RecIPE: Revisiting the Evaluation of Memory Error Defenses

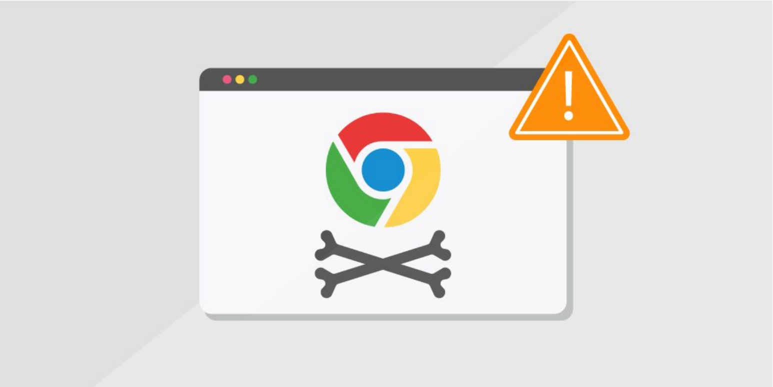
Yuancheng Jiang, Roland H.C. Yap, Zhenkai Liang, Hubert Rosier

AsiaCCS 2022

NUS National University of Singapore

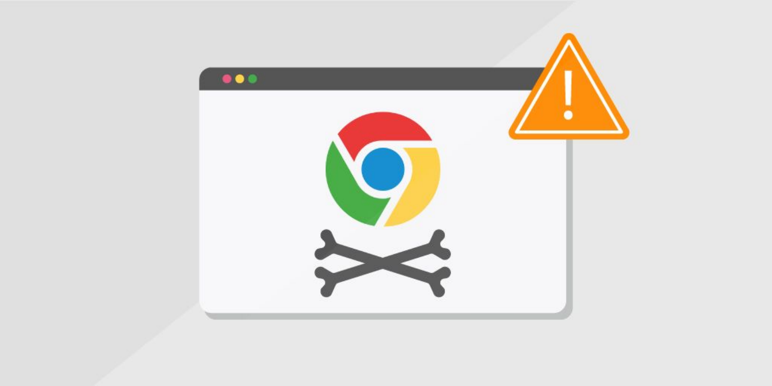
Memory Errors

- Memory errors usually form critical vulnerabilities in low-level languages



Memory Errors

- Memory errors usually form critical vulnerabilities in low-level languages

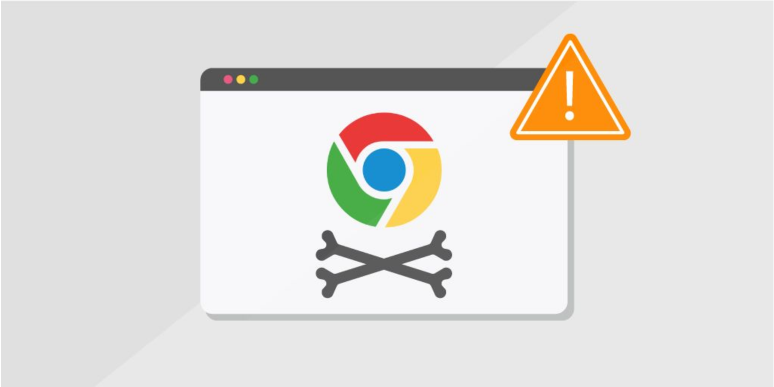


- The primary source of vulnerabilities
  - 70% severe security bugs in Chrome

Out of the 58 in-the-wild 0-days in 2021, 67% were memory corruption vulnerabilities.

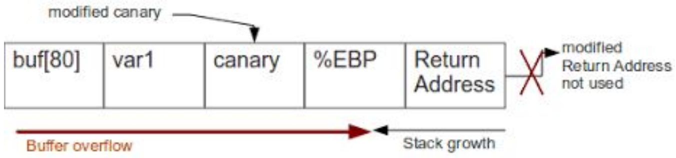
Memory Errors

- Memory errors usually form critical vulnerabilities in low-level languages



- The primary source of vulnerabilities
    - 70% severe security bugs in Chrome

- Defenses against memory errors

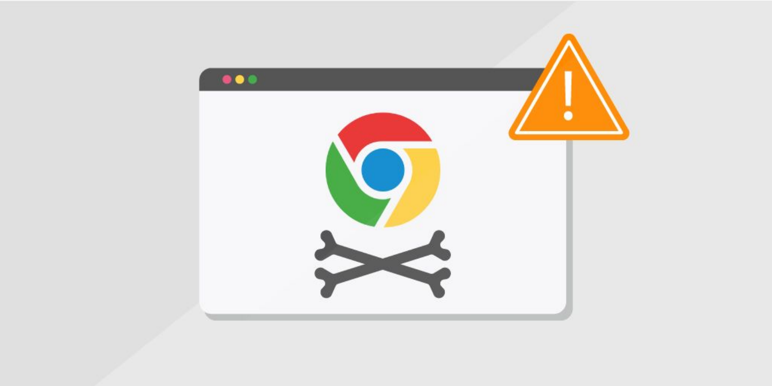


Stack Canary

Out of the 58 in-the-wild 0-days in 2021, 67% were memory corruption vulnerabilities.

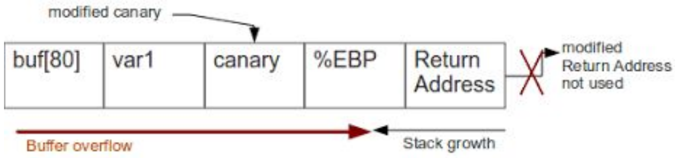
Memory Errors

- Memory errors usually form critical vulnerabilities in low-level languages

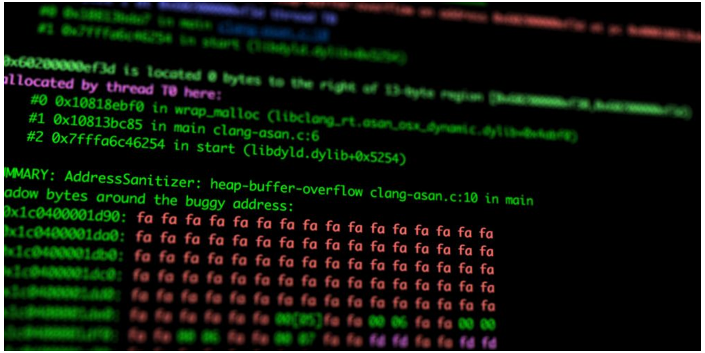


- The primary source of vulnerabilities
  - 70% severe security bugs in Chrome

- Defenses against memory errors



Stack Canary

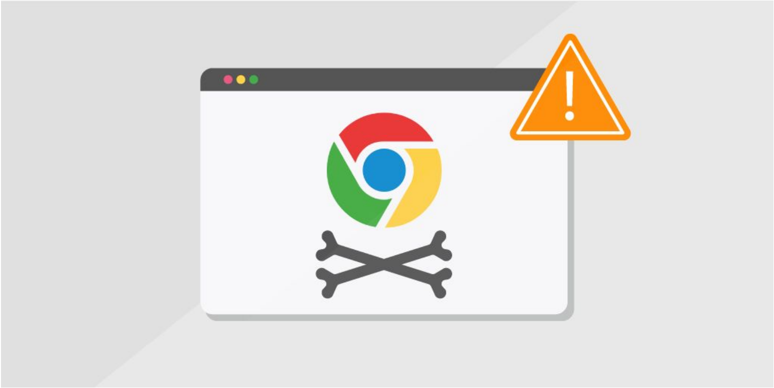


Address Sanitizer

Out of the 58 in-the-wild 0-days in 2021, 67% were memory corruption vulnerabilities.

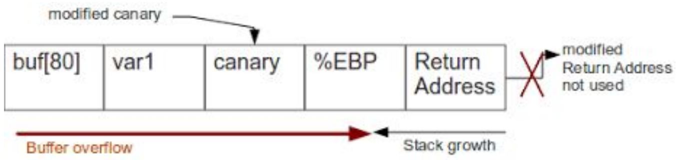
Memory Errors

- Memory errors usually form critical vulnerabilities in low-level languages

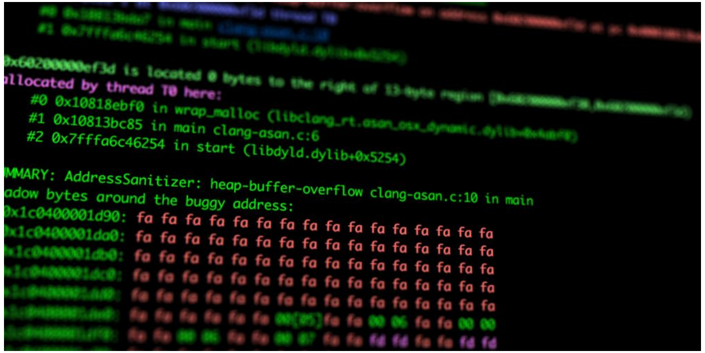


- The primary source of vulnerabilities
  - 70% severe security bugs in Chrome

- Defenses against memory errors



Stack Canary



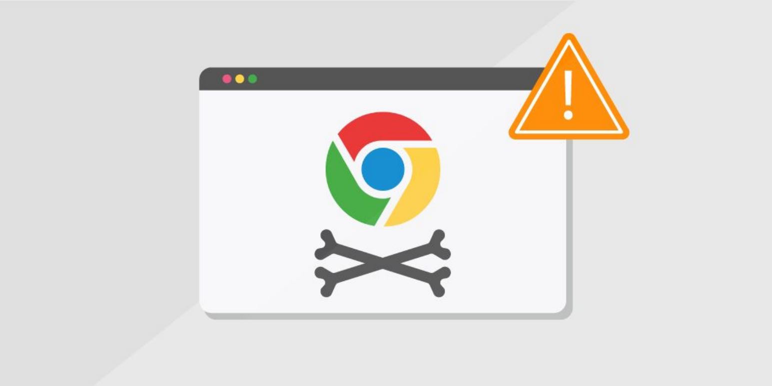
Address Sanitizer

- Tradeoffs due to constraints from overhead and compatibility.

Out of the 58 in-the-wild 0-days in 2021, 67% were memory corruption vulnerabilities.

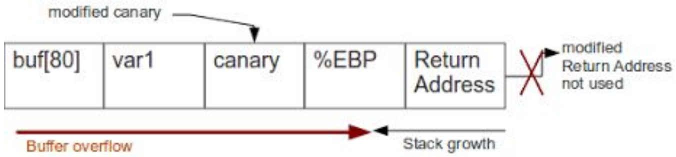
Memory Errors

- Memory errors usually form critical vulnerabilities in low-level languages

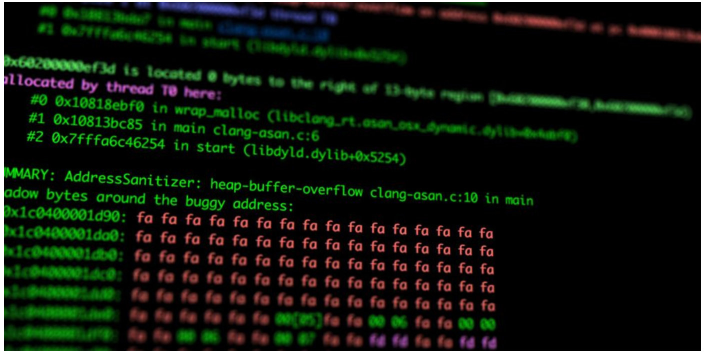


- The primary source of vulnerabilities
  - 70% severe security bugs in Chrome

- Defenses against memory errors



Stack Canary



Address Sanitizer

- Tradeoffs due to constraints from overhead and compatibility.

Question: How to accurately evaluate the security effectiveness of defenses?

Existing Evaluation Methods

Theoretical Validation

Experimental Validation

Existing Evaluation Methods

Theoretical Validation

Probability of B occurring given evidence A has already occurred

Probability of A occurring

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Probability of A occurring given evidence B has already occurred

Probability of B occurring

- Showing Probabilities
  - Hash Collision
  - Bypassing ASLR

Experimental Validation

Existing Evaluation Methods

Theoretical Validation

Probability of B occurring given evidence A has already occurred

Probability of A occurring

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Probability of A occurring given evidence B has already occurred

Probability of B occurring

- Showing Probabilities
  - Hash Collision
  - Bypassing ASLR

☹ GAP between T. and P.

Experimental Validation

Existing Evaluation Methods

Theoretical Validation

Probability of B occurring given evidence A has already occurred

Probability of A occurring

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Probability of A occurring given evidence B has already occurred

Probability of B occurring

- Showing Probabilities
  - Hash Collision
  - Bypassing ASLR

☹ GAP between T. and P.

Experimental Validation

CVEs

Common Vulnerabilities and Exposures

Real-world Case Studies

Existing Evaluation Methods

Theoretical Validation

Probability of B occurring given evidence A has already occurred

Probability of A occurring

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Probability of A occurring given evidence B has already occurred

Probability of B occurring

- Showing Probabilities
  - Hash Collision
  - Bypassing ASLR

☹ GAP between T. and P.

Experimental Validation

CVEs

Common Vulnerabilities and Exposures

Real-world Case Studies

☹ Limited Scope
Biased Choice

Existing Evaluation Methods

Theoretical Validation

Probability of B occurring given evidence A has already occurred

Probability of A occurring

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Probability of A occurring given evidence B has already occurred

Probability of B occurring

- Showing Probabilities
  - Hash Collision
  - Bypassing ASLR

☹ GAP between T. and P.

Experimental Validation

CVEs

Common Vulnerabilities and Exposures

Real-world Case Studies

☹ Limited Scope
Biased Choice

Benchmarks

Synthesized test cases

Existing Evaluation Methods

Theoretical Validation

Probability of B occurring given evidence A has already occurred

Probability of A occurring

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Probability of A occurring given evidence B has already occurred

Probability of B occurring

- Showing Probabilities
  - Hash Collision
  - Bypassing ASLR

☹ GAP between T. and P.

Experimental Validation

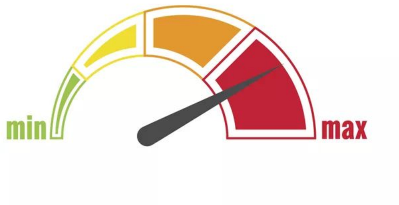
CVEs

Common Vulnerabilities and Exposures

Real-world Case Studies

☹ Limited Scope
Biased Choice

Benchmarks



Synthesized test cases

☺ Higher Coverage
Better Accuracy

Runtime Intrusion Prevention Evaluator(RIPE)

- Runtime Intrusion Prevention Evaluator(RIPE), ACSAC'11
  - RIPE generates hundreds of test cases via five dimensions

Runtime Intrusion Prevention Evaluator(RIPE)

- Runtime Intrusion Prevention Evaluator(RIPE), ACSAC'11
  - RIPE generates hundreds of test cases via five dimensions

- RIPE is widely used in evaluating memory error defenses' effectiveness
  - LBC, COTS CFI, Griffin, Fine-CFI, etc.

Runtime Intrusion Prevention Evaluator(RIPE)

- Runtime Intrusion Prevention Evaluator(RIPE), *ACSAC'11*
  - RIPE generates hundreds of test cases via five dimensions

- RIPE is widely used in evaluating memory error defenses' effectiveness
  - LBC, COTS CFI, Griffin, Fine-CFI, etc.

> "the exploits are all very similar; … the sole purpose of overflowing buffers; … it makes strong assumptions about the compiler and the operating systems.", High system-code security with low overhead, Wagner, SP'15

Runtime Intrusion Prevention Evaluator(RIPE)

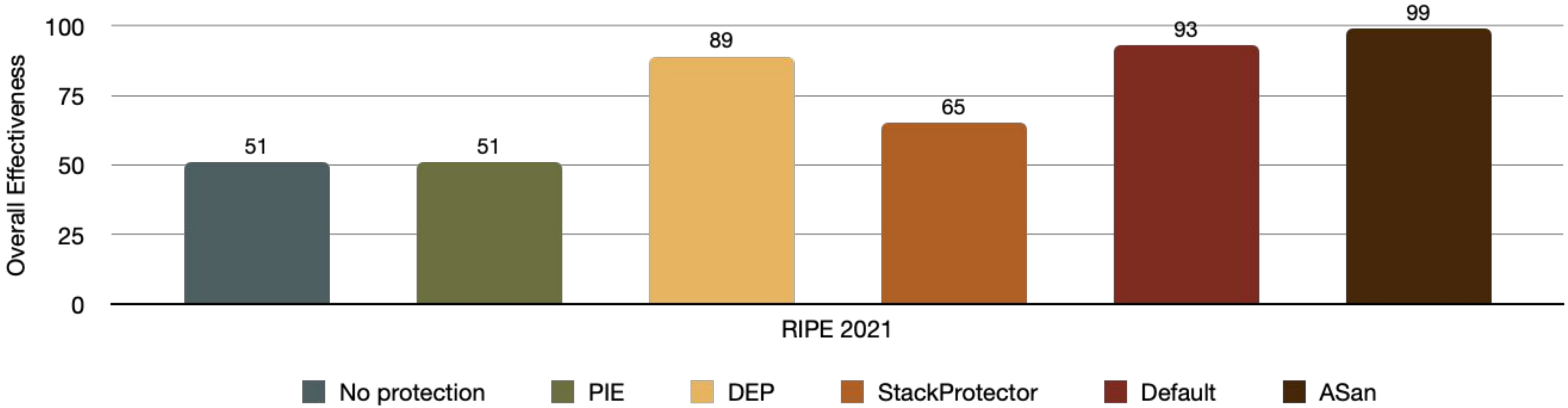
- Runtime Intrusion Prevention Evaluator(RIPE), ACSAC'11
  - RIPE generates hundreds of test cases via five dimensions

- RIPE is widely used in evaluating memory error defenses' effectiveness
  - LBC, COTS CFI, Griffin, Fine-CFI, etc.

> "the RIPE testbed only considers a limited number of buffer overflow vulnerabilities. There could be many other types of vulnerabilities and exploit skills in practice", Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms, Yuan, CCS'20

RIPE(2011) Performance in 2021

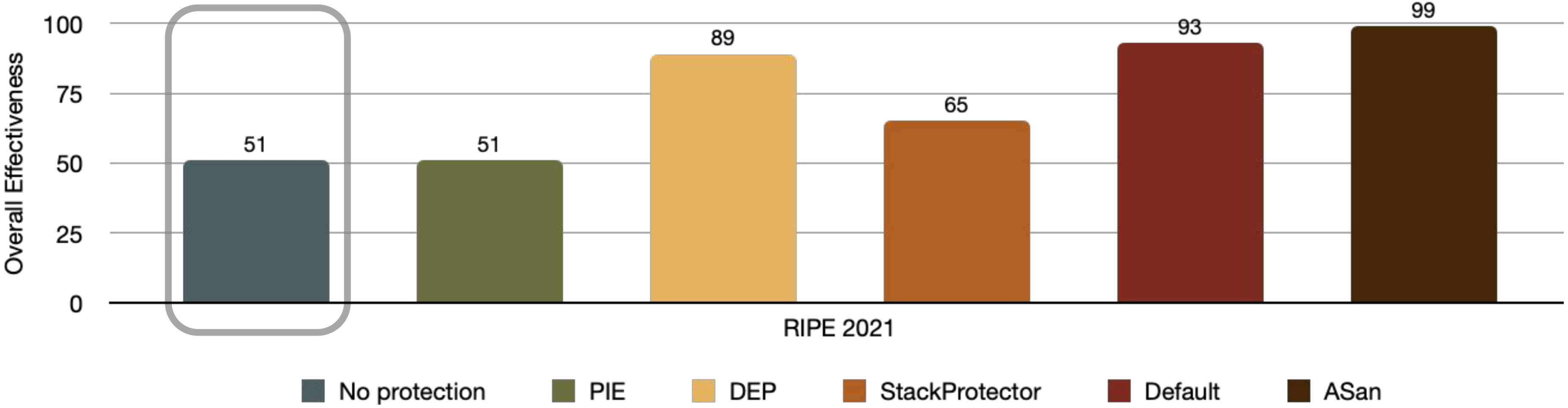
Effectiveness = Attacks/Exploits Prevented Rate



RIPE(2011) Performance in 2021

Effectiveness = Attacks/Exploits Prevented Rate

☹ Inaccurate Baseline Result – 51% (many RIPE exploits fail)

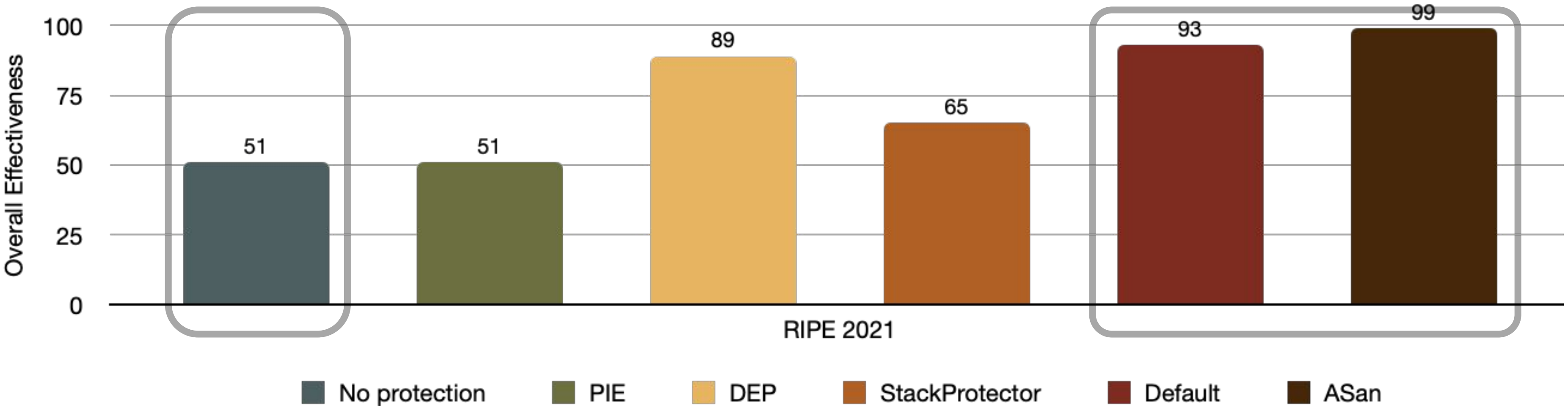


RIPE(2011) Performance in 2021

Effectiveness = Attacks/Exploits Prevented Rate

☹ Inaccurate Baseline Result - 51% (many RIPE exploits fail)

=> Overrated Effectivenesses on defenses - over 90%



RecIPE

An **extensible** and **comprehensive** successor to RIPE

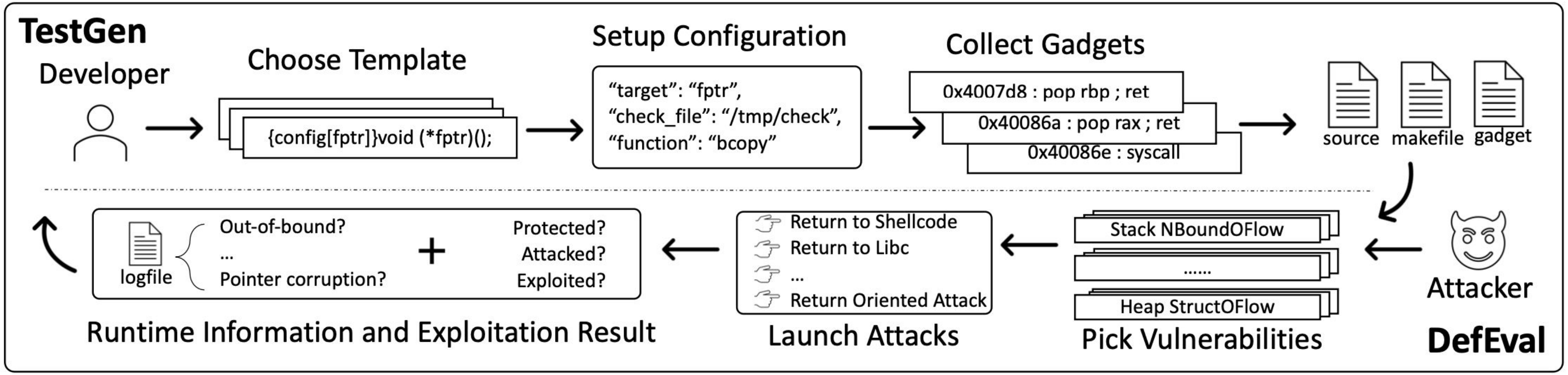
Design Goals

- ☺ Easy to extend and customize
- ☺ Diverse testcases and higher coverage
- ☺ Accurate and comprehensive measurement
- ☺ Support both 32 and 64 bits architecture

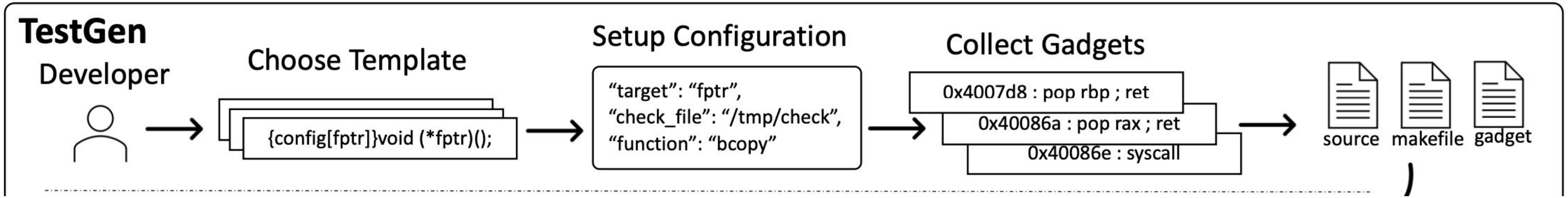
Benchmark Overview

Two Components:

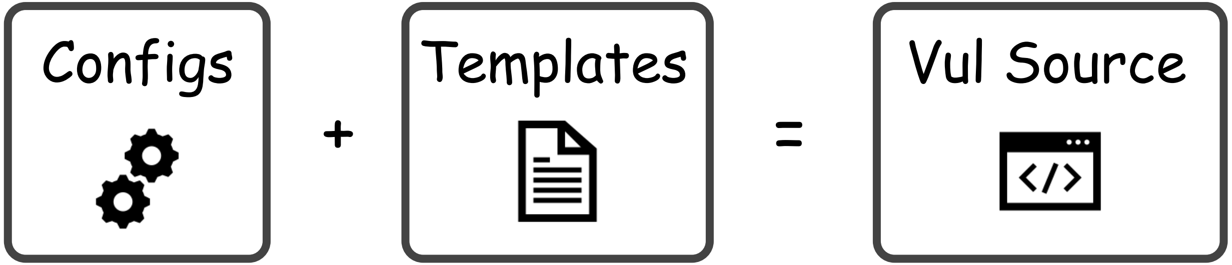
- TestGen: Generating Testcases to Individual Folders
- DefEval: Mimicing Attackers; Giving Analysis Results

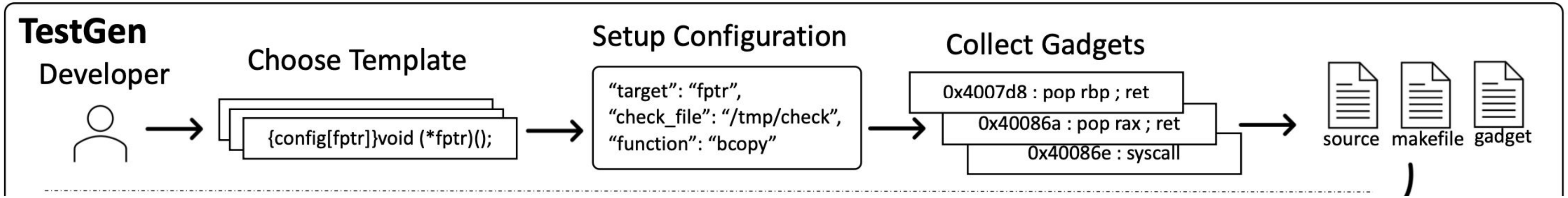


TestGen

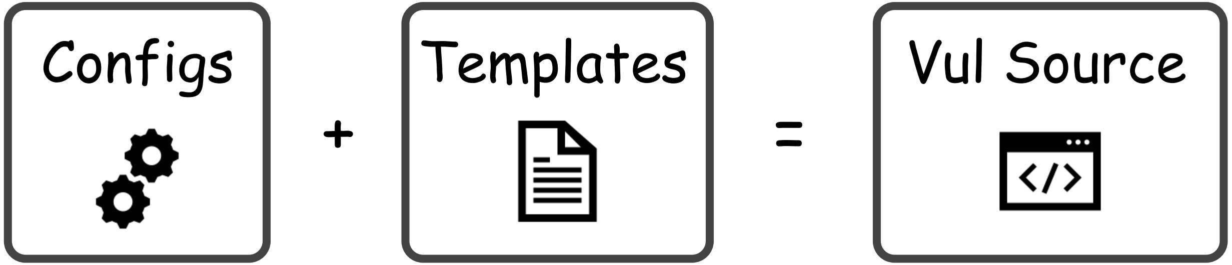

TestGen
Developer
Choose Template
{config[fptr]}void (*fptr)();
Setup Configuration
"target": "fptr",
"check_file": "/tmp/check",
"function": "bcopy"
Collect Gadgets
0x4007d8 : pop rbp ; ret
0x40086a : pop rax ; ret
0x40086e : syscall
source
makefile
gadget

TestGen

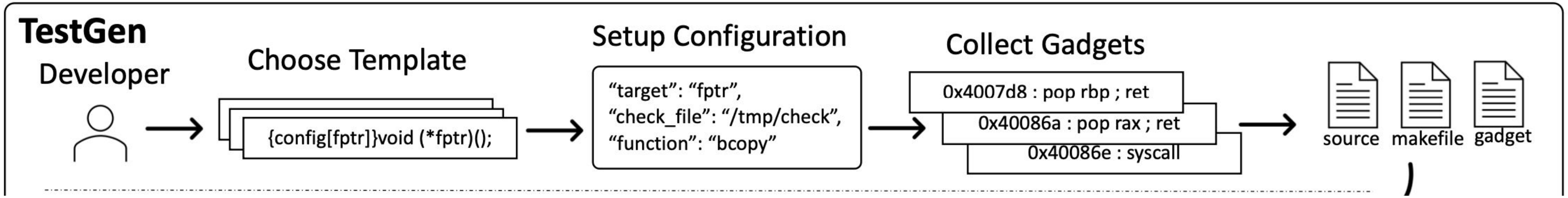

Configs
+
Templates
=
Vul Source


TestGen
Developer
Choose Template
{config[fptr]}void (*fptr)();
Setup Configuration
"target": "fptr",
"check_file": "/tmp/check",
"function": "bcopy"
Collect Gadgets
0x4007d8 : pop rbp ; ret
0x40086a : pop rax ; ret
0x40086e : syscall
source
makefile
gadget

TestGen



Configurable Templates => Extensibility & Customizability



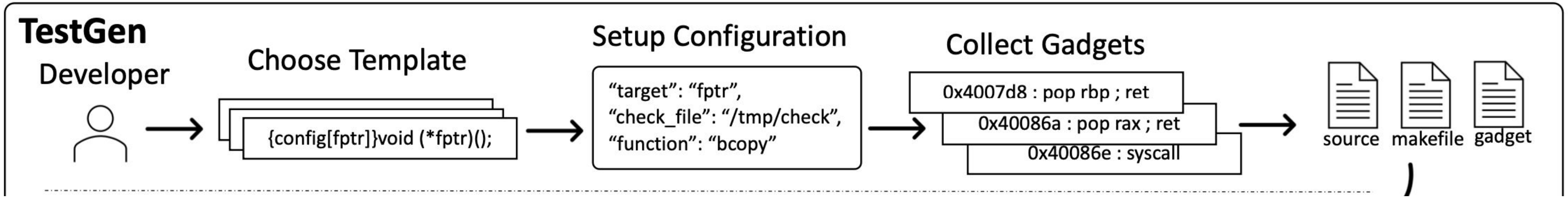
TestGen

Configs + Templates = Vul Source

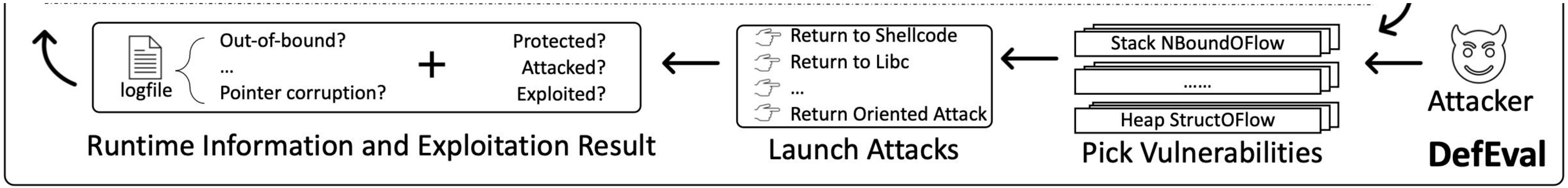
Configurable Templates => Extensibility & Customizability

Challenges:

- To be compatible with various attributes
- To switch context under various scenarios



DefEval



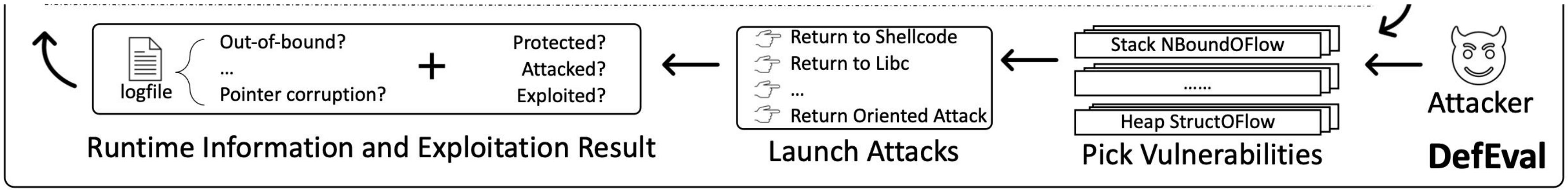
Out-of-bound?
...
Pointer corruption?
logfile
+
Protected?
Attacked?
Exploited?
Runtime Information and Exploitation Result
Return to Shellcode
Return to Libc
...
Return Oriented Attack
Launch Attacks
Stack NBoundOFlow
......
Heap StructOFlow
Pick Vulnerabilities
Attacker
DefEval

DefEval

Attacker

Equipped with exploits:

- return-to-shellcode
- return-to-libc
- ROP, SROP



DefEval

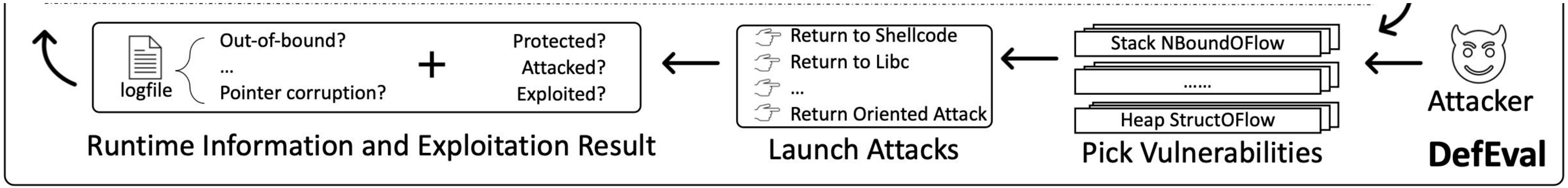
Attacker

Equipped with exploits:

- return-to-shellcode
- return-to-libc
- ROP, SROP

Analyzer

- Runtime Log => Attacked?
- File Existence => Exploited?



DefEval

Attacker

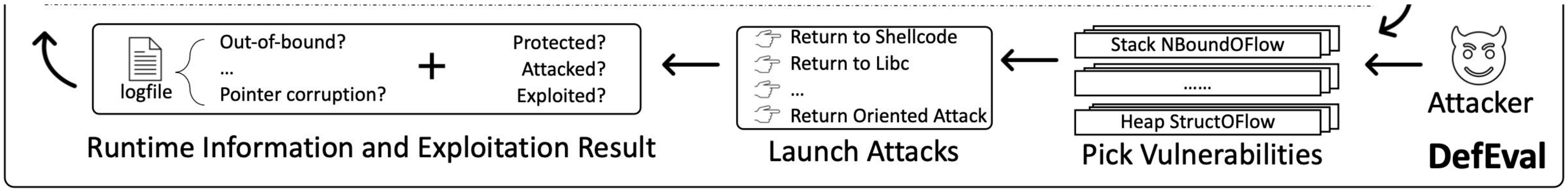
Equipped with exploits:

- return-to-shellcode
- return-to-libc
- ROP, SROP

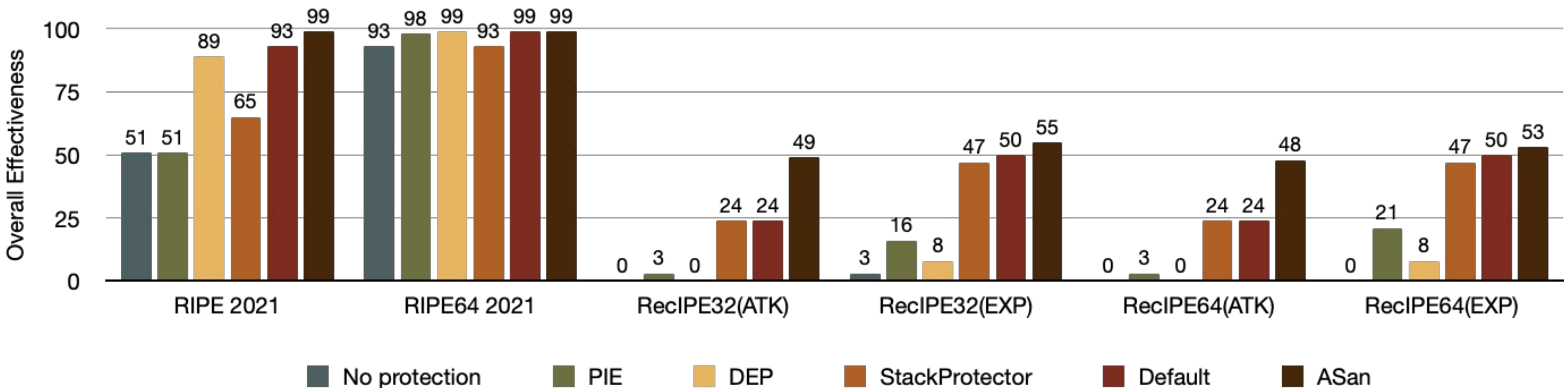
Analyzer

- Runtime Log => Attacked?
- File Existence => Exploited?

Attack? + Exploit?
=> Effectiveness Results

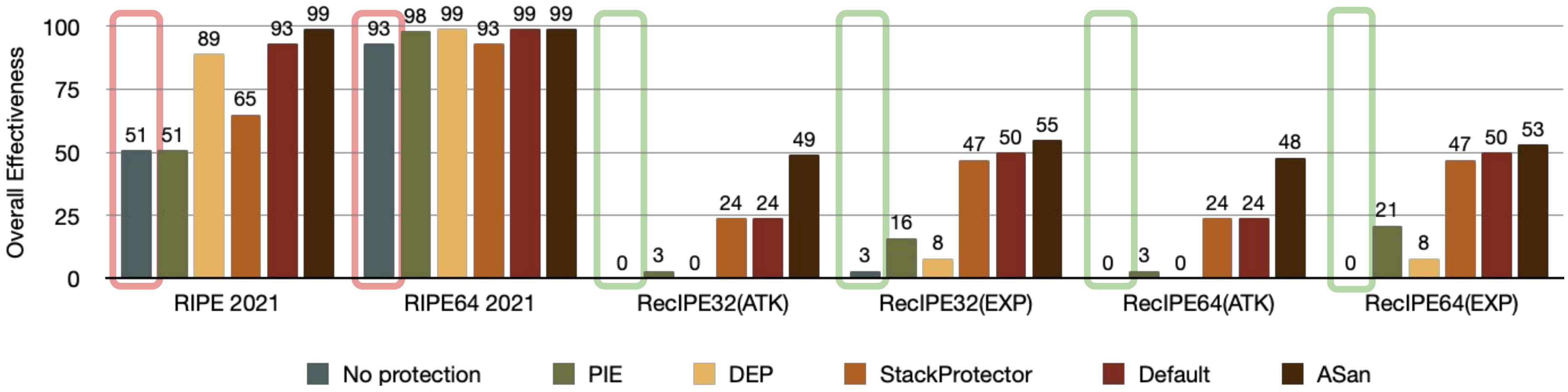


Effectiveness Evaluation



Effectiveness Evaluation

☺ Accurate Baseline — No Protection 0%

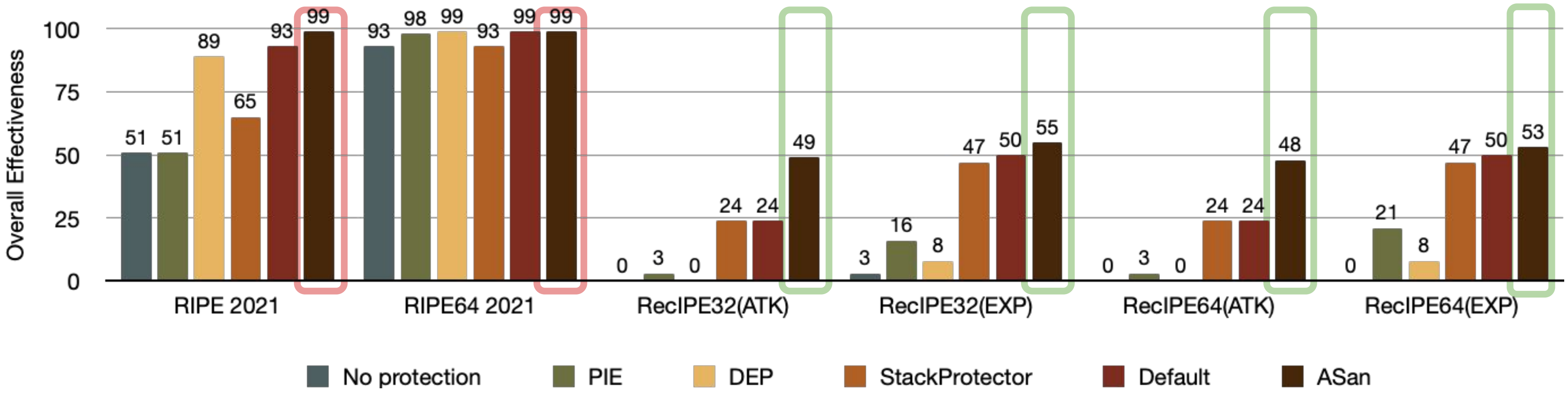


Effectiveness Evaluation

☺ Accurate Baseline — No Protection 0%

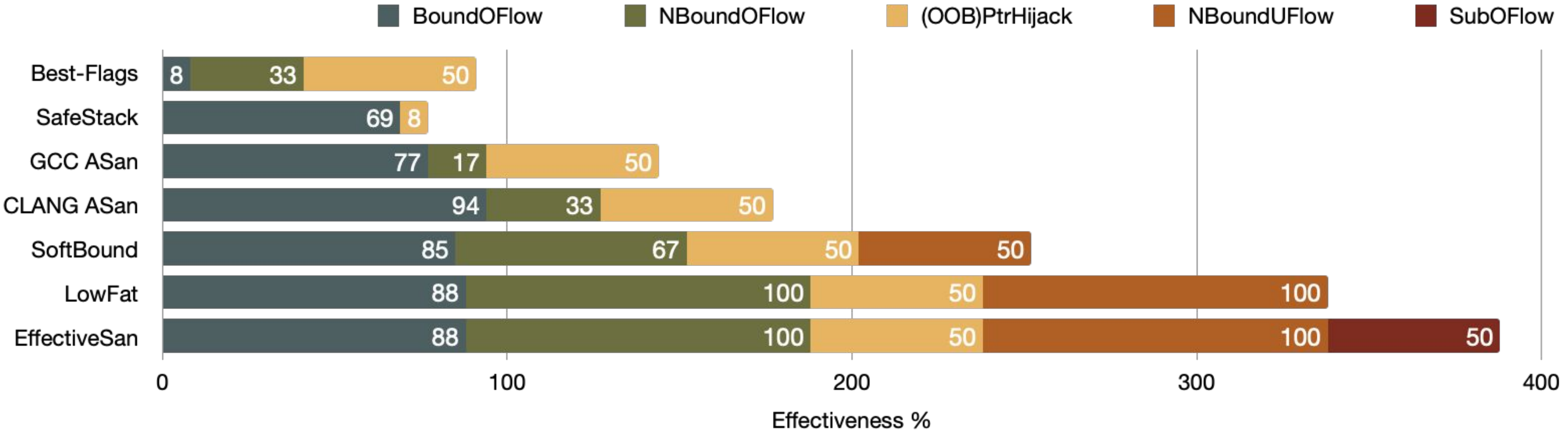
☺ Moderate Score — ASan 50%

=> Reserve space for stronger defense



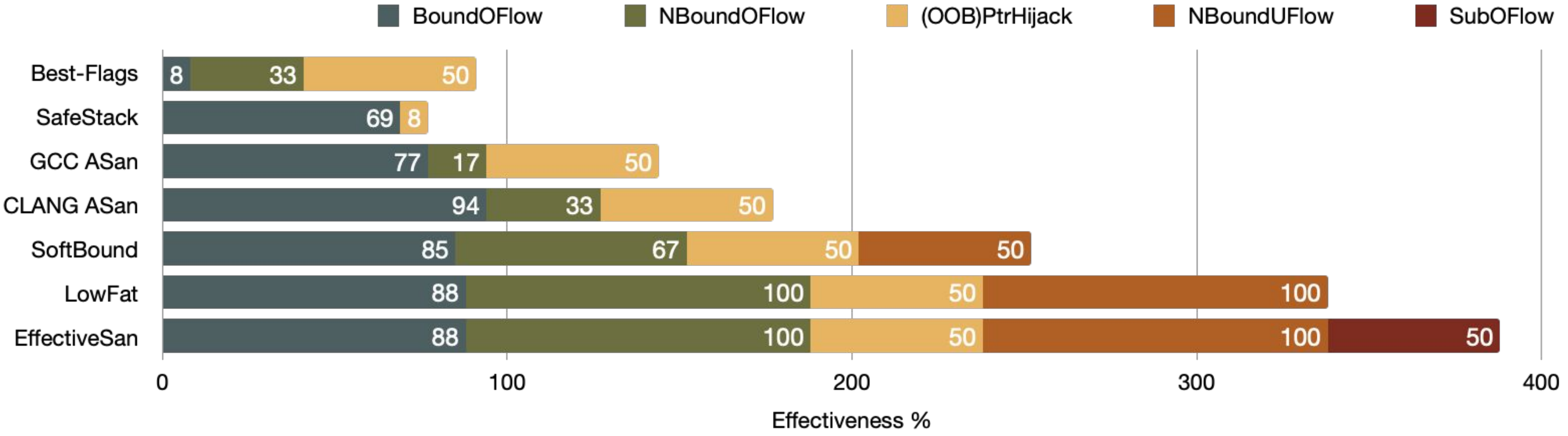
Effectiveness towards various defenses

- Clearly present pros and cons of each defense



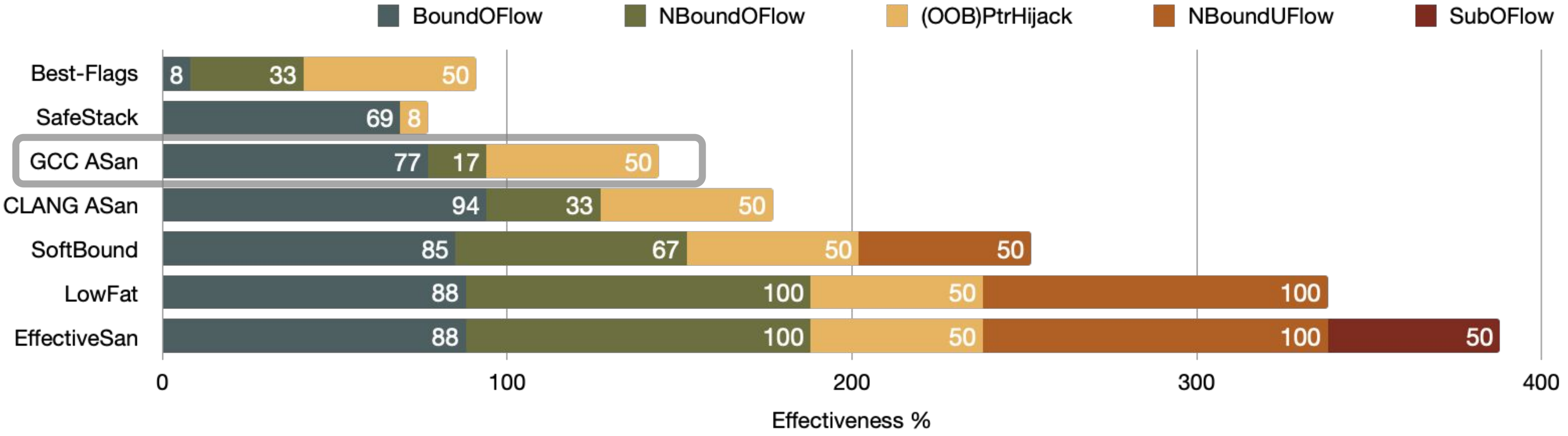
Effectiveness towards various defenses

- Clearly present pros and cons of each defense
- Help to understand limitations:



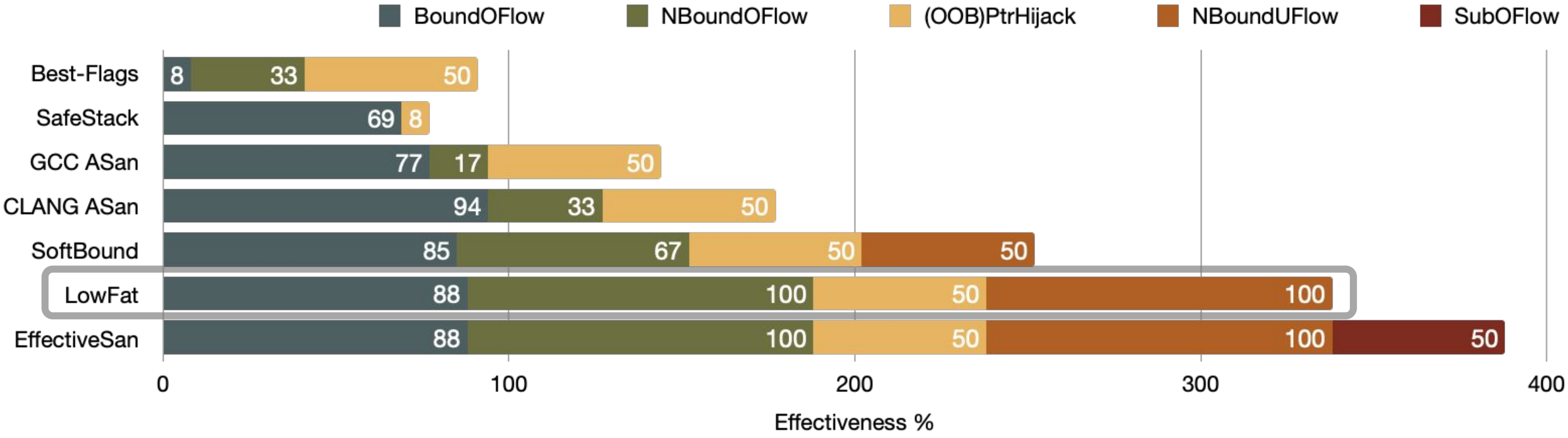
Effectiveness towards various defenses

- Clearly present pros and cons of each defense
- Help to understand limitations:
  - ASan does not protect non-linear OOB



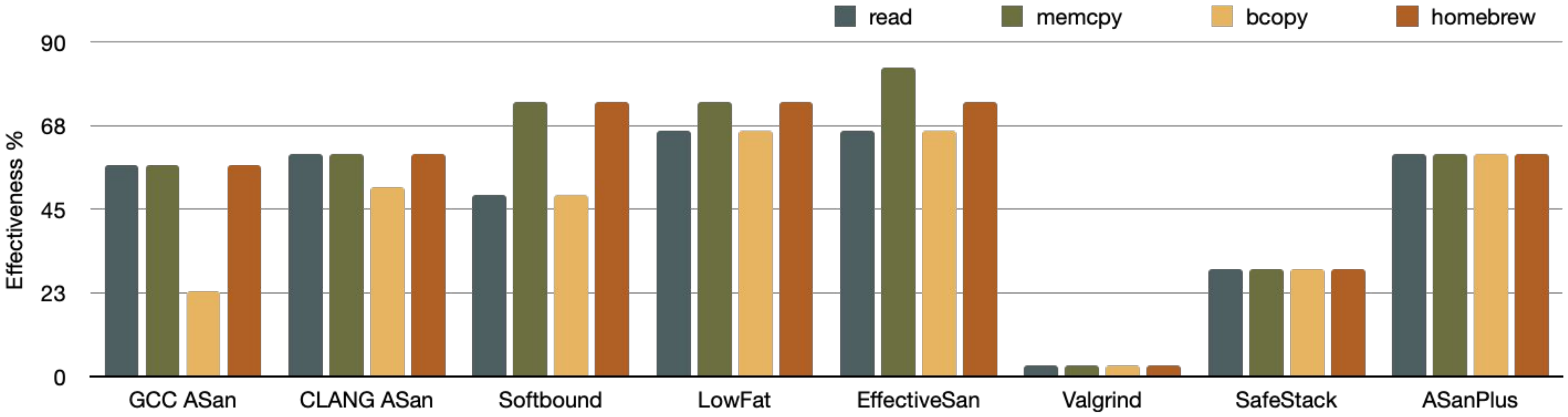
Effectiveness towards various defenses

- Clearly present pros and cons of each defense
- Help to understand limitations:
  - ASan does not protect non-linear OOB
  - LowFat does not protect Sub-Object Overflow



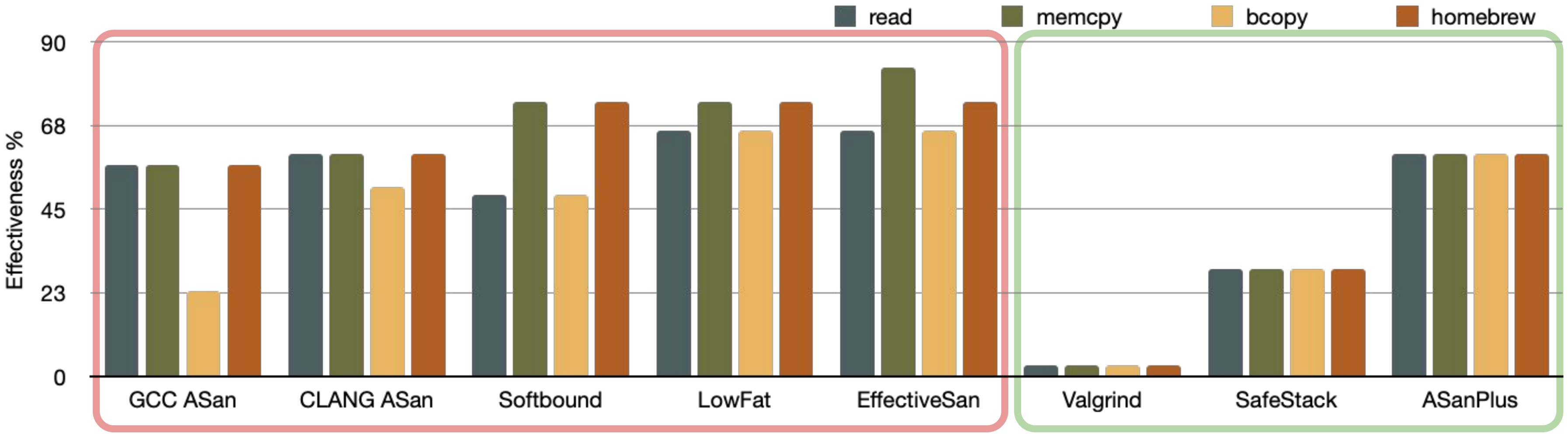
Some Surprises

- Special Hardenings on Special Functions



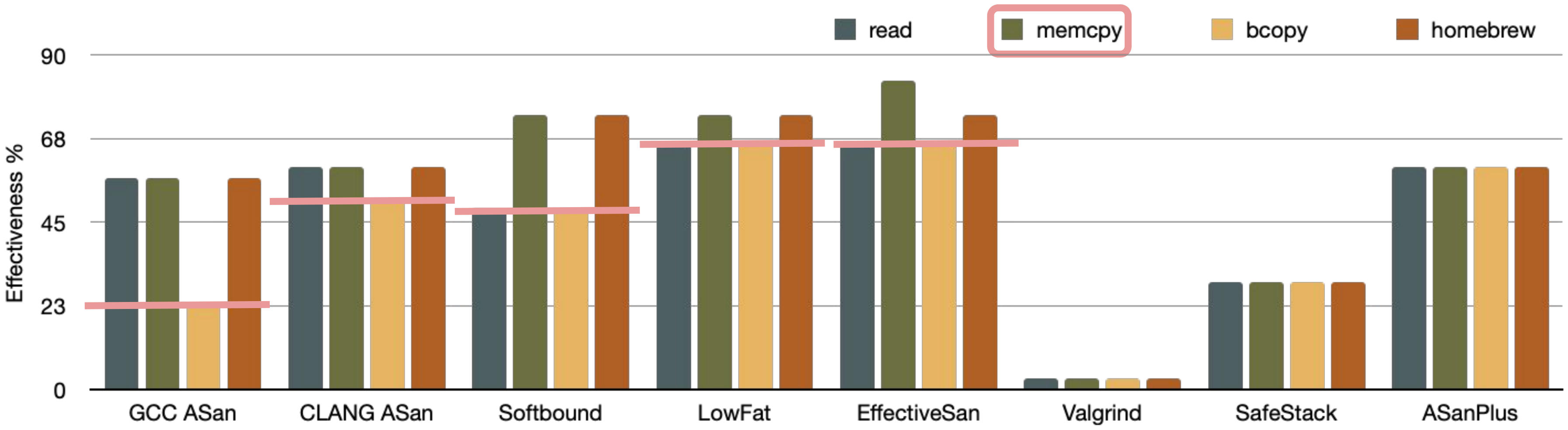
Some Surprises

- Special Hardenings on Special Functions



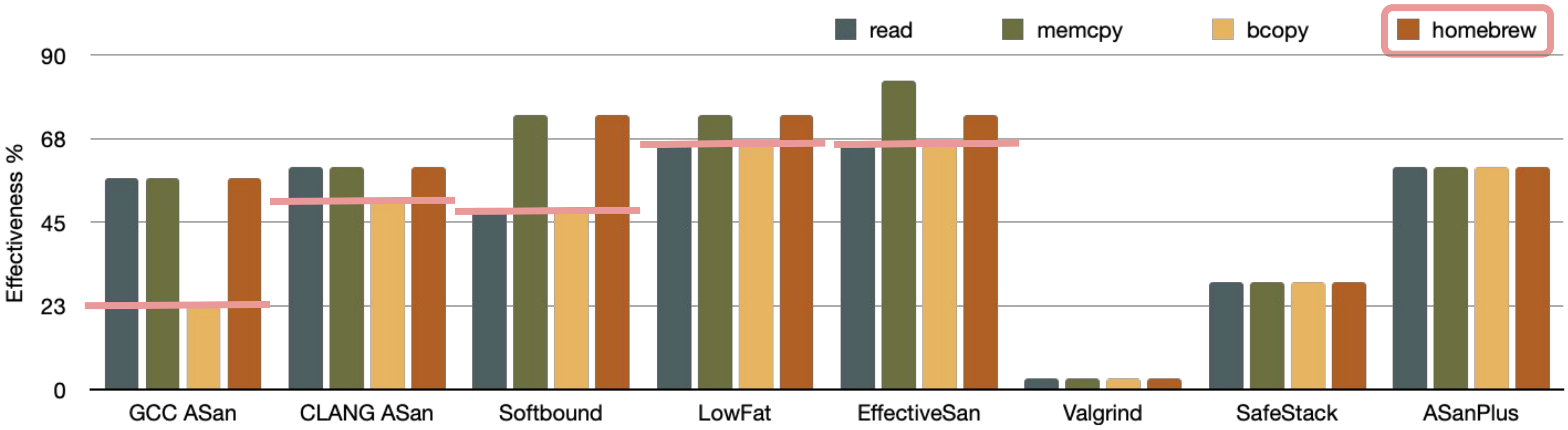
Some Surprises

- Special Hardenings on Special Functions
  - memcpy => “memcpy_safe”



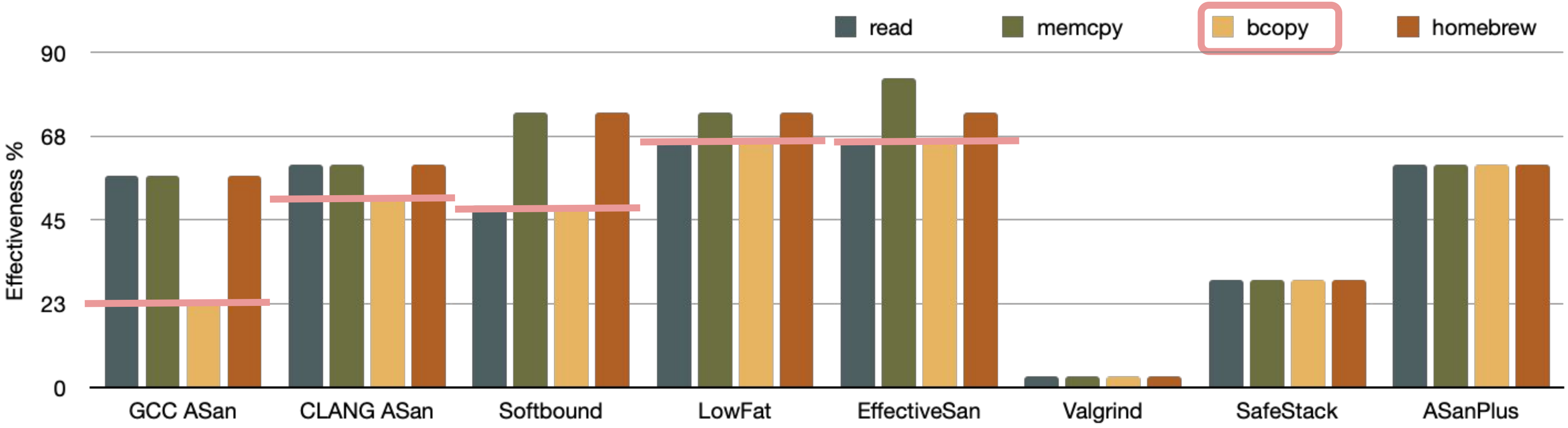
Some Surprises

- Special Hardenings on Special Functions
  - memcpy => “memcpy_safe”
  - homebrew memcpy => get “instrumented”



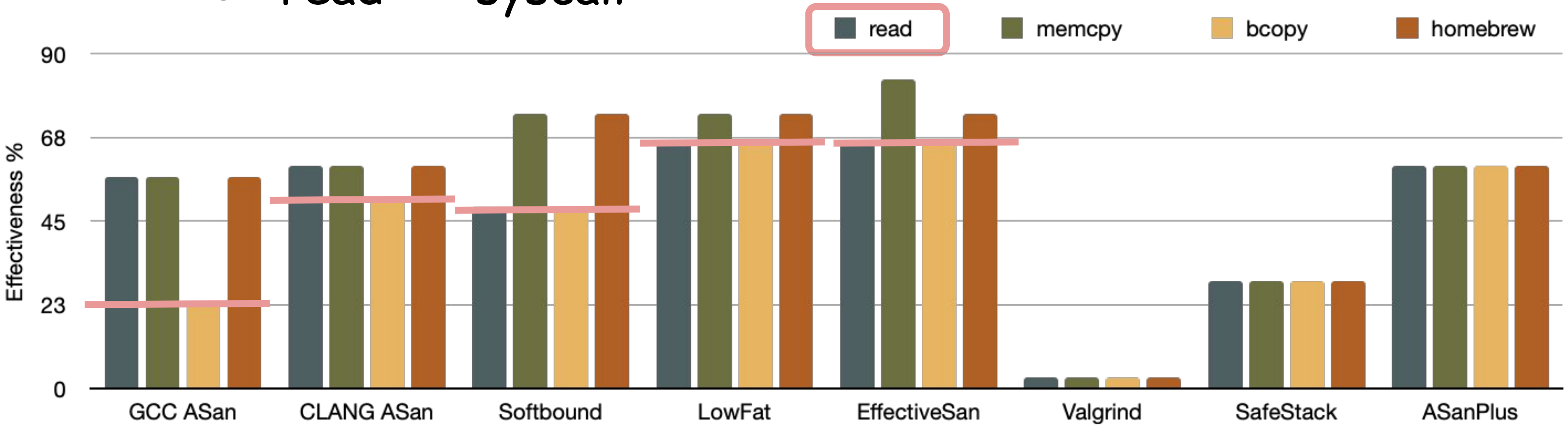
Some Surprises

- Special Hardenings on Special Functions
  - memcpy => "memcpy_safe"
  - homebrew memcpy => get "instrumented"
  - bcopy => not widely considered



Some Surprises

- Special Hardenings on Special Functions
  - memcpy => "memcpy_safe"
  - homebrew memcpy => get "instrumented"
  - bcopy => not widely considered
  - read => "syscall"



More Detailed Results

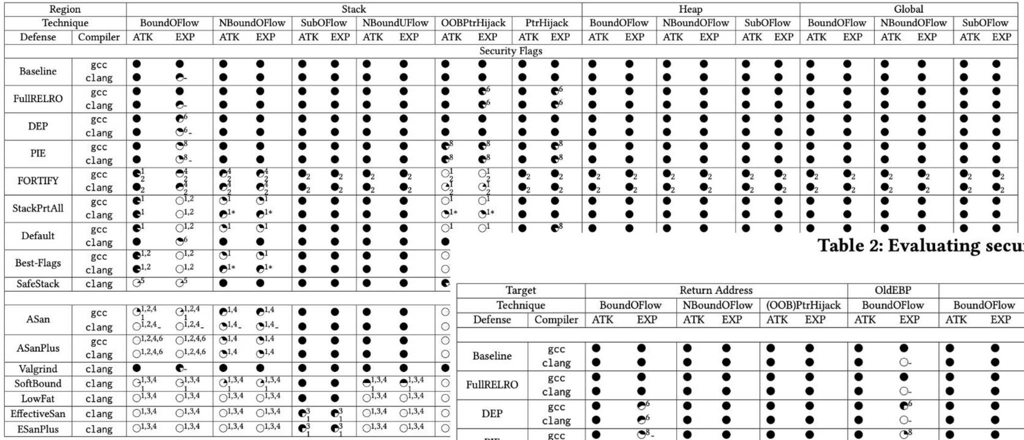
Table 1: Evaluating security flags and sanitizers from region and technique attributes

Region		Stack												Heap						Global					
Technique		BoundOFlow		NBoundOFlow		SubOFlow		NBoundUFlow		OOBPtrHijack		PtrHijack		BoundOFlow		NBoundOFlow		SubOFlow		BoundOFlow		NBoundOFlow		SubOFlow	
Defense	Compiler	ATK	EXP	ATK	EXP	ATK	EXP	ATK	EXP	ATK	EXP	ATK	EXP	ATK	EXP	ATK	EXP	ATK	EXP	ATK	EXP	ATK	EXP	ATK	EXP
Security Flags																											
Baseline	gcc	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	clang	●	◓-	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
FullRELRO	gcc	●	●	●	●	●	●	●	●	●	◕[6]	●	◕[6]	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	clang	●	◓-	●	●	●	●	●	●	●	◕[6]	●	◕[6]	●	●	●	●	●	●	●	●	●	●	●	●	●	●
DEP	gcc	●	◕[6]	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	clang	●	◔[6]-	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
PIE	gcc	●	◔[8]	●	●	●	●	●	●	◕[8]	◕[8]	●	◕[8]	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	clang	●	◔[8]-	●	●	●	●	●	●	◕[8]	◕[8]	●	◕[8]	●	●	●	●	●	●	●	●	●	●	●	●	●	●
FORTIFY	gcc	◕[1]_2	◓[4]_2	◕[4]_2	◕[4]_2	●_2	●_2	●_2	●_2	○[1]_2	○[1]_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2
	clang	●_2	◕[4]_2	◕[4]_2	◕[4]_2	●_2	●_2	●_2	●_2	◔[1]_2	◔[1]_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2	●_2
StackPrtAll	gcc	◕[1]	○[1,2]	◔[1]	◔[1]	●	●	●	●	○[1]	○[1]	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	clang	◕[1]	○[1,2]	◕[1*]	◕[1*]	●	●	●	●	◔[1*]	◔[1*]	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Default	gcc	◕[1]	○[1,2]	◔[1]	◔[1]	●	●	●	●	○[1]	○[1]	●	◕[8]	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	clang	●	◔[6]	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Best-Flags	gcc	◕[1,2]	○[1,2]	◔[1]	◔[1]	●	●	●	●	○[18]	○[168]	●	◕[68]	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	clang	◕[1,2]	○[1,2]	◕[1*]	◕[1*]	●	●	●	●	○[18]	○[168]	●	◕[68]	●	●	●	●	●	●	●	●	●	●	●	●	●	●
SafeStack	clang	◔[5]	◔[5]	●	●	●	●	●	●	◕-	◕-	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Sanitizers																											
ASan	gcc	◔[1,2,4]_1	◔[1,2,4]_1	◕[1,4]	◕[1,4]	●	●	●	●	○[1,2,4]	○[1,2,4]	●	◕[4]	◔[2,4]_1	◔[2,4]_1	●	●	●	●	◔[2,4]_1	◔[2,4]_1	●	●	●	●
	clang	○[1,2,4]-	○[1,2,4]-	◔[1,4]-	◔[1,4]-	●	●	●	●	○[1,2,4]	○[1,2,4]	●	◕[4]	◔[2,4]_1	◔[2,4]_1	●	●	●	●	◔[2,4]_1	◔[2,4]_1	●	●	●	●
ASanPlus	gcc	○[1,2,4,6]	○[1,2,4,6]	◔[1,4]	◔[1,4]	●	●	●	●	○[1,2,4]	○[1,2,4]	●	◕[4,6,8]	○[2,4]	○[2,4]	●	●	●	●	○[2,4]	○[2,4]	●	●	●	●
	clang	○[1,2,4,6]	○[1,2,4,6]	◔[1,4]	◔[1,4]	●	●	●	●	○[1,2,4]	○[1,2,4]	●	◕[4,6,8]	○[2,4]	○[2,4]	●	●	●	●	○[2,4]	○[2,4]	●	●	●	●
Valgrind	clang	●	◕-	●	●	●	●	●	●	●	◕[9]	●	◕[9]	○[7]	○[7]	●	●	●	●	●	●	●	●	●	●
SoftBound	clang	◔[1,3,4]_1	◔[1,3,4]_1	◔[1,3,4]_1	◔[1,3,4]_1	●	●	◓[1,3,4]_1	◓[1,3,4]_1	○[1,3,4]	○[1,3,4]	●	◔-	◓[1,3,4]_1	◓[1,3,4]_1	◓[1,3,4]_1	◓[1,3,4]_1	●	●	◓[1,3,4]_1	◓[1,3,4]_1	◓[1,3,4]_1	◓[1,3,4]_1	●	●
LowFat	clang	○[1,3,4]	○[1,3,4]	○[1,3,4]	○[1,3,4]	●	●	○[1,3,4]	○[1,3,4]	○[1,3,4]	○[1,3,4]	●	◕[9]	◓[3,4]_1	◓[3,4]_1	○[3,4]	○[3,4]	●	●	◓[3,4]_1	◓[3,4]_1	○[3,4]	○[3,4]	●	●
EffectiveSan	clang	○[1,3,4]	○[1,3,4]	○[1,3,4]	○[1,3,4]	◕[3]_1	◕[3]_1	○[1,3,4]	○[1,3,4]	○[1,3,4]	○[1,3,4]	●	◕[9]	◓[3,4]_1	○[3,4]	○[3,4]	○[3,4]	◕[3,4]_1	◕[3,4]_1	◓[3,4]_1	○[3,4]	○[3,4]	○[3,4]	◕[3,4]_1	◕[3,4]_1
ESanPlus	clang	○[1,3,4]	○[1,3,4]	○[1,3,4]	○[1,3,4]	◕[3]_1	◕[3]_1	○[1,3,4]	○[1,3,4]	○[1,3,4]	○[1,3,4]	●	◕[6,9]	◓[3,4]_1	○[3,4]	○[3,4]	○[3,4]	◕[3,4]_1	◕[3,4]_1	◓[3,4]_1	○[3,4]	○[3,4]	○[3,4]	◕[3,4]_1	◕[3,4]_1

Symbol	Meaning
○	All ATK/EXP Fail
◓	Half ATK/EXP Fail
●	All ATK/EXP Succeed
Super	Defense Strategy
◔[1]	Memory Layout Change
◔[1*]	`clang` Memory Layout
◔[2]	Guard/Redzone checks
◔[3]	Bound Checks
◔[4]	Compiler Hardening
◔[5]	Seperate Stack
◔[6]	Memory Permissions
◔[7]	Dynamic Instrumentation
◔[8]	Randomization
◔[9]	Allocator Change
Sub	Defense Weakness
◔_1	Missed Check
◔_2	Analysis Limitation
Suffix	Other Reasons
◔-	ATK/EXP Not Robust

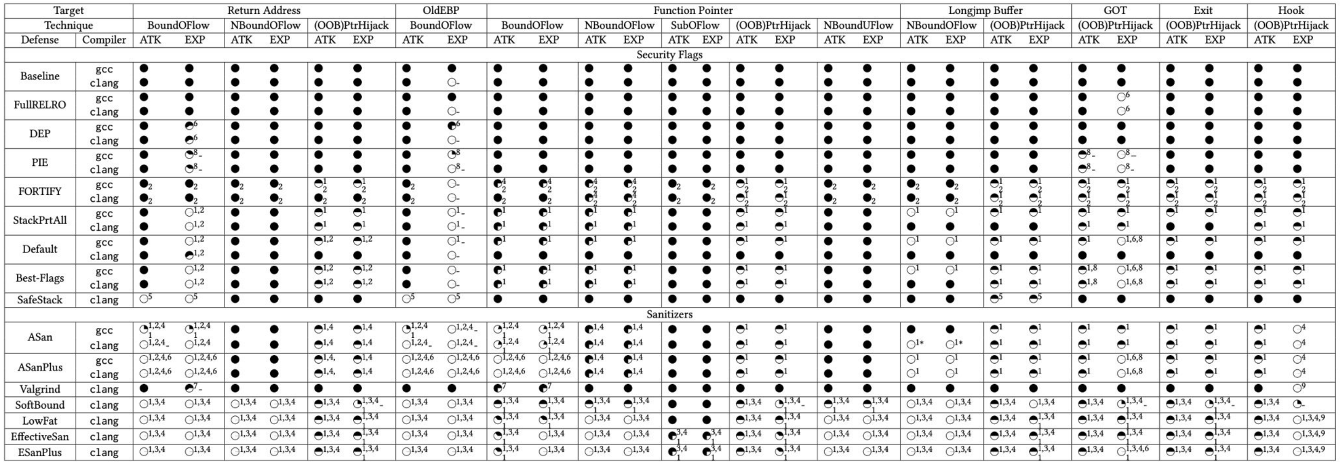
More Detailed Results

Table 1: Evaluating security flags and sanitizers from region and technique attributes



Symbol	Meaning
○	All ATK/EXP Fail
◓	Half ATK/EXP Fail
●	All ATK/EXP Succeed
Super	**Defense Strategy**
◔[1]	Memory Layout Change
◔[1*]	clang Memory Layout
◔[2]	Guard/Redzone checks
◔[3]	Bound Checks
◔[4]	Compiler Hardening
◔[5]	Seperate Stack
◔[6]	Memory Permissions
[illegible]	[illegible]

Table 2: Evaluating security flags and sanitizers from target and technique attributes



Summary

- We propose a newly designed benchmark, RecIPE, for evaluating memory error defenses' effectiveness.
- RecIPE is extensible, comprehensive, and accurate.
- RecIPE helps to understand the security coverage of memory error defense and even implementation details.
- RecIPE is available at https://github.com/YuanchengJiang/recipe-benchmark

Thanks

National University of Singapore