable Protection Against Data-Oriented Attacks

Thank You!

Q & A

https://github.com/salmanyam/DPP

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

Salman Ahmed

sahmed@ibm.com

@salmanyam

@salmanyam