SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs

CS3612, Operating System, Paper Reading

Chaofan Lin

2022. 6

Part I

Introduction

Overview Background: Sanitizer Motivation

SanRazor[1]

- Sanitizers are used widely in software but the overhead of sanitizer checks is is high.
- This Paper proposed a method called SanRazor to reduce redundant checks.
- How to define **redundant**: by comparing their Dynamic Patterns and Static Patterns.
- How to reduce: when detecting two redundant checks, remove the check which is dominated by the other.

---

[1] ZHANG J, et al. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs[C/OL]//15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). [S.l.]: USENIX Association, 2021: 479-494. https://www.usenix.org/conference/osdi21/presentation/zhang.

Sanitizer

SHANGHAI JIAO TONG UNIVERSITY

Problem Languages like C/C++ are unsafe.

Solutions

- Some tools like Valgrind are proposed to detect CVEs previously.
- Sanitizers: faster, more systematical, integrated in compilers (LLVM, gcc).

Sanitizer

Sanitizers insert sanitizer checks dynamically.
Different sanitizers are designed to detect different types of errors:

- AddressSanitizer
- ThreadSanitizer
- MemorySanitizer
- ...

ASan: AddressSanitizer[2]

SHANGHAI JIAO TONG UNIVERSITY

Target Memory Errors like buffer overflow and use-after-free (UAF).

Detail An instrumentation module (allocates shadow memory regions for each used address) and a runtime library (hooks malloc and free).

Example: A Memory Error

```cpp
int main(){
    int* p = new int;
    delete p;
    *p = 3; // use-after-free
}
```

---

[2] SEREBRYANY K, et al. AddressSanitizer: A Fast Address Sanity Checker[C]//2012 USENIX Annual Technical Conference (USENIX ATC 12). Boston, MA: USENIX Association, 2012: 309-318.

UBSan: UndefinedBehaviorsSanitizer[3]

Target A large set of common Undefined Behaviors, such as out-of-bounds access, divided by zero, and invalid shift.

Example: Out-of-bounds Array Access

```
int main(){
    // This error can also be detected by ASan.
    // But unlike ASan which relies on shadow memory, UBSan detects it by comparing array length and array index.
    char buf[42];
    int bufLen = 50;
    putchar(buf[bufLen]);
}
```

---

[3] https://clang.llvm.org/docs/ UndefinedBehaviorSanitizer.html.

Limitation: Runtime Overhead

The High Runtime Overhead of sanitizers inhibits their adoption in this application scenario!

Table: ASan Performance on Spec CPU 2006 (C/C++)[4]

BENCHMARK	O2	O2+ASan	Slowdown
400.perlbench	344.00	1304.00	3.79
401.bzip2	490.00	844.00	1.72
403.gcc	322.00	608.00	1.89

---

[4] **https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers**.

Redundant Checks

Idea Some checks are redundant!

Example: A Redundant Check

```
for (int i = 1; i <= n; ++i) {
    sum += a[i]; // ASan1, check the postion a+i
    a[i] = -1; // ASan2 (identical to ASan1)
}
```

The Workflow

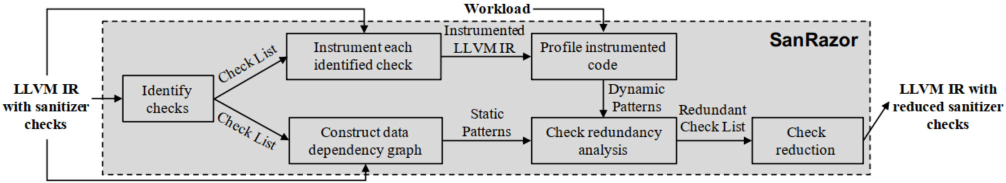


Figure: Workflow of SanRazor

Key Steps:

- Check Identification
- Check Redundancy Analysis (Static Patterns, Dynamic Patterns)
- Check Reduction

Part II

Problem Formulation

4 **Define A Check**

5 Define a Redundant Check

Define a Sanitizer Check

A sanitizer check $c(v)$ (v is the input parameter, usually some critical program information) can be defined as:

Define by a If-Statement

```
if (P(v) does not hold) { // P(v) is a property P w.r.t parameter v
  abort_or_alert(); // detect
}
```

For a sanitizer check c, we use $c.v$ and $c.P$ to represent its input parameter and its property.

Define a Sanitizer Check

A sanitizer check $c(v)$ (v is the input parameter, usually some critical program information) can be defined as:

Define by a If-Statement (in LLVM IR)

```
%o = icmp cond %a, %b
br i1 %o, label %bb1 , label %bb2
```

4 Define A Check

5 Define a Redundant Check

Basic Idea (Natural Language)

Definition

Assume that a sanitizer check c_i that could detect a hypothetical bug B in program p is removed. If B can still be detected, either by another sanitizer check c_j or by a user-defined check, then c_i is a redundant sanitizer check.

In short: removing check A will not effect bug-detection.

More Formally

SHANGHAI JIAO TONG UNIVERSITY

A nontrivial, single-threaded program p with a set of checks $c \in \mathbb{C}$.

Definition

Two checks c_i and c_j are deemed identical when the following condition holds:

$$(c_i \in \mathrm{dom}(c_j) \vee c_j \in \mathrm{dom}(c_i)) \wedge [[c_i.v]] = [[c_j.v]] \wedge c_i.P = c_j.P$$

- $(c_i \in \mathrm{dom}(c_j) \vee c_j \in \mathrm{dom}(c_i))$: a dominating relationship.
- $[[c_i.v]] = [[c_j.v]]$: $c_i.v$ and $c_j.v$ are semantically equivalent.
- $c_i.P = c_j.P$: they are the same type of checks.

More Formally

SHANGHAI JIAO TONG UNIVERSITY

- $(c_i \in \text{dom}(c_j) \vee c_j \in \text{dom}(c_i))$: a dominating relationship. (In some cases it can be challenging to perform the CFG analysis to recover the DomTree.)
- $[[c_i.v]] = [[c_j.v]]$: $c_i.v$ and $c_j.v$ are semantically equivalent (It could be very difficult according to computability theory (e.g. Rice's theorem[5]))
- $c_i.P = c_j.P$: they are the same type of checks. (Easy.)

---

[5] RICE H G. Classes of recursively enumerable sets and their decision problems[J]. Transactions of the American Mathematical Society, 1953, 74: 358-366.

Likely Redundant Checks

A compromise to theoretical challenge: Likely Redundant.

- For dominating analysis, replace the condition by: c_i and c_j have correlated dynamic code coverage patterns.
- For semantical equivalence, check whether $[[c_i.P(c_i.v)]] \approx [[c_j.P(c_j.v)]]$ by static data dependency patterns.

In short: two checks are deemed redundant when they yield identical dynamic and static patterns.

Part III

Design and Implementation

Review: The Workflow

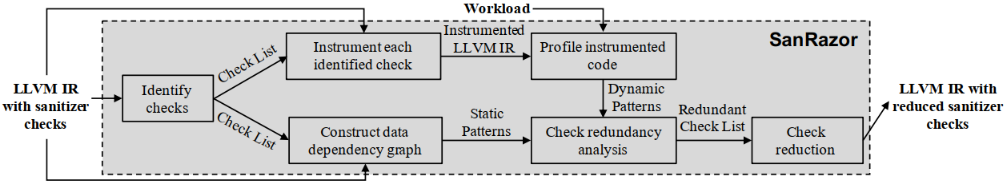


Figure: Workflow of SanRazor

Key Steps:

- Check Identification
- Check Redundancy Analysis (Static Patterns, Dynamic Patterns)
- Check Reduction

6 **Check Identification**

7 Dynamic Check Pattern Capturing

8 Static Check Pattern Capturing

9 Sanitizer Check Reduction

How to find a Check

SHANGHAI JIAO TONG UNIVERSITY

Recall: a sanitizer check is just a if-statement!

Define by a If-Statement (in LLVM IR)

```
    %o = icmp cond %a, %b
    br i1 %o, label %bb1 , label %bb2
```

Distinguish from User Checks

Problem Simply search the icmp is unreasonable because there are many icmp instructions in the source code originally.

Solution Find the sanitizer icmp.

Note that the result of a sanitizer check is abort or alert which is called in bb1 or bb2, we can identify it by check whether there is such call in two BBs.

Distinguish from User Checks

Example: A typical Sanitizer Check

```
    ...
    %o = icmp cond %a, %b
    br i1 %o, label %bb1 , label %bb2
    ...
  bb1:
    <normal code>
    ...
  bb2:
    call _ASan_handle_XXX
```

Dynamic Check Pattern Capturing

SHANGHAI JIAO TONG UNIVERSITY

Idea Each SC is a if-statement. Use three counters to recod how many times:

- the branch instruction
- the true branch
- the false branch

is executed.

Comparing Dynamic Coverage Patterns

For each sanitizer check sc_i, its dynamic pattern is a tuple $< sb_i, stb_i, sfb_i >$. (And similarly user check uc_i: $< ub_i, utb_i, ufb_i >$)

Check whether two sanitizer checks sc_i, sc_j are identical:

$$(sb_i = sb_j) \wedge ((stb_i = stb_j) \vee (stb_i = sfb_j))$$

Check whether a SC sc_i and a UC uc_i are identical: (if they satisfy one of the following conditions)

$$(sb_i = ub_j) \wedge ((stb_i = stb_j) \vee (stb_i = sfb_j))$$

$$(sb_i = utb_j) \wedge ((stb_i = sb_j) \vee (sfb_i = sb_j))$$

$$(sb_i = ufb_j) \wedge ((stb_i = sb_j) \vee (sfb_i = sb_j))$$

Question the dominating cases of two SCs?

Comparing Dynamic Coverage Patterns

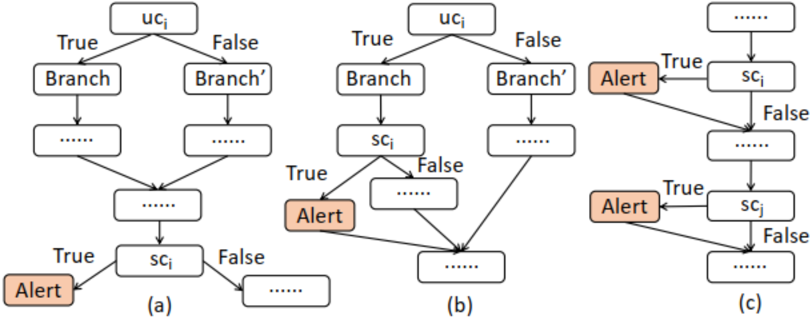


Figure: Coverage patterns

6 Check Identification

7 Dynamic Check Pattern Capturing

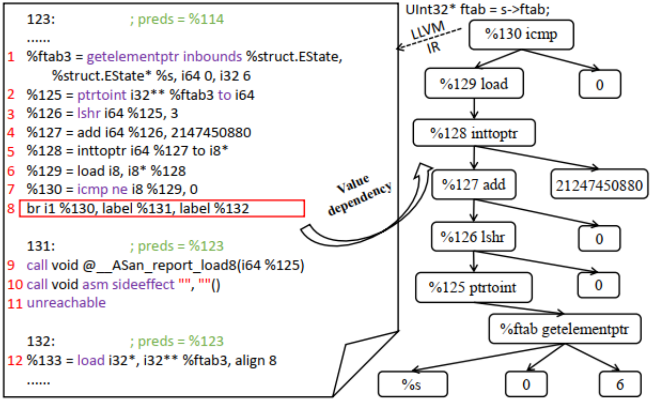
8 Static Check Pattern Capturing

9 Sanitizer Check Reduction

Static Check Pattern Capturing

Idea Perform backward-dependency analysis to construct the data dependency graph. Start from the condition operand of the br instruction.

Backward-dependency Analysis



```
123:                ; preds = %114
......
%ftab3 = getelementptr inbounds %struct.EState,
      %struct.EState* %s, i64 0, i32 6
%125 = ptrtoint i32** %ftab3 to i64
%126 = lshr i64 %125, 3
%127 = add i64 %126, 2147450880
%128 = inttoptr i64 %127 to i8*
%129 = load i8, i8* %128
%130 = icmp ne i8 %129, 0
br i1 %130, label %131, label %132

131:                ; preds = %123
call void @__ASan_report_load8(i64 %125)
call void asm sideeffect "", ""()
unreachable

132:                ; preds = %123
%133 = load i32*, i32** %ftab3, align 8
......
```

Figure: Example: Backward-dependency Analysis

Three Schemes

- L0 gathers all the leaf nodes on the dependency tree into a set.
- L1 which canonicalizes the collected set of leaf nodes, by eliminating all constants from the set except constant operands from the icmp instruction associated with each sanitizer or user check.
- L2 which canonicalizes the collected set of leaf nodes, by eliminating all constants from the set.

Note: the set represents the static pattern of a SC.

Three Schemes: Security Consideration

Question The aggressive scheme causes false positive (i.e. two SCs are deemed identical by our method, but indeed not.)

Example: A Security Consideration

```
    int a = *ptr; // ASan check on ptr
    int b = *(ptr + 4); // ASan check on (ptr +4)
```

Comparing Static Coverage Patterns

Directly compare the set. c_i and c_j are identical iff S_i and S_j (the set generated by backward-dependency analysis) are identical.

6 Check Identification

7 Dynamic Check Pattern Capturing

8 Static Check Pattern Capturing

9 Sanitizer Check Reduction

Sanitizer Check Reduction

- SanRazor does not remove user-defined checks (UC).
- If two sanitizer c_i and c_j are likely redundant, remove the dominator. (But it is just the default setting. User can also configure it to decide which one to remove.)
- In detail, SanRazor sets the condition as false, and just let the Dead-Code-Elimination to remove the branch.

Part IV

Evaluation

10 Cost Study

11 Vulnerability Detectability Study

Environment

- Benchmark: SPEC CPU2006[6]. (contains 19 C/C++ programs)
- Testcases: 401.bzip2, 429.mcf, 445.gobmk, 456.hmmer, 458.sjeng, 462.libquantum, 433.milc, 444.namd, 470.lbm, 482.sphinx3, and 453.povray.
- SanRazor with Clang compiler version 9.0.0.

---

[6] SPRADLING C D. SPEC CPU2006 Benchmark Tools[J]. SIGARCH Comput. Archit. News, 2007, 35(1): 130-134. DOI: 10.1145/1241601.1241625.

Metrics

SHANGHAI JIAO TONG UNIVERSITY

- M_0: the execution time reduction after eliminating redundant checks.
- M_1: the number of removed sanitizer checks.
- M_2: the execution cost (in terms of CPU cycles) saved by reducing sanitizer checks.

Comparison results w.r.t. M_0 metrics

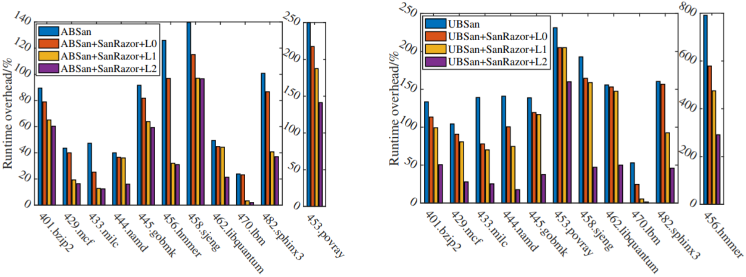
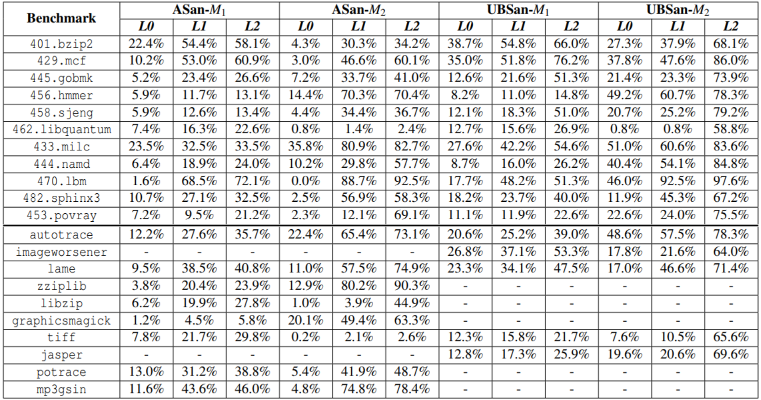


Figure: Left: ASan, Right: UBSan

Cost Evaluation Results



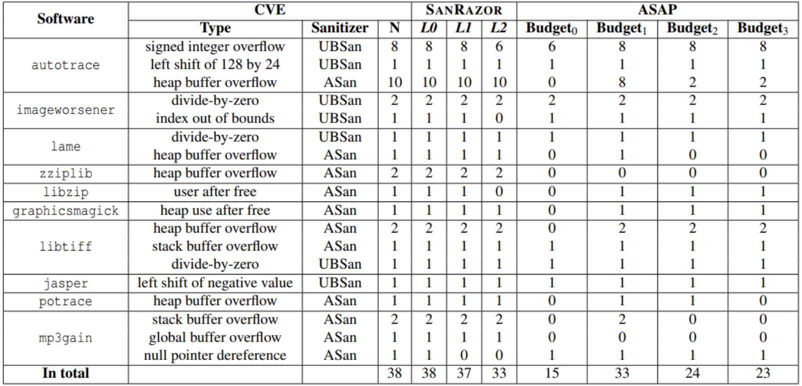
Benchmark	ASan-M_1			ASan-M_2			UBSan-M_1			UBSan-M_2		
	L0	*L1*	*L2*	*L0*	*L1*	*L2*	*L0*	*L1*	*L2*	*L0*	*L1*	*L2*
401.bzip2	22.4%	54.4%	58.1%	4.3%	30.3%	34.2%	38.7%	54.8%	66.0%	27.3%	37.9%	68.1%
429.mcf	10.2%	53.0%	60.9%	3.0%	46.6%	60.1%	35.0%	51.8%	76.2%	37.8%	47.6%	86.0%
445.gobmk	5.2%	23.4%	26.6%	7.2%	33.7%	41.0%	12.6%	21.6%	51.3%	21.4%	23.3%	73.9%
456.hmmer	5.9%	11.7%	13.1%	14.4%	70.3%	70.4%	8.2%	11.0%	14.8%	49.2%	60.7%	78.3%
458.sjeng	5.9%	12.6%	13.4%	4.4%	34.4%	36.7%	12.1%	18.3%	51.0%	20.7%	25.2%	79.2%
462.libquantum	7.4%	16.3%	22.6%	0.8%	1.4%	2.4%	12.7%	15.6%	26.9%	0.8%	0.8%	58.8%
433.milc	23.5%	32.5%	33.5%	35.8%	80.9%	82.7%	27.6%	42.2%	54.6%	51.0%	60.6%	83.6%
444.namd	6.4%	18.9%	24.0%	10.2%	29.8%	57.7%	8.7%	16.0%	26.2%	40.4%	54.1%	84.8%
470.lbm	1.6%	68.5%	72.1%	0.0%	88.7%	92.5%	17.7%	48.2%	51.3%	46.0%	92.5%	97.6%
482.sphinx3	10.7%	27.1%	32.5%	2.5%	56.9%	58.3%	18.2%	23.7%	40.0%	11.9%	45.3%	67.2%
453.povray	7.2%	9.5%	21.2%	2.3%	12.1%	69.1%	11.1%	11.9%	22.6%	22.6%	24.0%	75.5%
autotrace	12.2%	27.6%	35.7%	22.4%	65.4%	73.1%	20.6%	25.2%	39.0%	48.6%	57.5%	78.3%
imageworsener	-	-	-	-	-	-	26.8%	37.1%	53.3%	17.8%	21.6%	64.0%
lame	9.5%	38.5%	40.8%	11.0%	57.5%	74.9%	23.3%	34.1%	47.5%	17.0%	46.6%	71.4%
zziplib	3.8%	20.4%	23.9%	12.9%	80.2%	90.3%	-	-	-	-	-	-
libzip	6.2%	19.9%	27.8%	1.0%	3.9%	44.9%	-	-	-	-	-	-
graphicsmagick	1.2%	4.5%	5.8%	20.1%	49.4%	63.3%	-	-	-	-	-	-
tiff	7.8%	21.7%	29.8%	0.2%	2.1%	2.6%	12.3%	15.8%	21.7%	7.6%	10.5%	65.6%
jasper	-	-	-	-	-	-	12.8%	17.3%	25.9%	19.6%	20.6%	69.6%
potrace	13.0%	31.2%	38.8%	5.4%	41.9%	48.7%	-	-	-	-	-	-
mp3gsin	11.6%	43.6%	46.0%	4.8%	74.8%	78.4%	-	-	-	-	-	-

Figure: Evaluation results w.r.t. M_1 and M_2

10 Cost Study

11 Vulnerability Detectability Study

Vulnerability Detectability Study



Software	CVE			SanRazor			ASAP			
	Type	Sanitizer	N	*L0*	*L1*	*L2*	Budget$_0$	Budget$_1$	Budget$_2$	Budget$_3$
`autotrace`	signed integer overflow	UBSan	8	8	8	6	6	8	8	8
	left shift of 128 by 24	UBSan	1	1	1	1	1	1	1	1
	heap buffer overflow	ASan	10	10	10	10	0	8	2	2
`imageworsener`	divide-by-zero	UBSan	2	2	2	2	2	2	2	2
	index out of bounds	UBSan	1	1	1	0	1	1	1	1
`lame`	divide-by-zero	UBSan	1	1	1	1	1	1	1	1
	heap buffer overflow	ASan	1	1	1	1	0	1	0	0
`zziplib`	heap buffer overflow	ASan	2	2	2	2	0	0	0	0
`libzip`	user after free	ASan	1	1	1	0	0	1	1	1
`graphicsmagick`	heap use after free	ASan	1	1	1	1	0	1	1	1
`libtiff`	heap buffer overflow	ASan	2	2	2	2	0	2	2	2
	stack buffer overflow	ASan	1	1	1	1	1	1	1	1
	divide-by-zero	UBSan	1	1	1	1	1	1	1	1
`jasper`	left shift of negative value	UBSan	1	1	1	1	1	1	1	1
`potrace`	heap buffer overflow	ASan	1	1	1	1	0	1	1	0
`mp3gain`	stack buffer overflow	ASan	2	2	2	2	0	2	0	0
	global buffer overflow	ASan	1	1	1	1	0	0	0	0
	null pointer dereference	ASan	1	1	0	0	1	1	1	1
In total			38	38	37	33	15	33	24	23

Figure: CVE case study comparing with ASAP[7]

Part V

Discussion

Characteristics of Removed Checks False Positive Analysis False Negative Analysis Effects of Workload Selection

Characteristics of Removed Checks

Type 1 Checks that are identical with other checks.

Example: the code snippet in bzip2.c

```
void BZ_blockSort (EState* s) {
  UInt32* ptr = s->ptr; // UBSan1: check whether s is nullptr
  UChar* block = s->block; // UBSan2: check whether s is nullptr
}
```

Characteristics of Removed Checks

Type 2 Checks that are strongly correlated.

Example: CVE-2017-9169

```
    *(temp ++)= buffer[xpos * 3 + 2]; // ASan1
    *(temp ++)= buffer[xpos * 3 + 1]; // ASan2
    *(temp ++)= buffer[xpos * 3]; // ASan3
```

False Positive Analysis

SHANGHAI JIAO TONG UNIVERSITY

False Positive cases are caused mainly by:

- The captured dynamic patterns only provide statistical information of sanitizer checks
- The static pattern sets could be optimistic (L1, L2 scheme)

Example: CVE-2017-12858

```
    void _zip_buffer_free (zip_buffer_t *buffer){
      if (buffer == NULL) return; // UserCheck
      if(buffer->free_data){ // CVE. An ASan inserted here.
        free(buffer->data);
      ...
```

False Negative Analysis

SanRazor also has some False Negative (i.e.Redundant checks but not removed) because our compromise (likely redundant).

Example: piece of code in 462.libquantum

```
for(i=0; i<reg->size; i++) {
  if(reg ! node[i].state & ...) {
    if(reg ! node[i].state & ...) {
```

Effects of Workload Selection

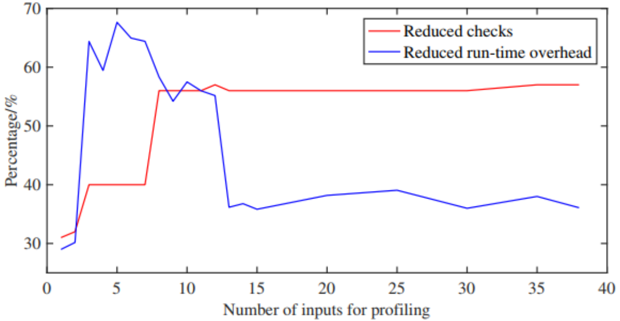


Figure: Effects of Workload Selection Evaluation on ASan, profiling bzip2

Part VI

Conclusion

- SanRazor effectively lower the overhead but still retaining high vulnerability detection capability.
- It is important to abstract and formulate problem properly.
- Identical (or likely identical) analysis is crucial in reduction.
- Futher work: Some methods to reduce false positive and false negative. (may be a better domination analysis)

Part VII

References

References

[1] ZHANG J, WANG S, RIGGER M, et al. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs[C/OL]//15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). [S.l.]: USENIX Association, 2021: 479-494. https://www.usenix.org/conference/osdi21/presentation/zhang.

[2] SEREBRYANY K, BRUENING D, POTAPENKO A, et al. AddressSanitizer: A Fast Address Sanity Checker[C]//2012 USENIX Annual Technical Conference (USENIX ATC 12). Boston, MA: USENIX Association, 2012: 309-318.

[3] RICE H G. Classes of recursively enumerable sets and their decision problems[J]. Transactions of the American Mathematical Society, 1953, 74: 358-366.

[4] SPRADLING C D. SPEC CPU2006 Benchmark Tools[J]. SIGARCH Comput. Archit. News, 2007, 35(1): 130-134. DOI: 10.1145/1241601.1241625.

[5] WAGNER J, KUZNETSOV V, CANDEA G, et al. High System-Code Security with Low Overhead[C]//2015 IEEE Symposium on Security and Privacy. [S.l. : s.n.], 2015: 866-879. DOI: 10.1109/SP.2015.58.

SHANGHAI JIAO TONG UNIVERSITY

Thank You

Chaofan Lin · SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs