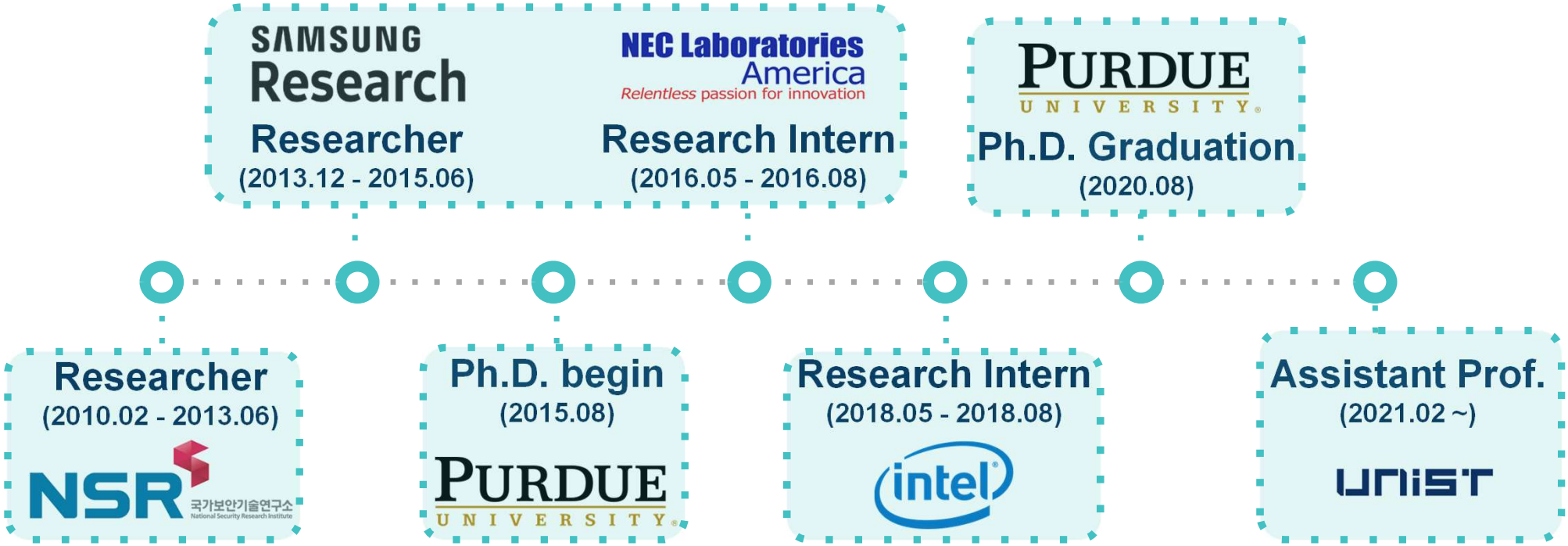
Practical Type and Memory Safety Violation Detection and Mitigation Approaches.

Secure Software Lab (S2Lab)

July 7th, 2023
Yuseok Jeon

About Me

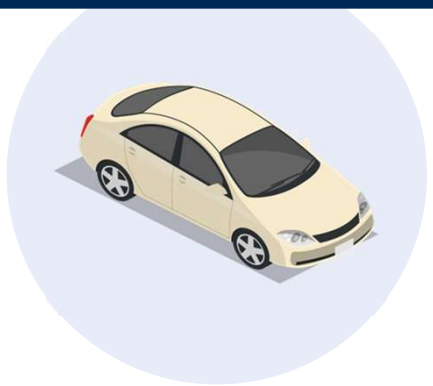
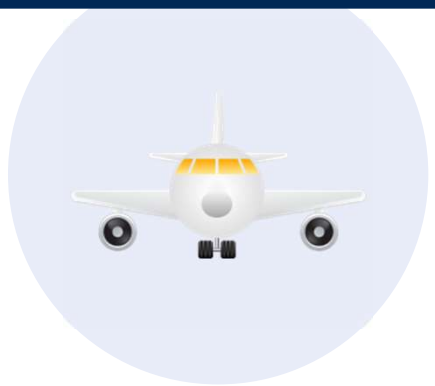
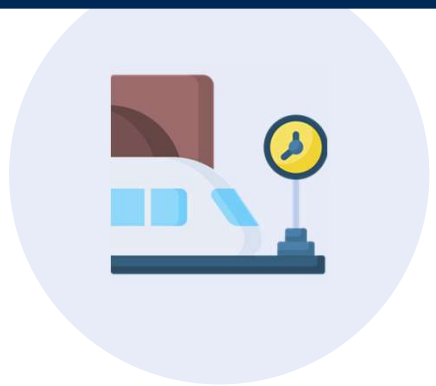


- Research Area: Software/System security

Vulnerabilities are Critical



Computers are everywhere

Vulnerabilities are Critical

Vulnerabilities + Hackers = Critical attacks

Vulnerabilities are Critical

threatpost Cloud Security / Malware / Vulnerabilities / Waterfall Security Spotlight

Forget BlueKeep: Beware the GoldBrute

Critical Flaws in Amcrest HDSeries Camera Allow Complete Takeover

Someone can spy on you !!!

HOME › NEWS › CYBER SECURITY

Virus shuts down TSMC factories, impacting chip production

$255m revenue hit predicted after WannaCry variant ran rampant across unpatched systems

August 06, 2018 By: Sebastian Moss

Taiwan Semiconductor Manufacturing Company (TSMC) was forced to shut down a number of its factories over the weekend after a virus infected computer systems and fab tools.

TSMC is the world's largest dedicated pure-play semiconductor foundry, manufacturing chips for companies including Xilinx, Nvidia, Qualcomm and AMD.

Shut down factories !!!

Why Heartbleed is the most dangerous security flaw on the web

The 'catastrophically bad' bug has left Yahoo, Imgur, and countless other services vulnerable

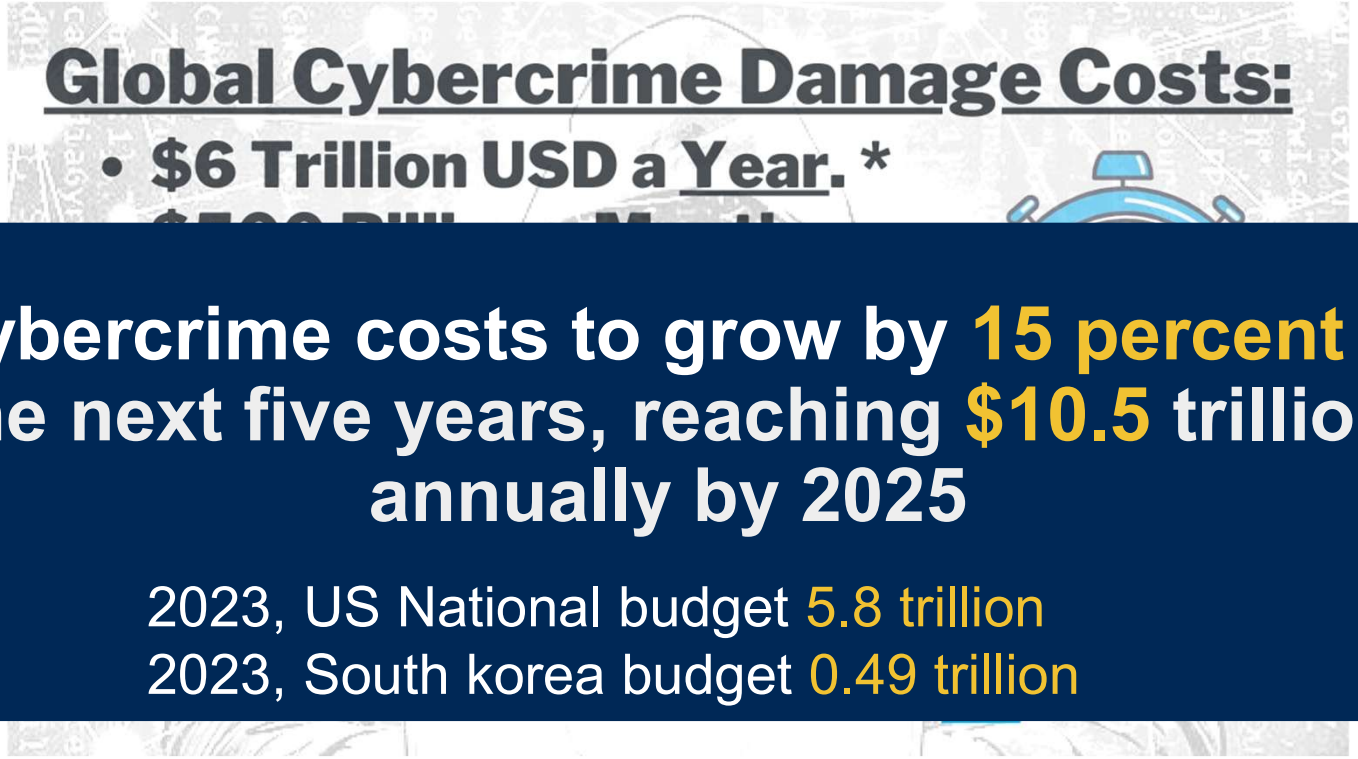
By Russell Brandom | Apr 8, 2014, 1:53pm EDT

Source Heartbleed

SHARE

Affected 500,000 servers !!!

Vulnerabilities are Critical



Global cybercrime costs to grow by 15 percent per year over the next five years, reaching $10.5 trillion USD annually by 2025

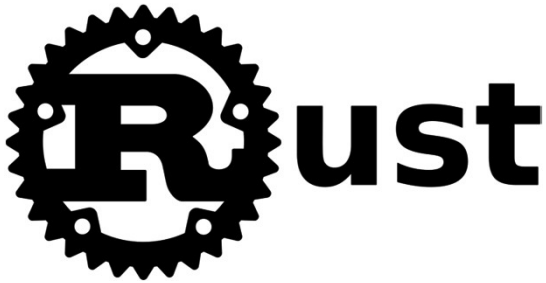
2023, US National budget 5.8 trillion
2023, South korea budget 0.49 trillion

S2Lab Research Area

❖ Focus on enforcing **software/system security**

Enforcing type/memory safety in C/C++

Rust Language Security



Autonomous Vehicles, Drones and Web Browser Security

S2Lab Research Area

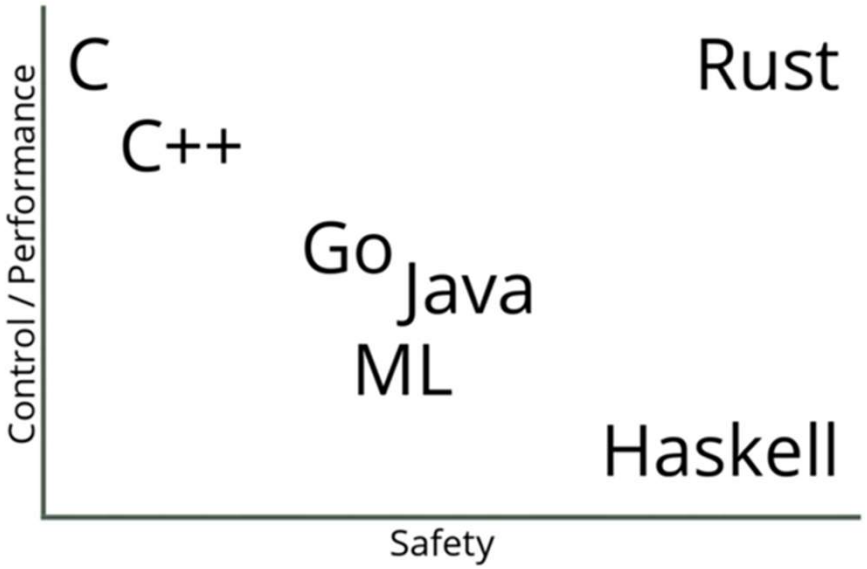
- Focus on enforcing **software/system security**

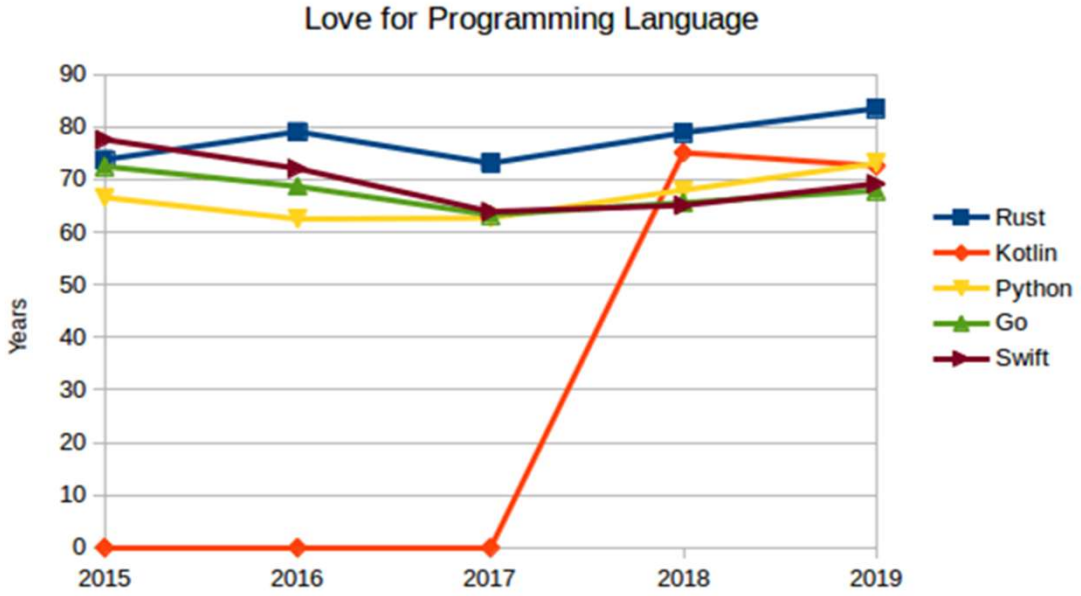
Enforcing type/memory safety in C/C++

Rust Language Security

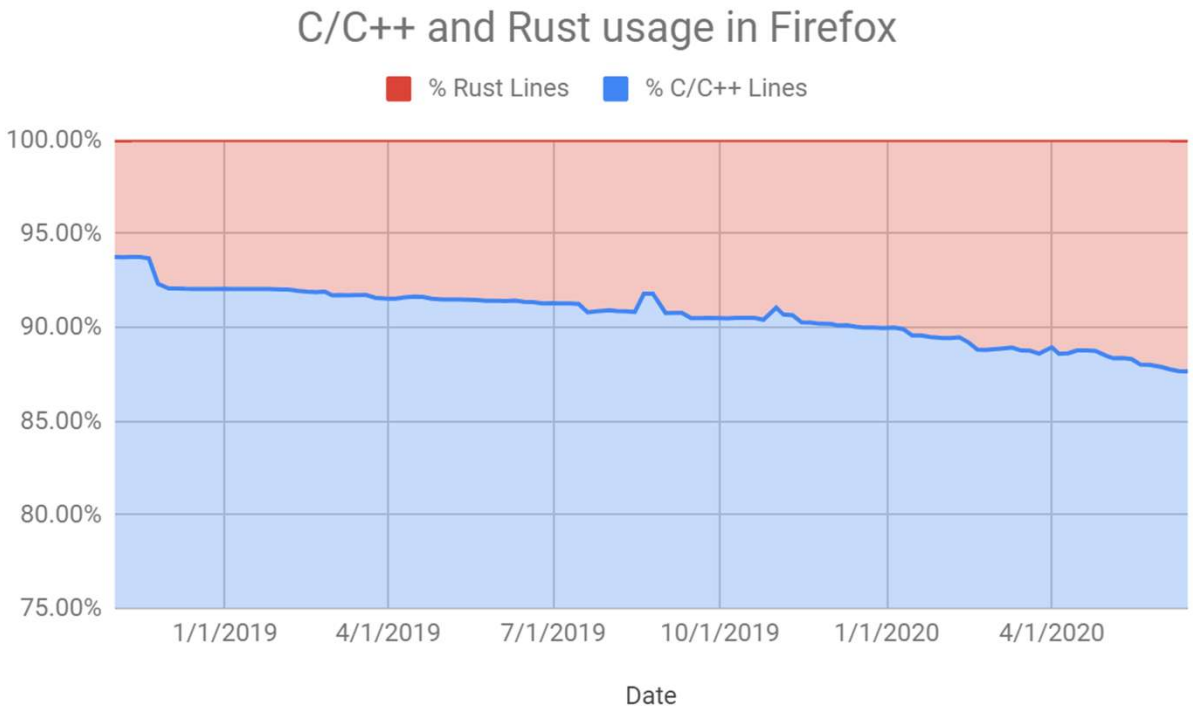
Autonomous Vehicles, Drones and Web Browser Security

Rust Language Security





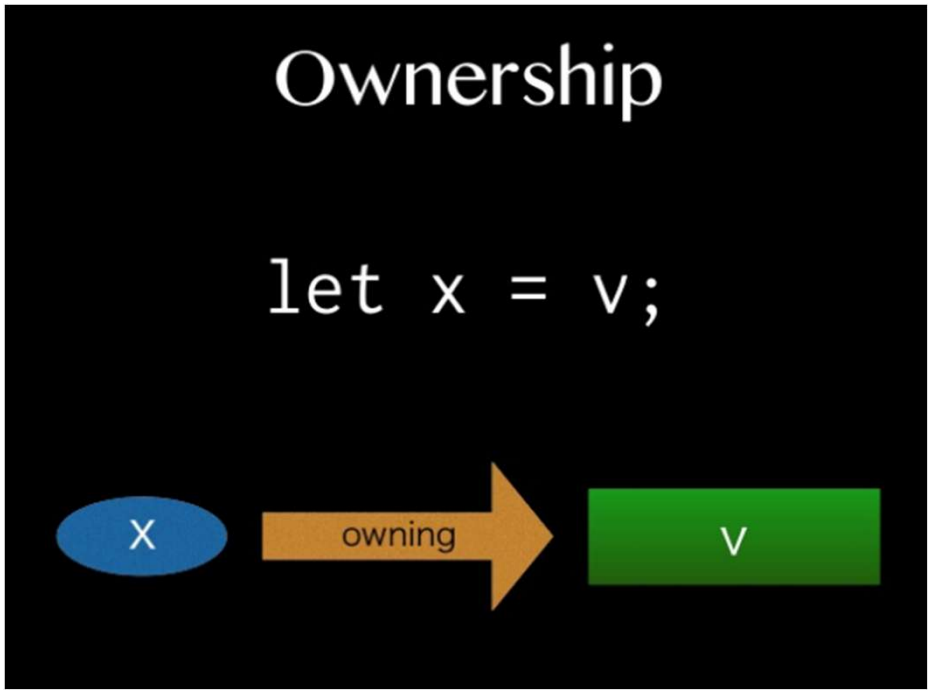
C/++ and RUST usage in Firefox



출처 : https://docs.google.com/spreadsheets/d/1flUGg6Ut4bjtyWdyH_9emD9EAN01ljTAVft2S4Dq620/edit#gid=885787479

RUST Ownership

- All allocated memory is “owned” by a unique someone.
- Ownership can transfer to another variable.



How RUST prevent memory safety violations?

- Preventing dangling pointer
  - Rust can prohibit shared mutable aliases

- Preventing buffer overflow
  - Rust generally maintains a length field for the object and performs boundary checks automatically during runtime

Need to protect Unsafe Rust

- Rust has a second language hidden inside it that doesn't enforce these memory safety guarantees
- Works just like regular Rust, but gives us extra superpowers
    - Dereference a raw pointer
    - Call an unsafe function or method
    - Access or modify a mutable static variable

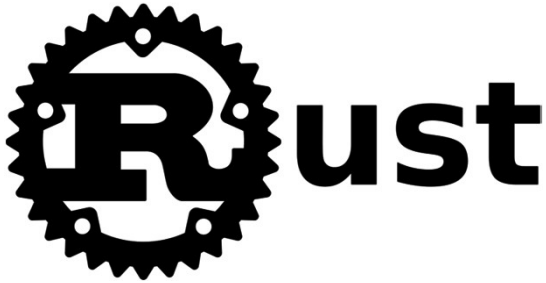
RUST is not entirely secure !

S2Lab Research Area

- Focus on enforcing **software/system security**

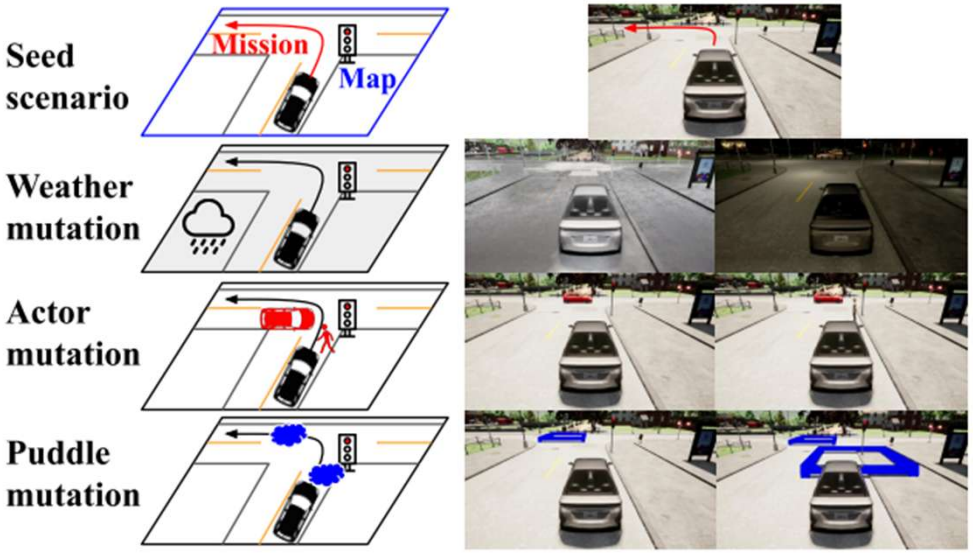
Enforcing type/memory safety in C/C++

Rust Language Security

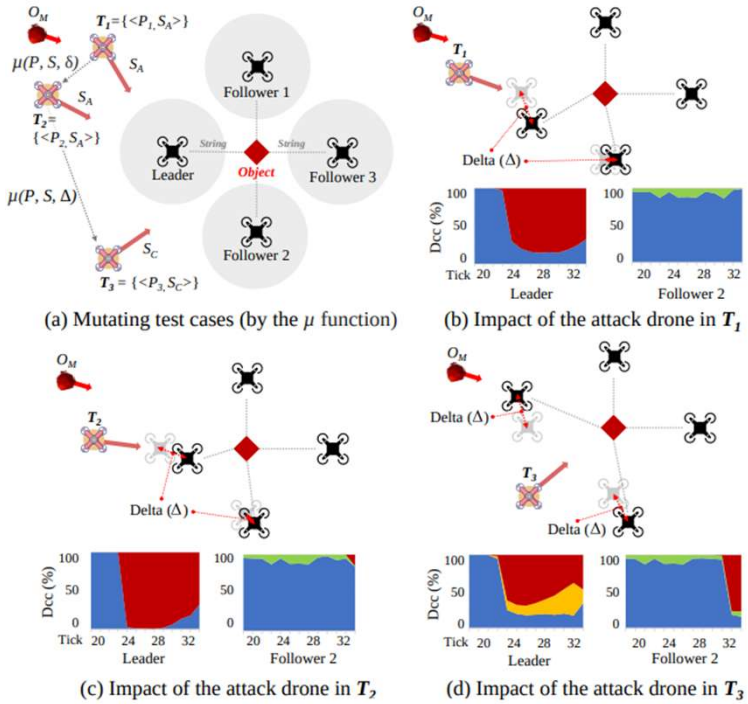


Autonomous Vehicles, Drones and Web Browser Security

Autonomous Vehicles and Drones Security



DriveFuzz
(ACM CCS 2022)



(a) Mutating test cases (by the μ function)
(b) Impact of the attack drone in T_1
(c) Impact of the attack drone in T_2
(d) Impact of the attack drone in T_3

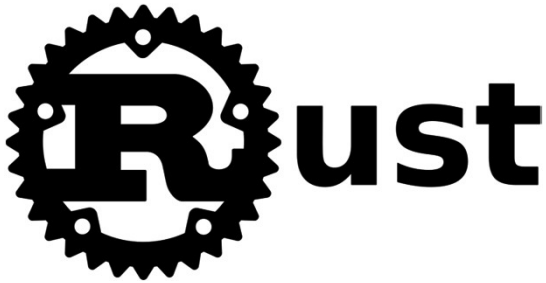
ADVERSARIALSWARM
(IEEE S&P 2022)

S2Lab Research Area

❖ Focus on enforcing **software/system security**

Enforcing type/memory safety in C/C++

Rust Language Security



Autonomous Vehicles, Drones and Web Browser Security

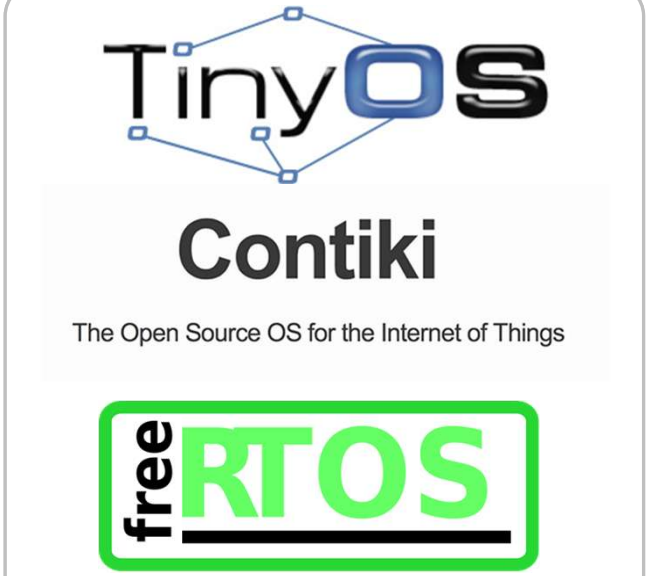
C/C++ (Unsafe languages) are Everywhere

❖ Modern computer systems are mainly **implemented in C/C++**

Operating system/ applications

AI

IoT

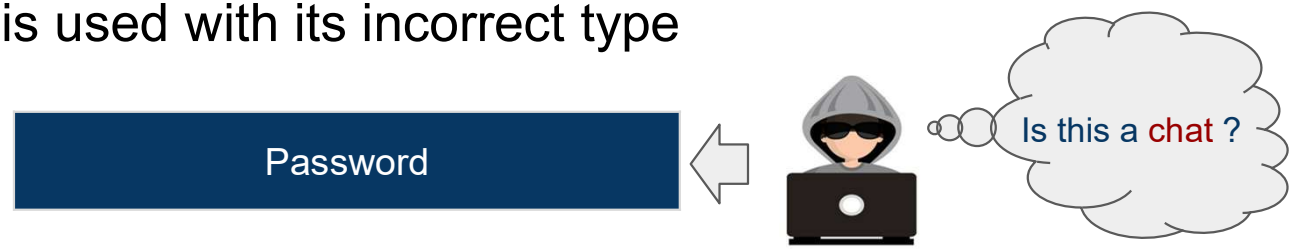


C/C++ Type and Memory Safety Violation Issues

❖ C/C++ trade type and memory safety for performance

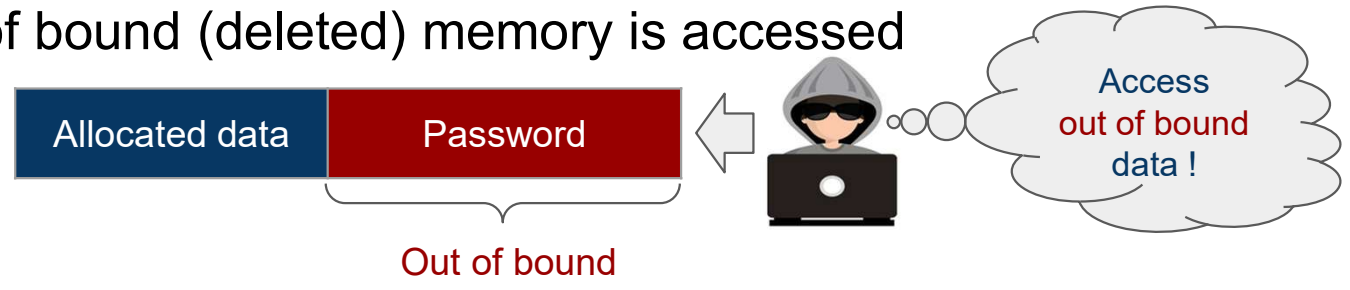
Type safety violation

❖ Data is used with its incorrect type



Memory safety violation

❖ Out of bound (deleted) memory is accessed



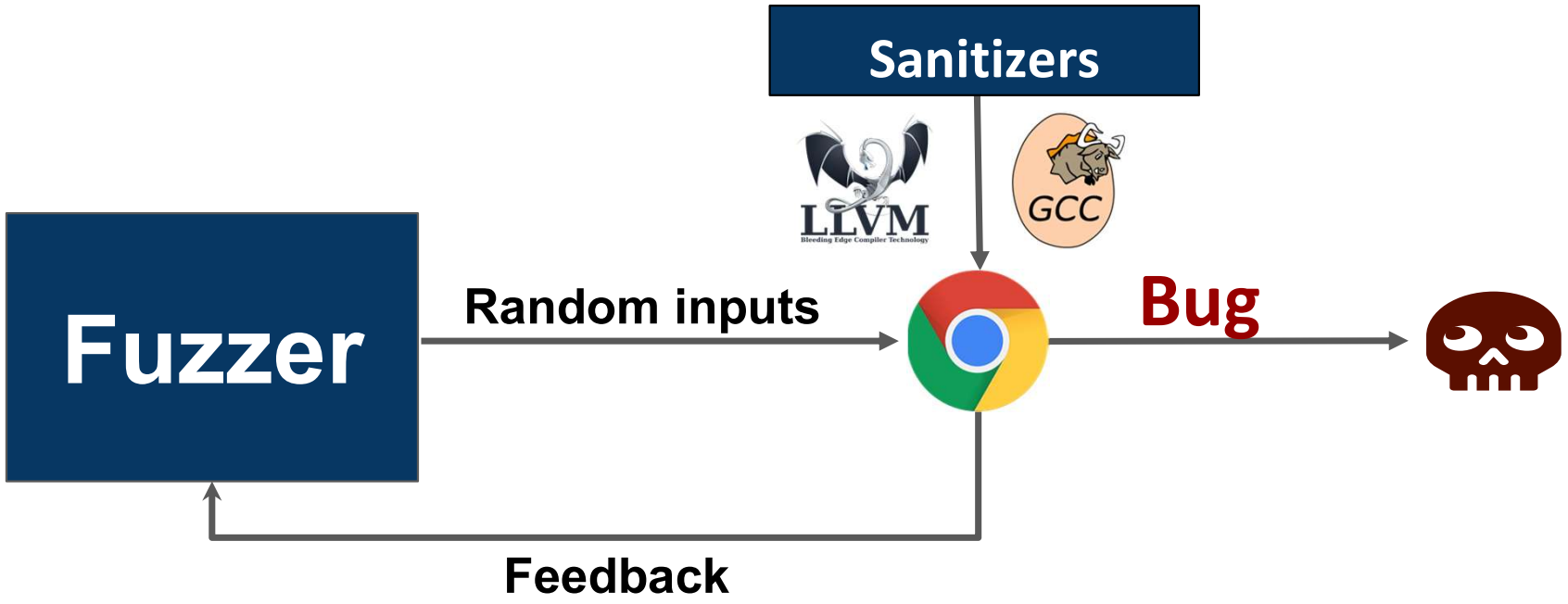
Sanitizer: Debug Policy Violations

- Observe actual execution and flag incorrect behavior
  - e.g., detect memory corruption or memory leak

- Many different sanitizers exist
  - Address Sanitizer (ASan)
  - Memory Sanitizer (MSan)
  - Thread Sanitizer (TSan)
  - Undefined Behavior Sanitizer (UBSan)

Fuzzing

- Fuzzing is an automated software testing technique
- To detect triggered bugs, fuzzers leverage sanitizers
- Popular and effective combination: Fuzzer + Sanitizer



Enforcing type/memory safety in C/C++

- Focus on developing advanced **Sanitizer**, **Fuzzer**, and **Mitigation** technices

Sanitizers

TypeSan [ACM CCS 16]
HexType [ACM CCS 17]
PsyPry [USENIX SEC 23]
Type++ [IEEE S&P 24 under review]
HwTypeSan [In preparation]

Fuzzer

FuZZan [USENIX ATC 20]
SwarmFlawFinder [IEEE S&P 22]
Autofuzzer [ACM CCS 22]
...

Mitigation

MT-Sysfilter [NDSS 24 under review]

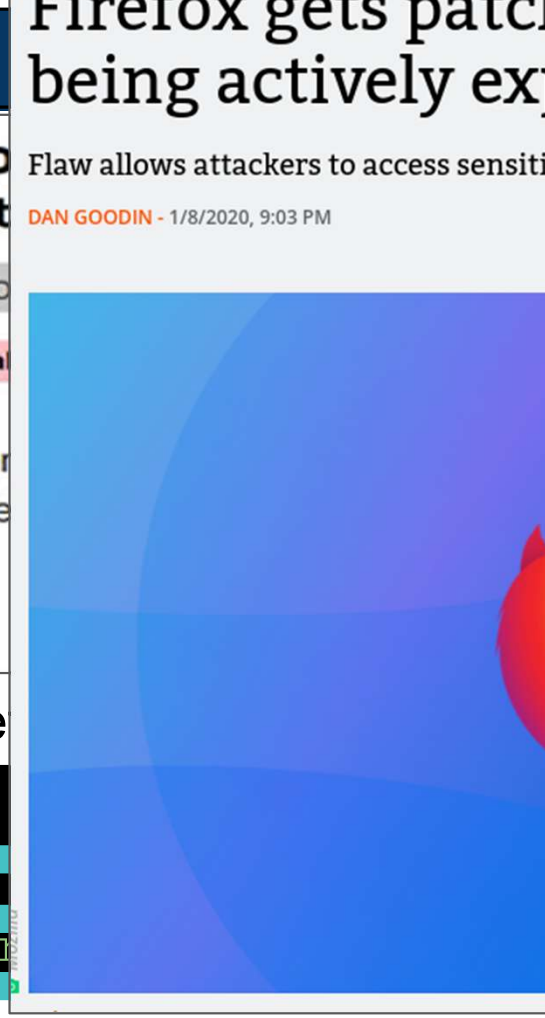
HexType

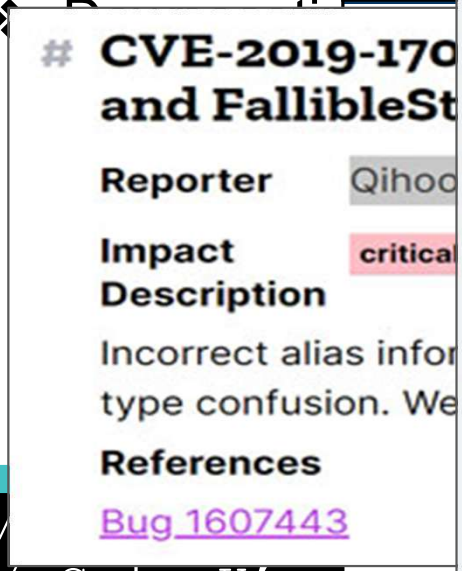
Motivation

- C++ is a popular programming language
  - Google Chrome, Firefox, and Java Virtual Machine
- Type confusion bugs are major attack vectors
- Existing type confusion detection approaches still are impractical
  - Typesan's 12% detection coverage for Firefox
  - Typesan's 78.23% overhead for SPEC CPU2006

Type Casting & Illegal Down Casting

- Upcasting:






```
// C is K's
P *pptr;
C *cptr;
...
static cast<cptr
```

P

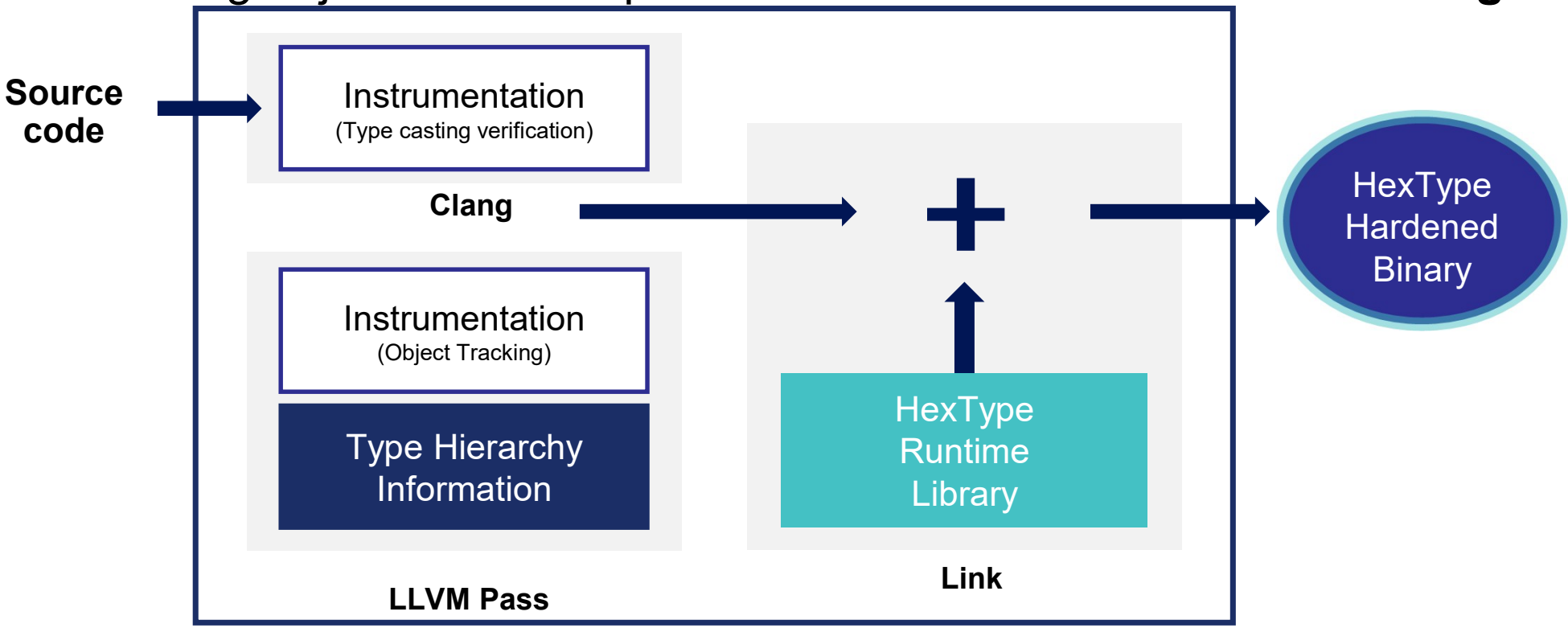
vptr

HexType

DESIGN

HexType Overview

- HexType effectively detects type confusion (illegal down casting)
  - optimization **to minimize the performance impact**
  - handling object allocation patterns **to maximize detection coverage**



Type Casting Verification

[Source code]

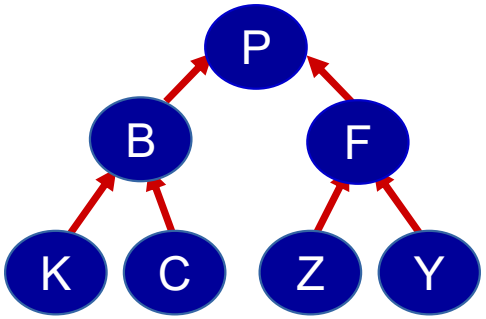
```
.....................
B *pB = new B;
Update_objTypeMap(pB,B);
C *pC = static_cast<C*>(pB);
Verify_type_casting(pB,C);
```

[Runtime library]

```
Verify_type_casting (Ptr *SrcPtr, TypeInfo Dst) {
  TypeInfo Src = getSrcType(SrcPtr);
  isTypeConfusionCast(Src, Dst);
}
```

Object to Type Mapping Table	
0x1232	Type1
0x2312	Type2
0x2333	Type3

Type relation information



HexType

Optimization

Optimization

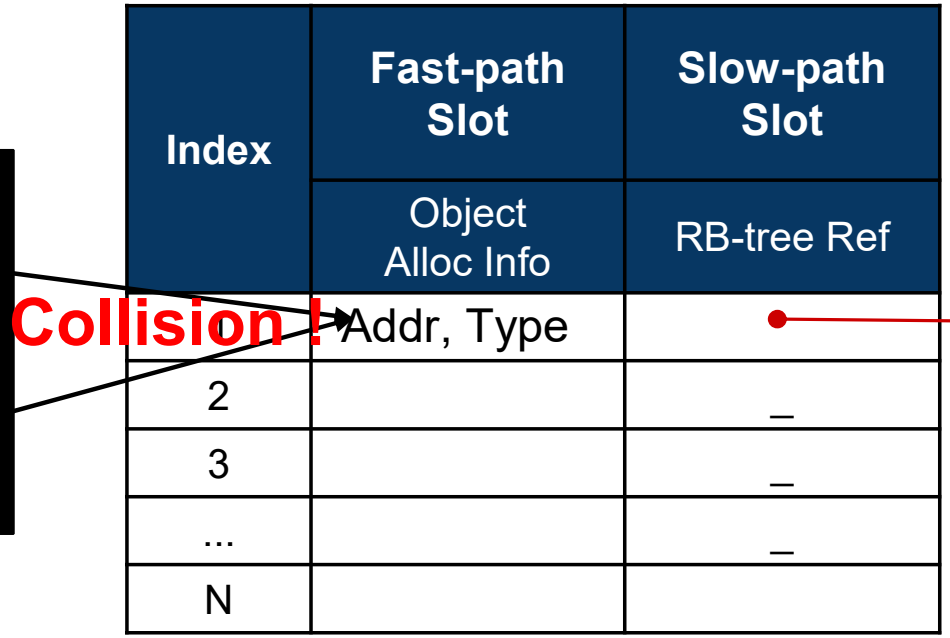
- Existing approaches have high performance overhead
  - e.g., TypeSan (by 78.23% on SPEC CPU2006)

- Propose three new optimizations
  - new disjoint metadata structure
  - selective object tracing
  - dynamic casting replace

New Metadata Structure (Hashmap + RBTree)

```
P *ptr = new P;
Index = Hash(ptr);

C *ptr2 = new C;
Index = Hash(ptr2);
```



New Metadata Structure Evaluation

Programs		Allocated objects			Fast-path Hit ratio (%) (update)	Fast-path Hit ratio (%) (lookup)
		stack	heap	global		
SPEC CPU2006	Omnetpp	1m	478m	0.001m	100.00	100.00
	Xalancbmk	3,150m	45m	0.003m	100.00	100.00
	DealII	497m	283m	0m	99.99	100.00
	Soplex	21m	639m	197m	99.69	100.00
Firefox	Octane	593m	7m	0.125m	98.82	98.65
	Drom-js	2,775m	11m	0.125m	99.65	98.43
	Drom-dom	34,900m	607m	0.125m	99.71	94.10

*** m = million**

Selective Object Tracing

- Only trace type casting related objects
- Safe objects: never used for casting
  - ➢ do not need to keep track of them
- Unsafe objects: casting related objects
  - ➢ need to keep track of them

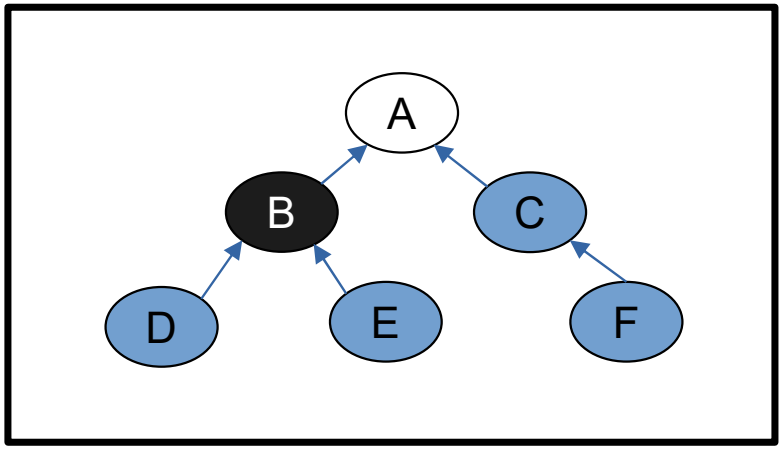
Collect Unsafe Type Set

Code

```
.....
B *pB = new B;
C *pC = new C;

static_cast<D*>(pB);
```

Type hierarchy



① **Extract unsafe types**

② **Initialize unsafe type set**

B, ...

③ **Extract all children types**

④ **Extend unsafe type set**

B, D, E, ...

Only Tracing Unsafe Objects

Programs		# of objects	Safe objects ratio (%)
SPEC CPU2006	Omnetpp	480m	54.77
	Xalancbmk	3,196m	99.42
	DealII	781m	83.81
	Soplex	858m	97.87
	Povray	6,550m	100.00
	Astar	28m	100.00
	Namd	2m	100.00
Firefox	Octane	600m	44.11
	Drom-js	2,491m	40.26
	Drom-dom	37,538m	21.54

*** m = million**

Replace Dynamic Cast

- Replaced dynamic_cast to use our fast check
- Evaluated SPEC CPU2006 C++ benchmarks
  - dealII performs a large number of dynamic_cast (206m)
  - reduced dealII's performance overhead by 4%

Performance Overhead

Programs		CaVer	TypeSan	HexType		
		%	%	%	X1	X2
SPEC CPU2006	Omnetpp	-	49.13	9.69	NA	5.07
	Xalancbmk	29.60	41.35	1.25	23.68	33.08
	DealII	-	78.23	13.13	NA	5.96
	Soplex	20.00	1.16	0.76	26.32	1.53
	Povray	-	0.36	0.34	NA	1.06
	Astar	-	0.16	-0.37	NA	1.00
	Namd	-	26.73	0.80	NA	33.41
Firefox	Octane	45.00	19.37	30.87	1.45	-1.59
	Drom-js	40.00	25.18	25.89	1.55	-1.03
	Drom-dom	55.00	97.15	126.03	-2.29	-1.30

* X1 denotes the ratio between CaVer and HexType
*X2 denotes the ratio between TypeSan and HexType

HexType

Coverage

Increasing Detection Coverage

- The state of the art still has low detection coverage
  - Firefox 10% ~ 45%
- Assume that objects are allocated individually
  - e.g., malloc or new
- Need to handle allocation using memory pools
  - e.g., placement new, reinterpret_cast

Typecasting Verification Coverage

Programs		# of casting	Type San	HexType	
			%	%	X
SPEC CPU2006	Omnetpp	2,014m	100	100	1
	Xalancbmk	283m	90	100	1.1
	DealII	3,596m	100	100	1
	Soplex	0.209m	100	100	1
Firefox	ff-octane	623m	10	70	7
	ff-drom-js	4,229m	20	80	4
	ff-drom-dom	10,786m	50	90	1.8

*X denotes the ratio between TypeSan and HexType

Newly Discovered Vulnerabilities

- Discovered 10 new type confusion vulnerabilities
  - Qt base library (4)
  - Apache Xerces-C++ (4)
  - MySql (2)

- Confirmed and patched by the developers

Conclusion

- Increased detection coverage
- Reduced overhead using several new optimizations
- Discovered 10 new type confusion bugs

 https://github.com/HexHive/HexType

FuZZan

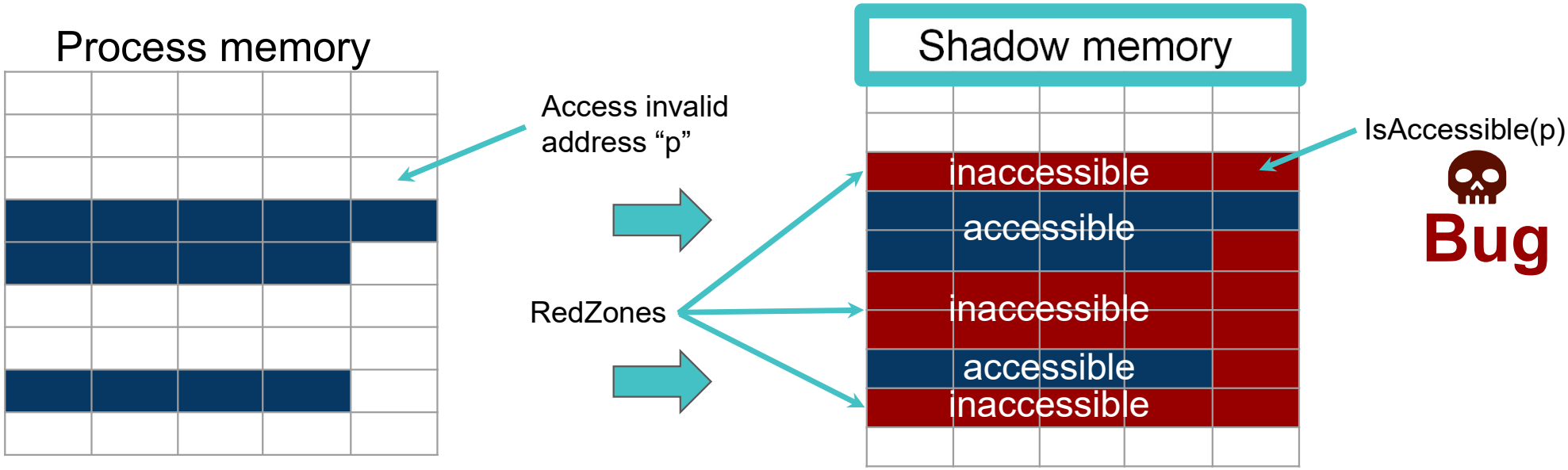
Sanitizer: Debug Policy Violations

- Observe actual execution and flag incorrect behavior
  - e.g., detect memory corruption or memory leak

- Many different sanitizers exist
  - Address Sanitizer (ASan)
  - Memory Sanitizer (MSan)
  - Thread Sanitizer (TSan)
  - Undefined Behavior Sanitizer (UBSan)

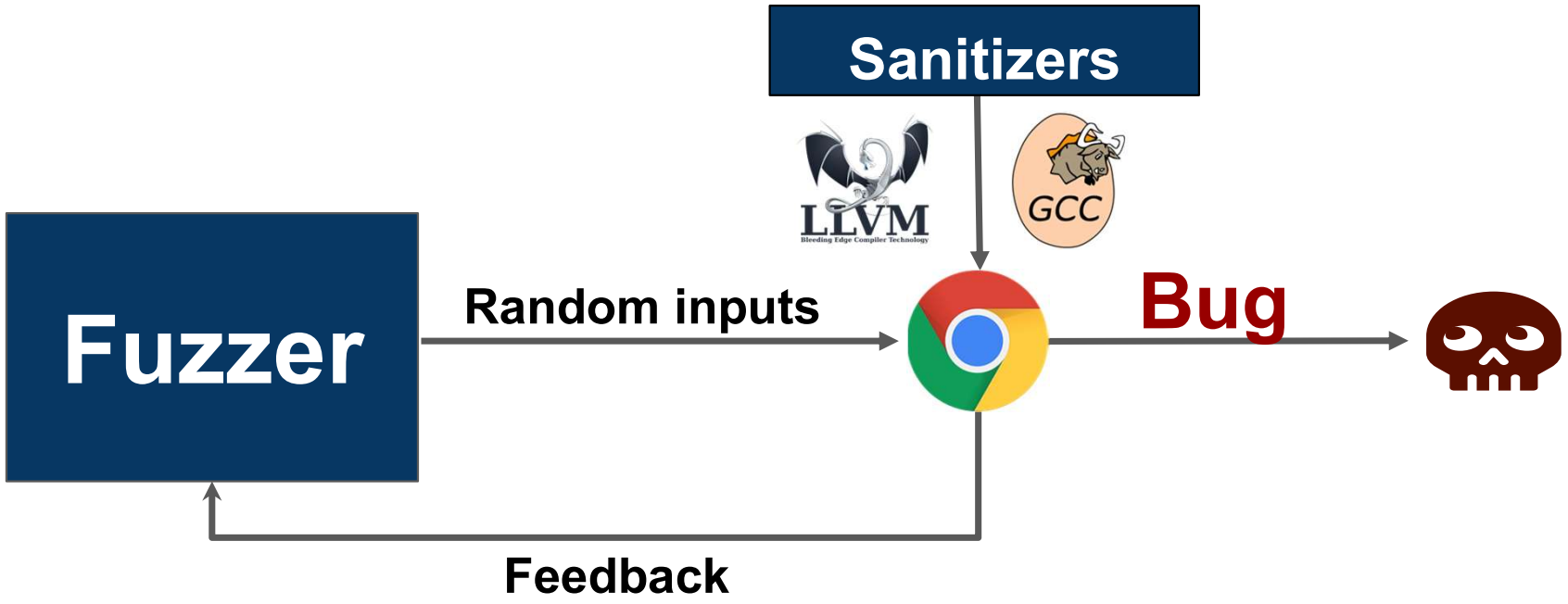
Address Sanitizer (ASan)

- Address Sanitizer is the most widely used sanitizer
  - focuses on memory safety violations
  - inserts ***redzone*** around objects
  - uses ***shadow memory*** to record whether each byte is accessible
  - detected over 10,000 memory safety violations



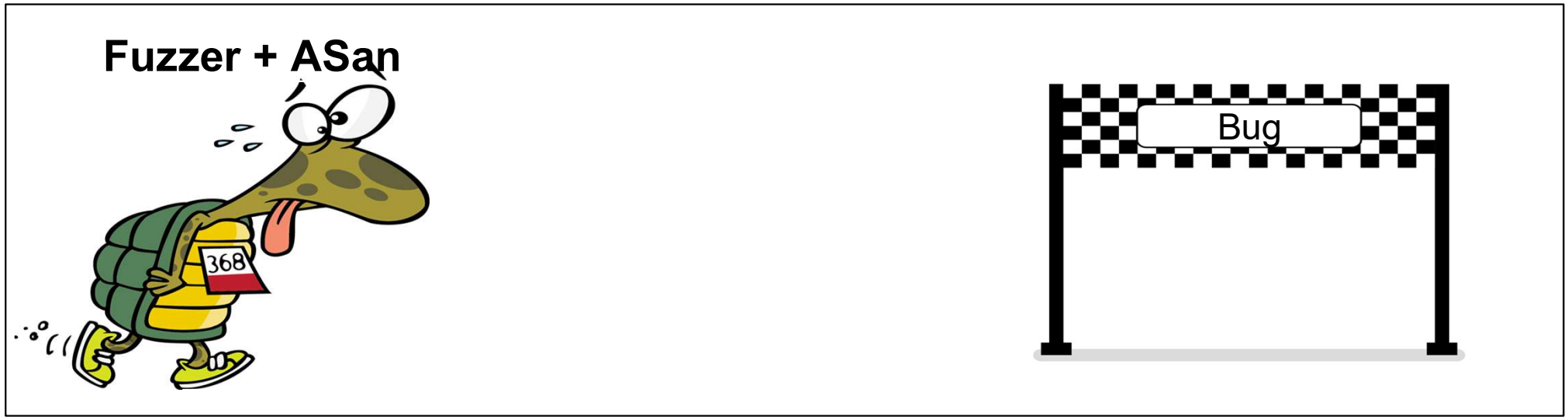
Fuzzing and Context

- Fuzzing is an automated software testing technique
- To detect triggered bugs, fuzzers leverage sanitizers
- Popular and effective combination: Fuzzer + Sanitizer



Motivation

- Sanitizers are not optimized for fuzzing environment
  - ➢ highly repetitive and short execution
- Adapting ASan increases fuzzing performance overhead
  - ➢ e.g., avg 3.4x (up to 6.59x)

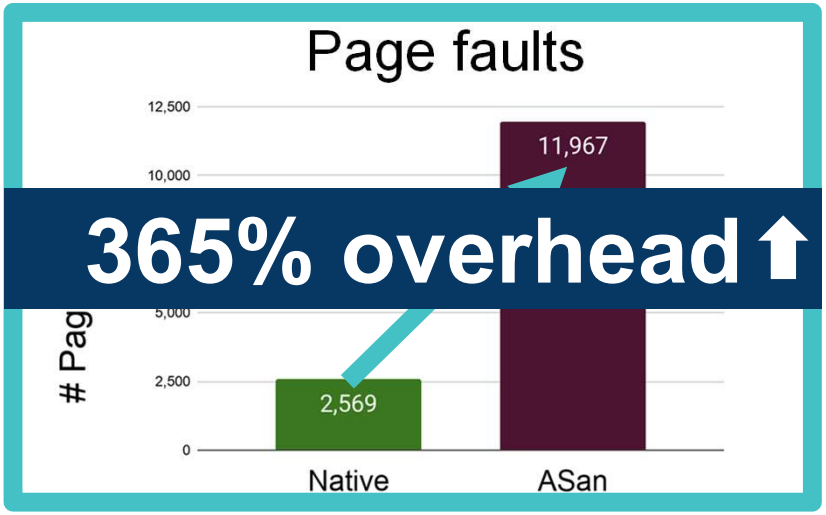


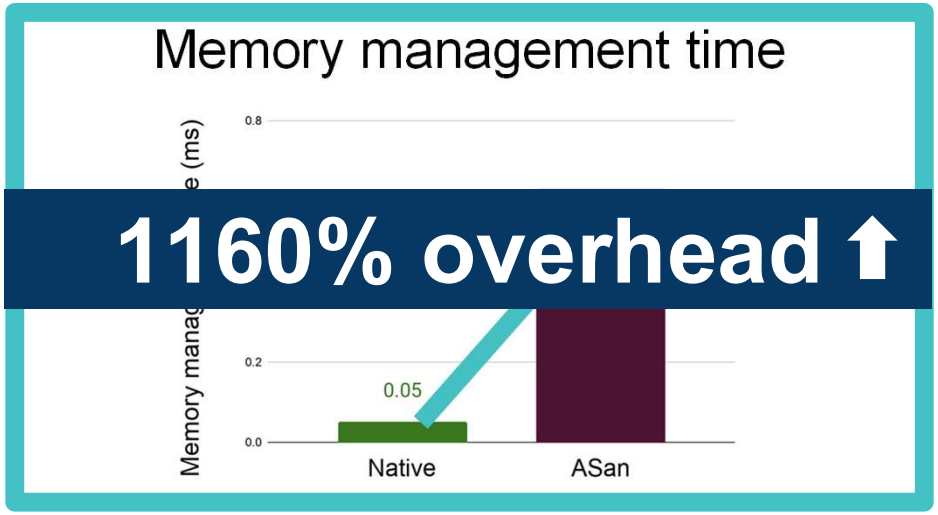
High Overhead with Sanitizers

(1) Memory management
- ➢ accessing large virtual memory area incurs overhead
- ➢ large memory area causes sparse Page Table Entries

(1) ASan initialization

(1) ASan logging





[*] Memory manage functions: (i) do_wp_page, (ii) sys_mmap, (iii) unmap_vmas, and (iv) free_pgtable

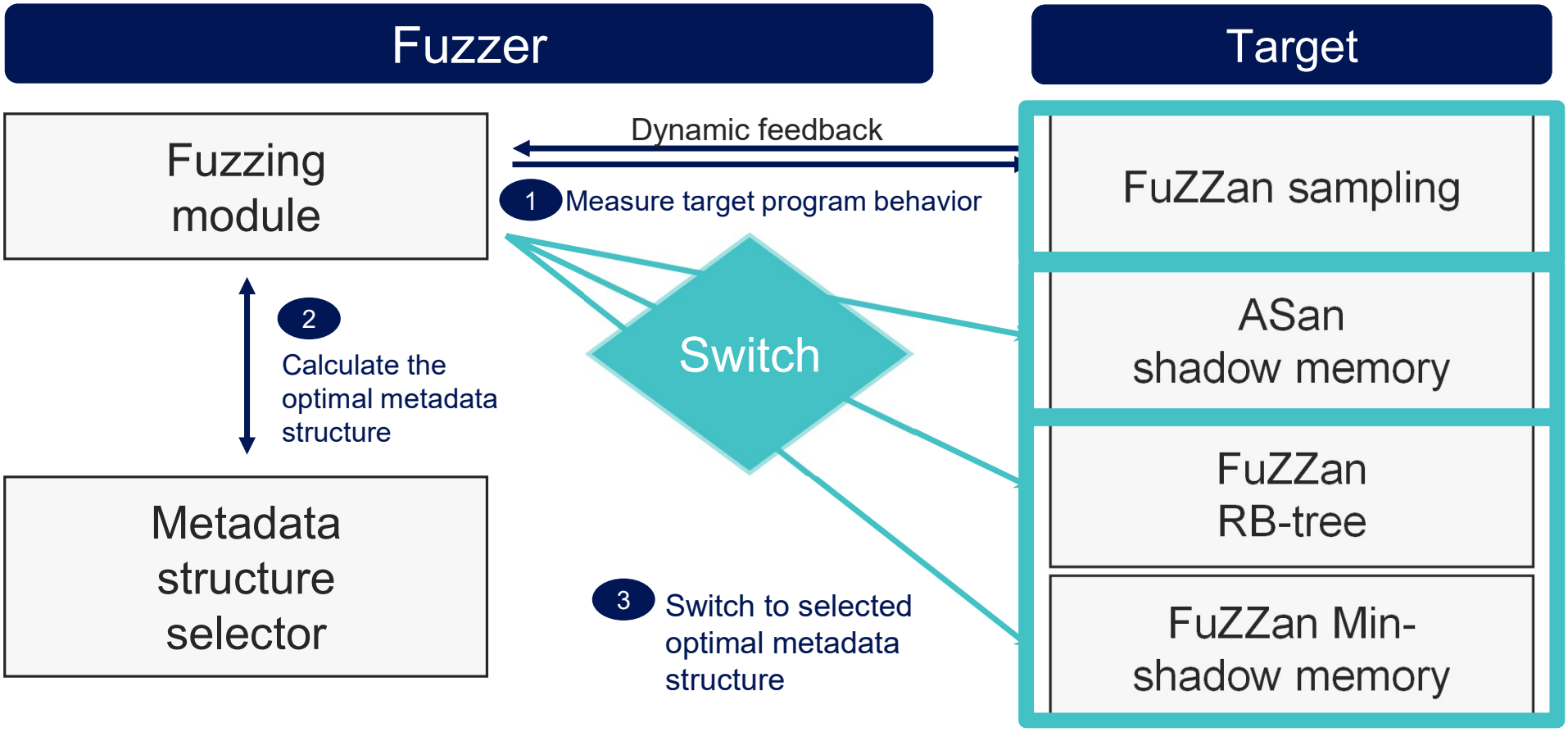
FuZZan

Design

FuZZan

- Introduce alternate light-weight metadata structures
  - avoid sparse Page Table Entries
  - minimize memory management overhead
- Runtime profiling to select optimal metadata structure
- Remove ASan logging overhead
- Remove ASan initialization overhead

FuZZan Design

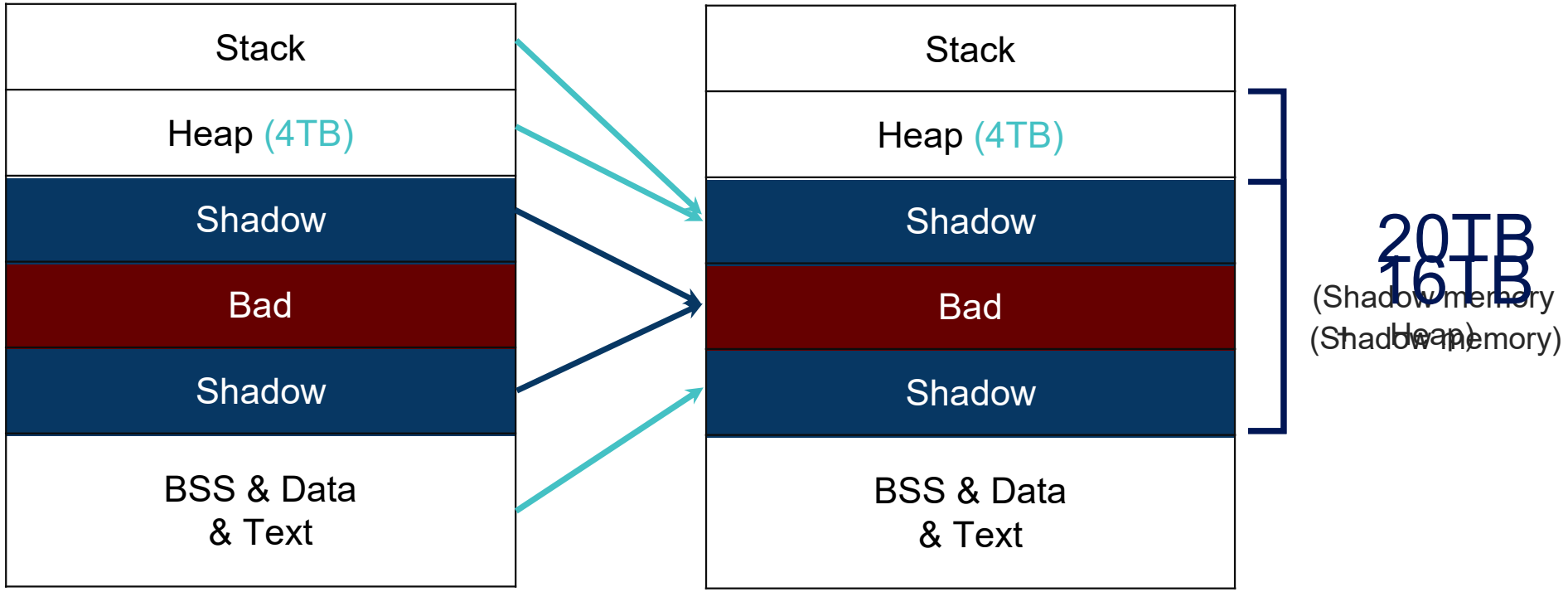


New Metadata Structures

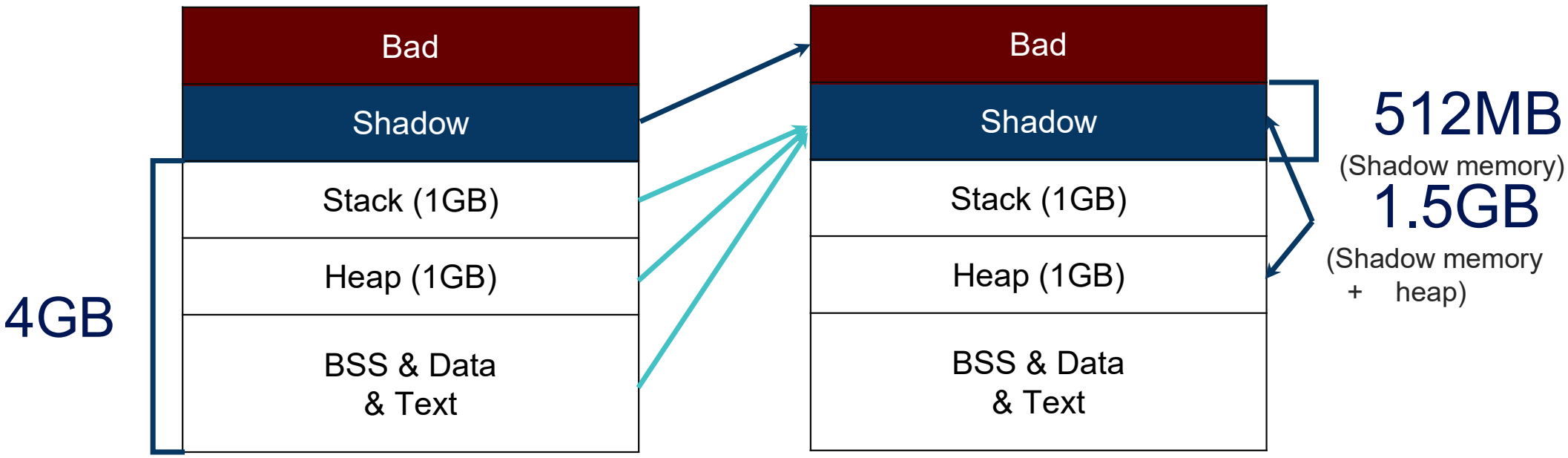
- Propose two different light-weight metadata structures

Metadata Structures		Memory Management Cost	Metadata Access Cost	Target
Address Sanitizer		High	Low O(1)	
FuZZan	RB-tree	Low	High O(log n)	Few metadata accesses
	Min-shadow	Medium	Low O(1)	Frequent metadata accesses

ASan Memory Mapping



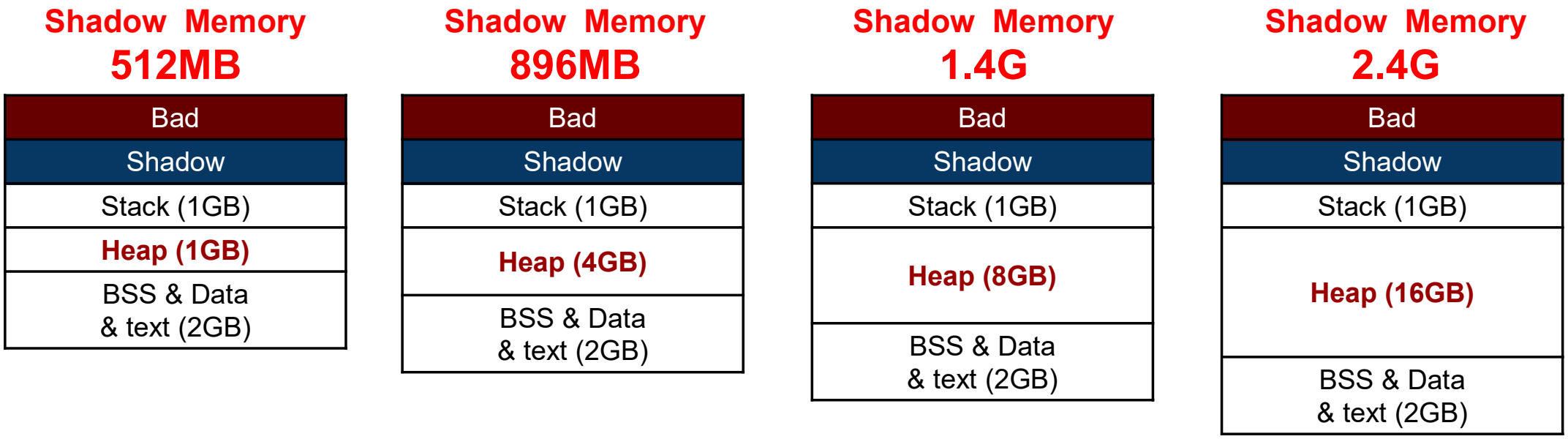
Min-shadow Memory Mapping



20TB -> 1.5GB

Other Min-shadow Memory Modes

- Create additional min-shadow memory modes
  - ➢ to accommodate large heap size
  - ➢ 1GB, 4GB, 8GB, and 16GB



Dynamic Switching Mode

❖ Switch to selected metadata structure during fuzzing

(1) Avoid user's manual extra effort to select optimal metadata structure
- ➢ no single metadata structure is optimal across all applications
- ➢ e.g., RB tree for allocating a few objects

(2) Change metadata structure according to the target's behavior
- ➢ profile at runtime and switch to selected metadata structure
- ➢ e.g., find new path

(3) Increase heap size when target exceeds limitation

Sampling Mode

- Periodically measure the target program's behavior
  - metadata access count (stack, heap, and global)
  - heap object allocation size

- Maintain ASan's error detection capabilities

Initialization/Logging Overhead

- Use the fork server to avoid unnecessary re-initialization
  - e.g., poisoning of global variable
  - move ASan's initialization point before the fork server's entry point
- Modify ASan to disable the logging functionality
  - complete logging can be recovered with full ASan

FuZZan

Evaluation

Detection Capability

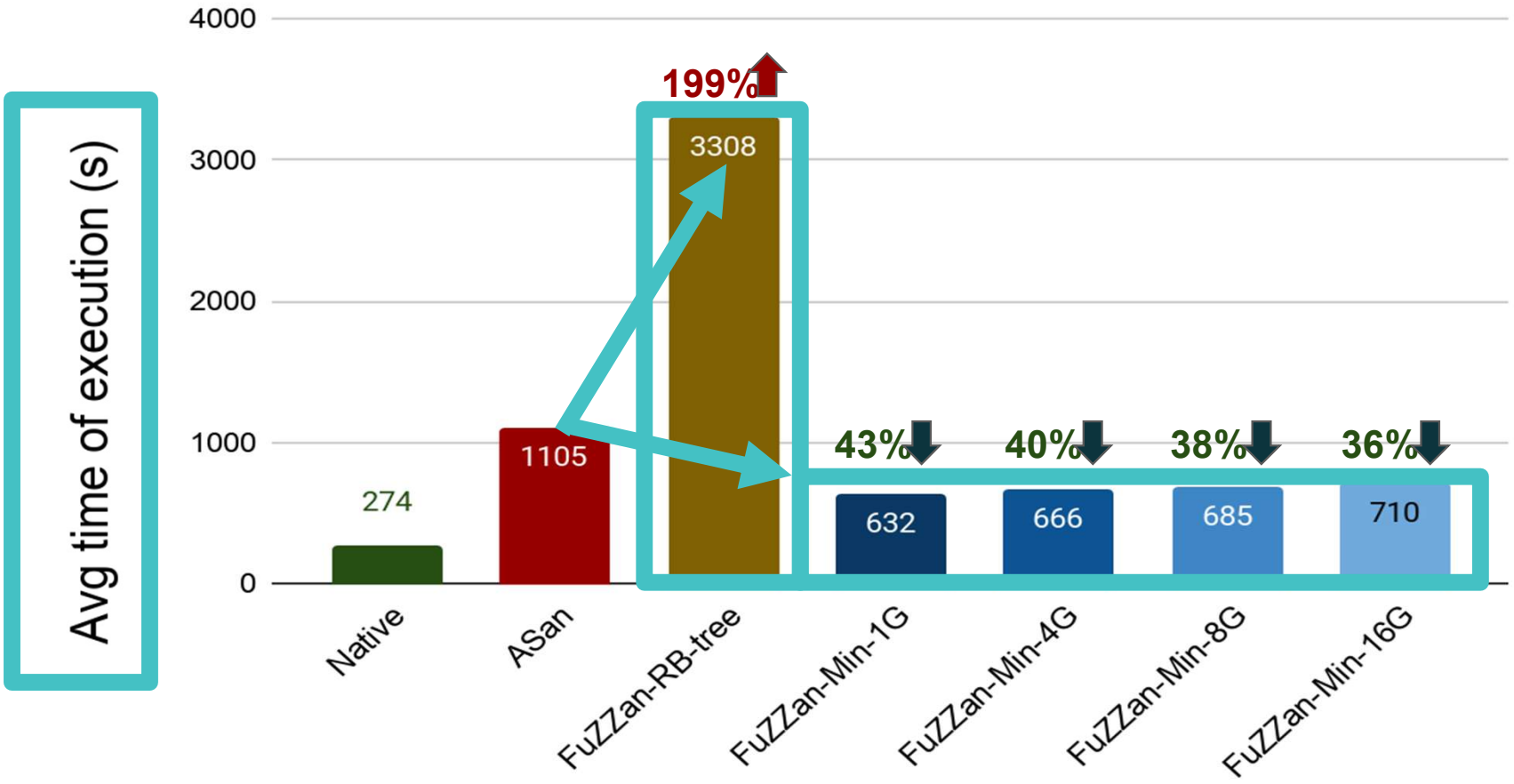
- Juliet Test Suite
  - NIST provides a test suite of all CWEs called Juliet
  - test using memory corruption CWEs
  - verified pass or fail all test cases using ASan

- Address Sanitizer provided unit test
  - verified pass all available test cases

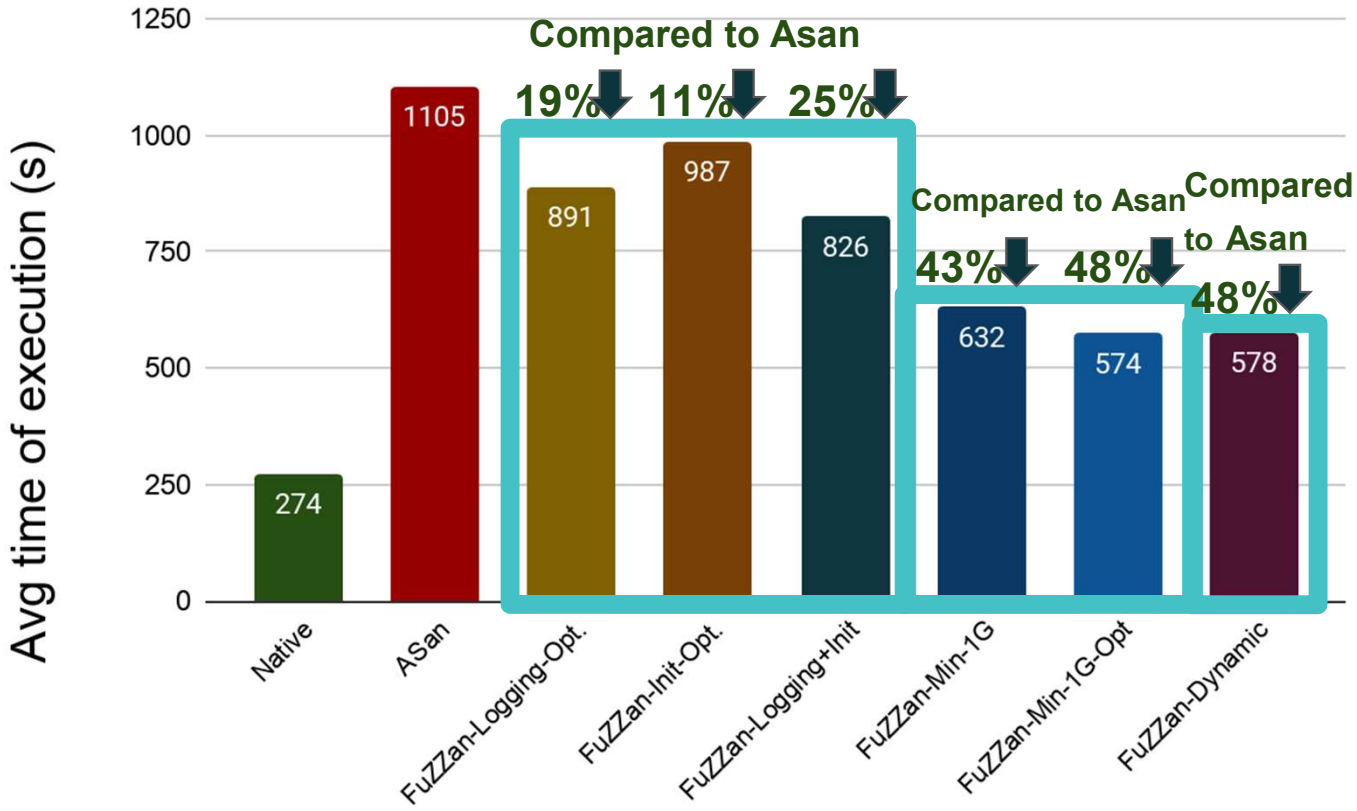
- Fuzzing test using Google Fuzzer Test Suite
  - fuzzing using 26 applications in test suite
  - verified same detection capability during fuzzing

CWE: Common Weakness Enumeration

Metadata Structure Performance

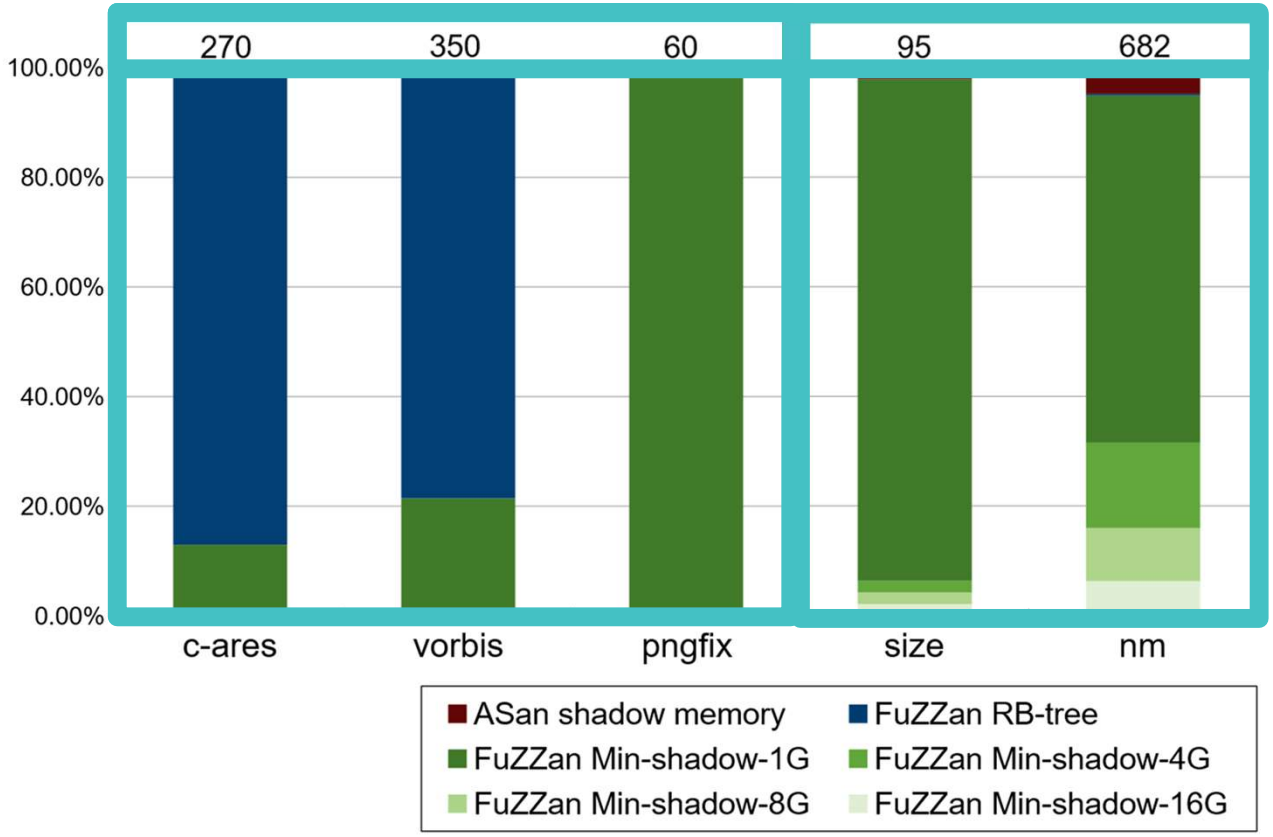


Performance Optimizations



FuZZan-Logging-Opt: optimization for logging overhead
FuZZan-Init-Opt: optimization for Initialization overhead
FuZZan-Min-1G-Opt: min-shadow memory (1G) mode with logging and initialization overhead

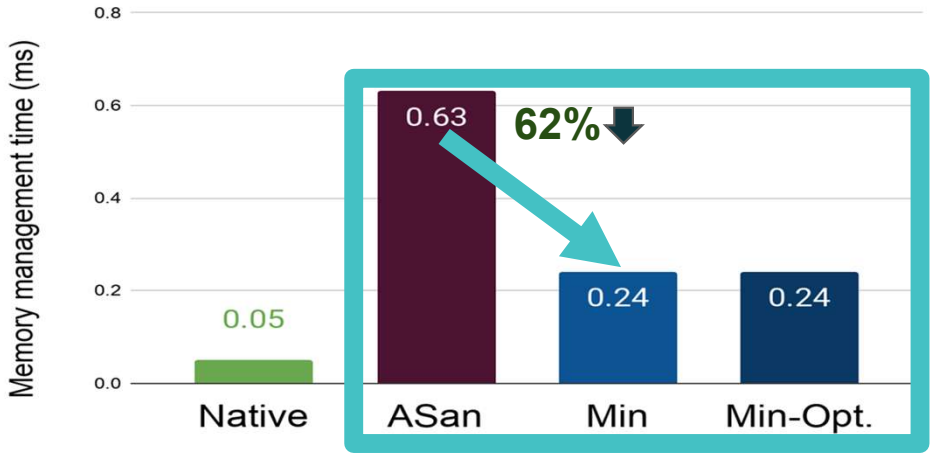
Dynamic Switching Performance



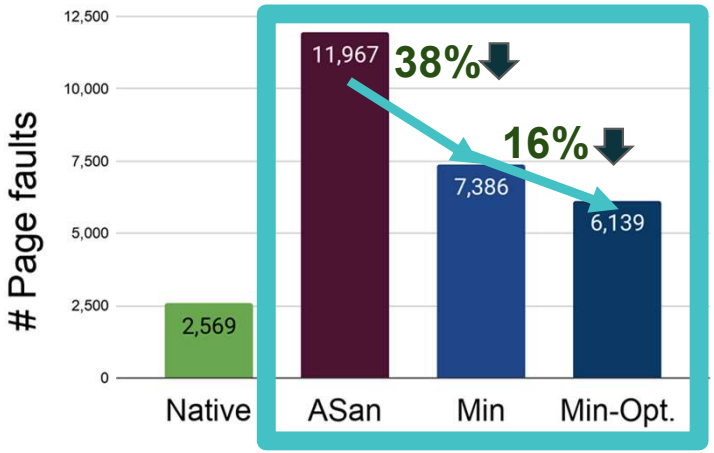
[*] The number on each bar indicates the total metadata switches

Performance Overhead Analysis

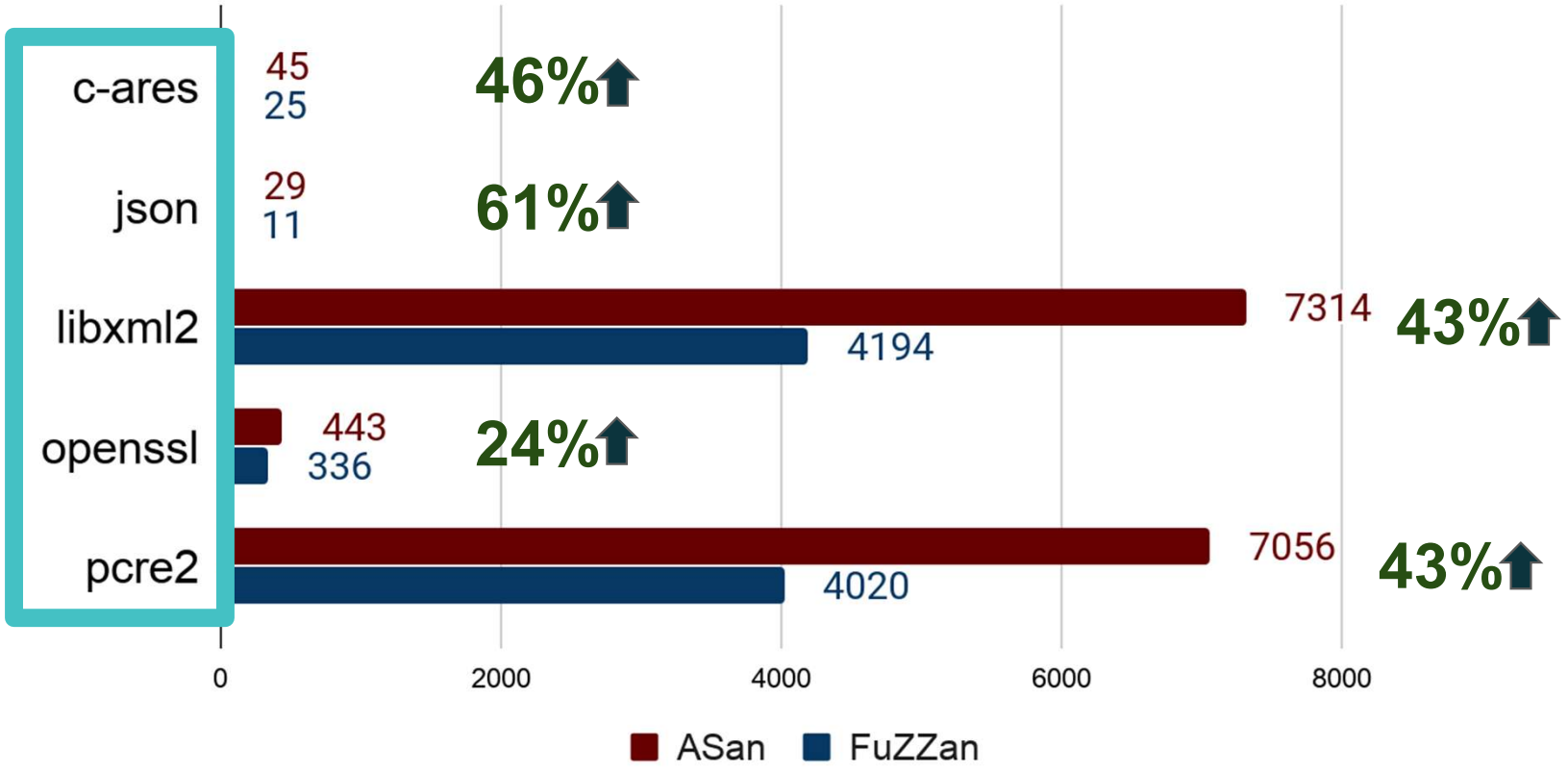
Memory management time



Page faults



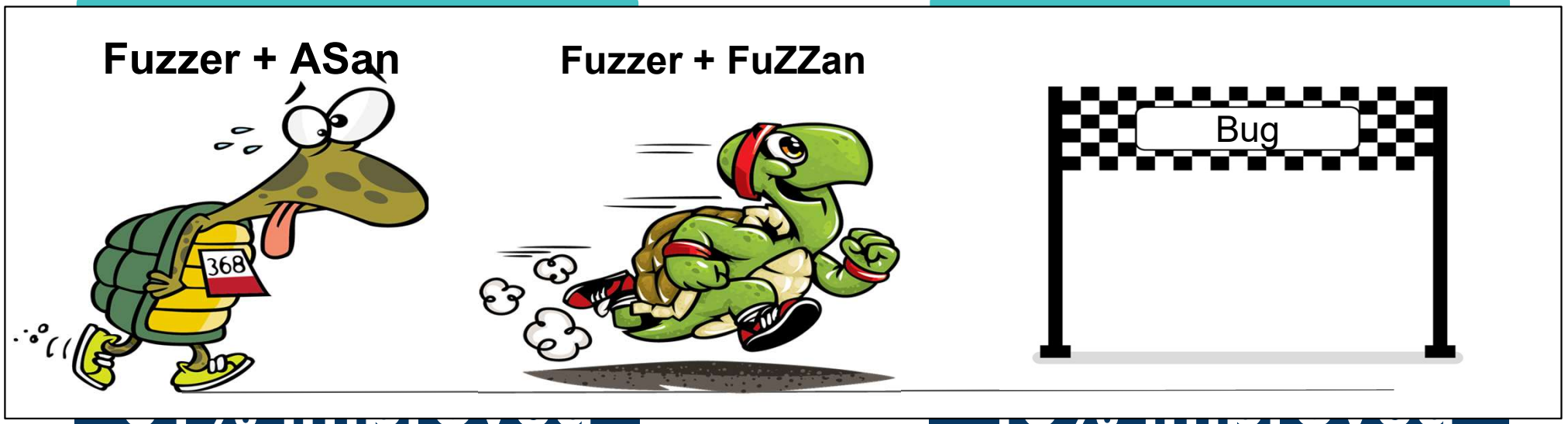
Bug Finding Speed Testing



Real-world Fuzz Testing

Total execution number

Unique discovered path



* the (M) denotes 1,000,000 (one million)

Conclusion

- Combining a fuzzer with a sanitizer hurts performance
- FuZZan reduces performance overhead
  - novel metadata structures to condense memory space
  - dynamic switching between metadata structures
  - removing unnecessary operations
- FuZZan improves fuzzing throughput over ASan
  - improves fuzzing throughput by 52%
  - discovers 13% more unique paths given the same 24 hours
  - provides flexibility to other sanitizers and AFL-based fuzzers

https://github.com/HexHive/FuZZan

Conclusion

Conclusion

- Memory/type safety violation detection still faces several challenges
- Improve existing **type safety violation** detectors
  - ➢ reduced performance overhead and increased detection coverage
  - ➢ found new bugs in real world applications
- Optimize **memory safety violation** detectors for fuzzing
  - ➢ improved fuzzing throughput
- Introduce new mitigation approach to minimize the damage caused by attacks targeting remaining bugs

Thank you, Question?