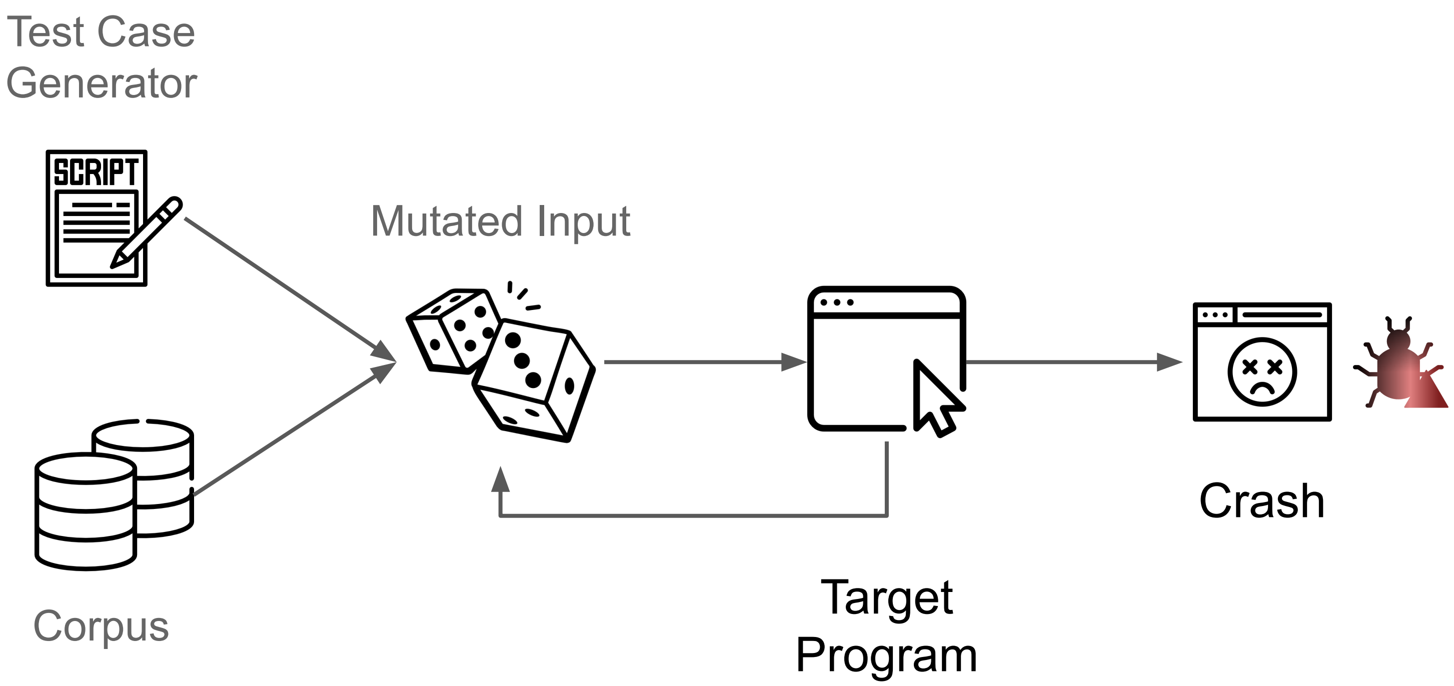
MTSan: A Feasible and Practical Memory Sanitizer for Fuzzing COTS Binaries

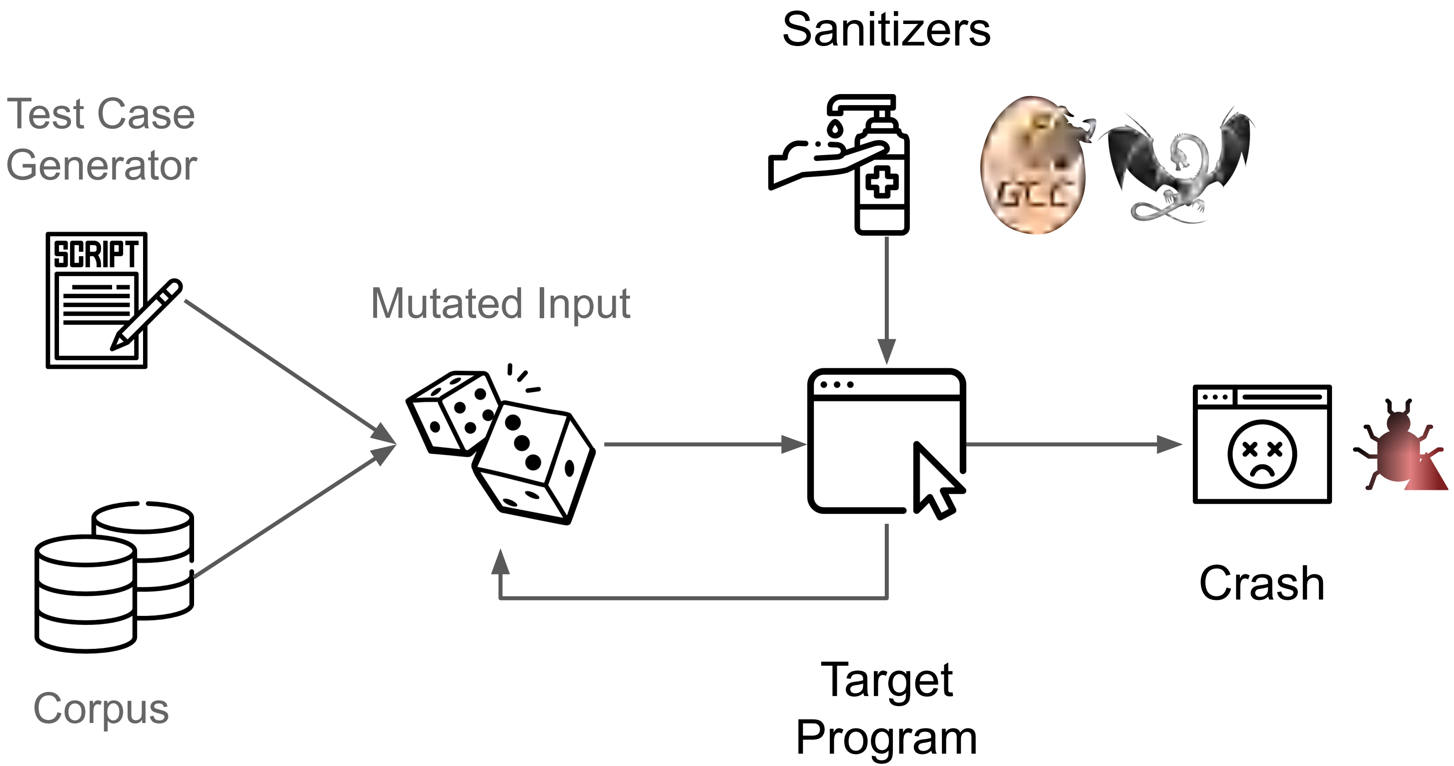
Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, **Ruoyu Wang**, Haixin Duan, Haoyu Wang, Chao Zhang*

中国科学院信息工程研究所
INSTITUTE OF INFORMATION ENGINEERING,CAS

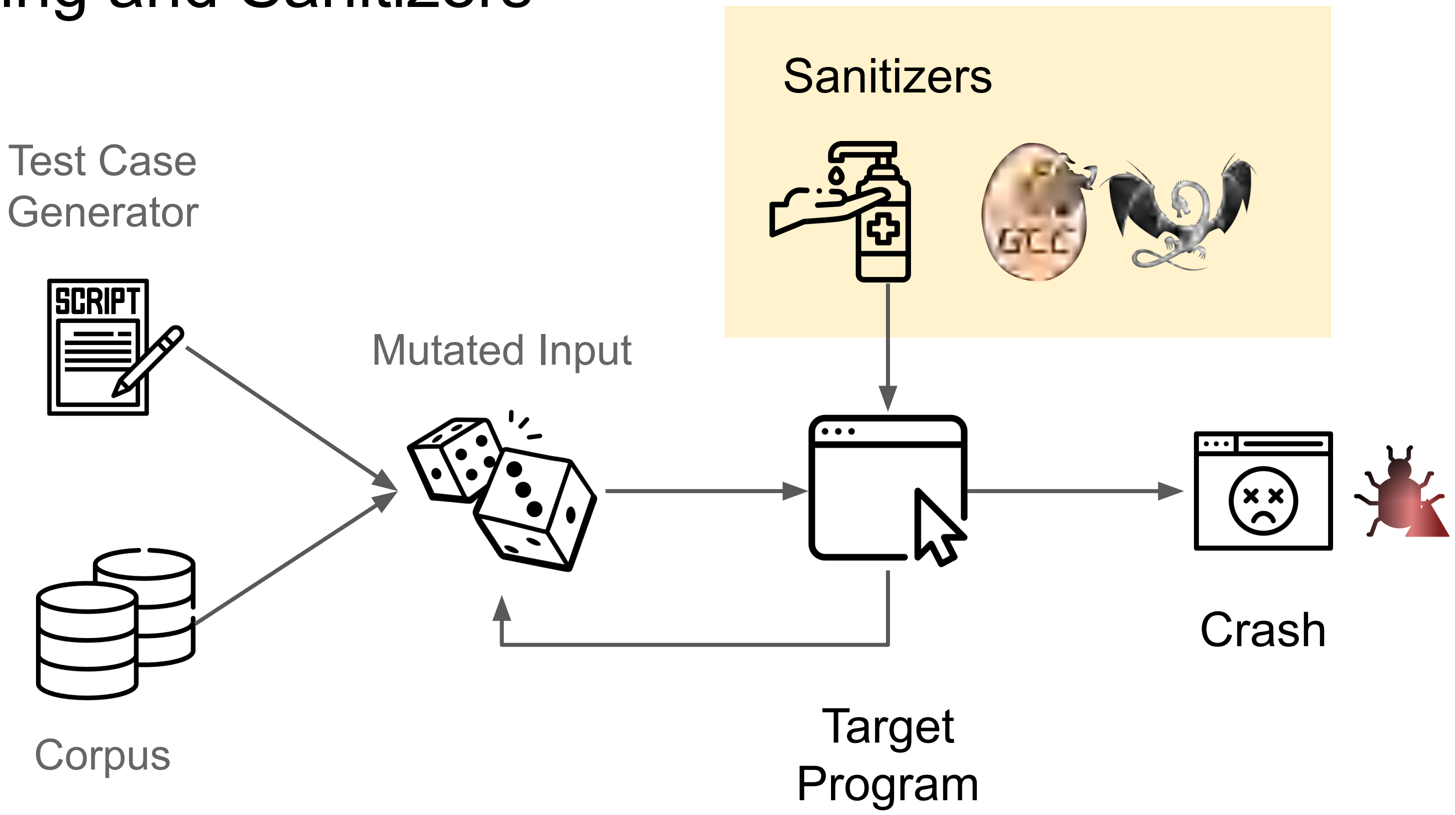
Fuzzing and Sanitizers



Fuzzing and Sanitizers

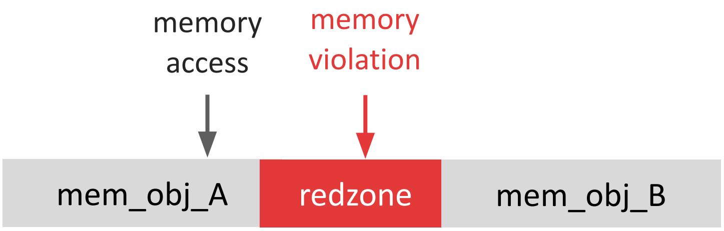


Fuzzing and Sanitizers



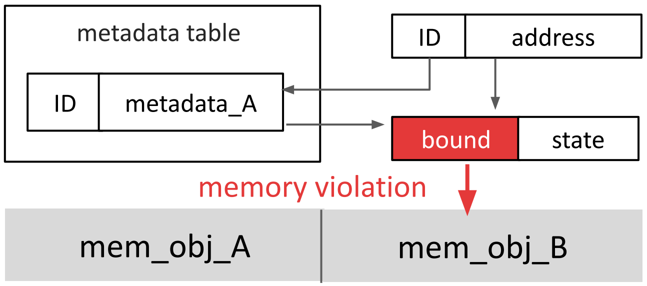
Sanitizers and Memory Safety Violations

- Detects spatial and temporal violation
- E.g., AddressSanitizer (ASan)
  - Location-based (redzones)
    - *Purify, Oscar,* etc.



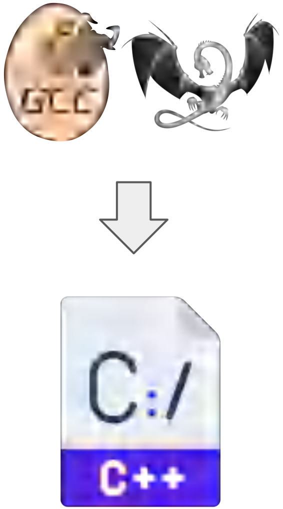
Sanitizers and Memory Safety Violations

- Detects spatial and temporal violation
- E.g., AddressSanitizer (ASan)
  - Location-based (redzones)
    - *Purify, Oscar,* etc.
- E.g., PacMem
  - Identity-based (metadata)
    - *SoftBound+CETS, Low-fat Pointer*, etc



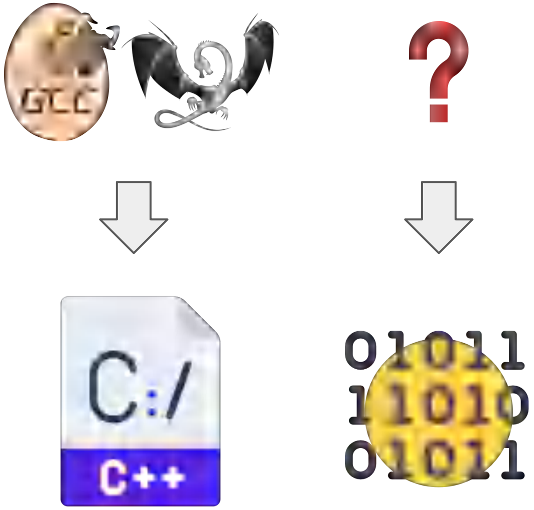
Sanitizers and Memory Safety Violations

- Detects spatial and temporal violation
- E.g., AddressSanitizer (ASan)
    - Location-based (redzones)
        - *Purify, Oscar,* etc.
- E.g., PacMem
    - Identity-based (metadata)
        - *SoftBound+CETS, Low-fat Pointer*, etc

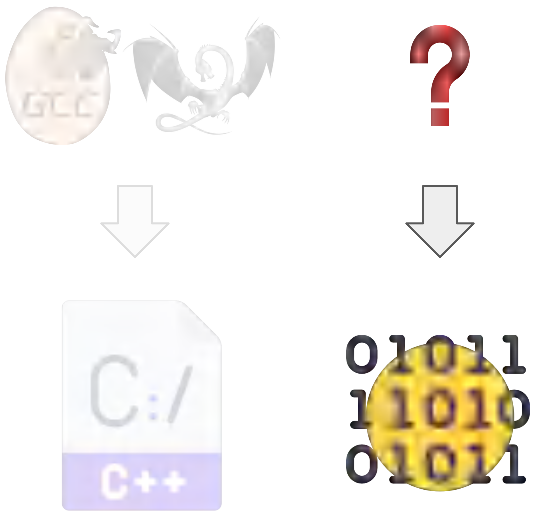


Sanitizers and Memory Safety Violations

- Detects spatial and temporal violation
- E.g., AddressSanitizer (ASan)
  - Location-based (redzones)
    - *Purify, Oscar,* etc.
- E.g., PacMem
  - Identity-based (metadata)
    - *SoftBound+CETS, Low-fat Pointer*, etc



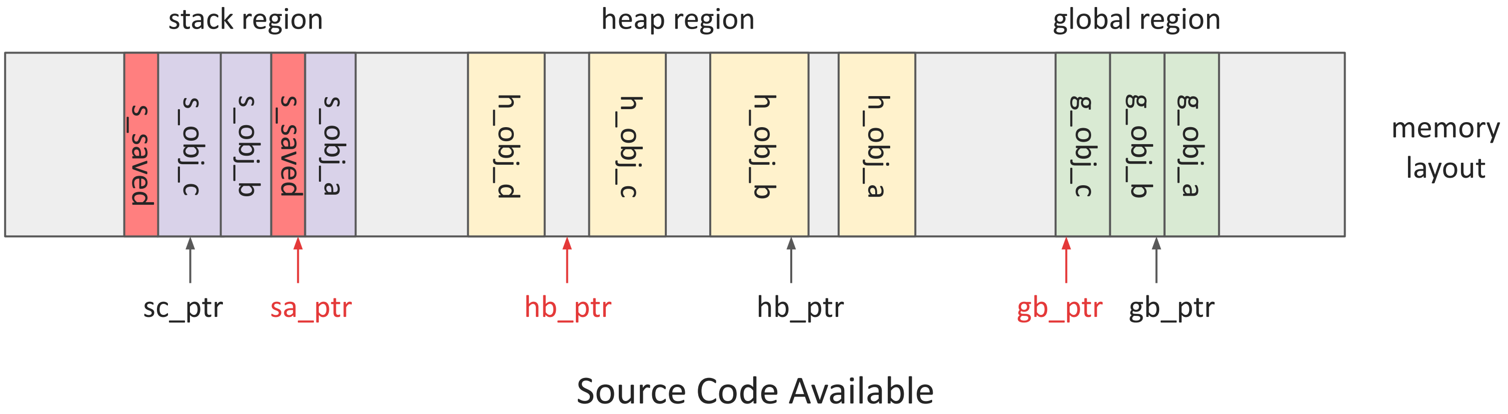
Binary Sanitizers



- Undangle [ISSTA'12]
- Dr. Memory [CGO'11]
- Memcheck [ATC'05]
- QASan [SecDev'20]
- ASan-Retrowrite [S&P'20]

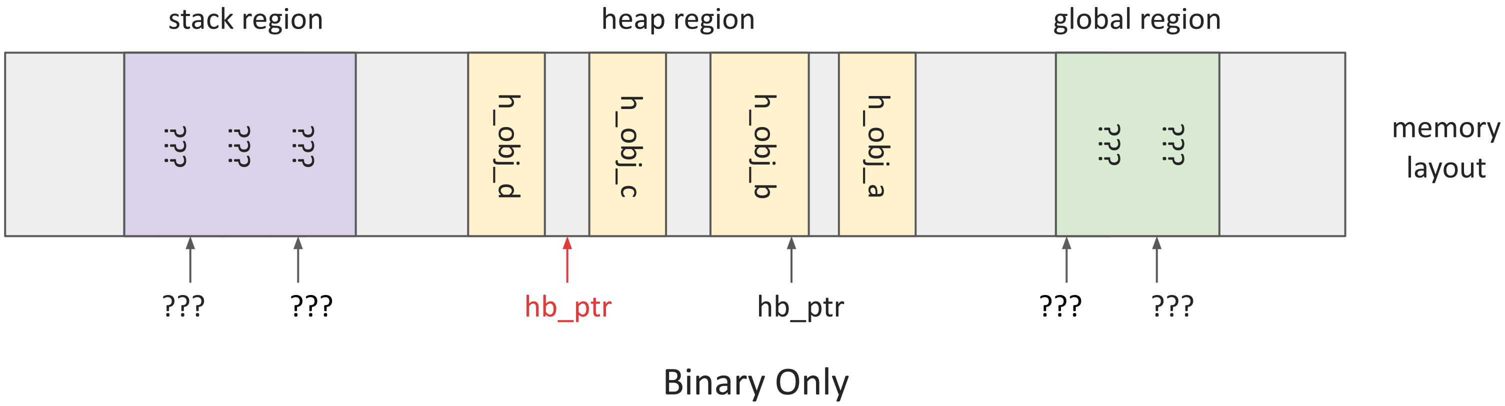
Limitations of Existing Binary Sanitizers

1. They only support heap objects, neglecting memory errors in stack and global regions.



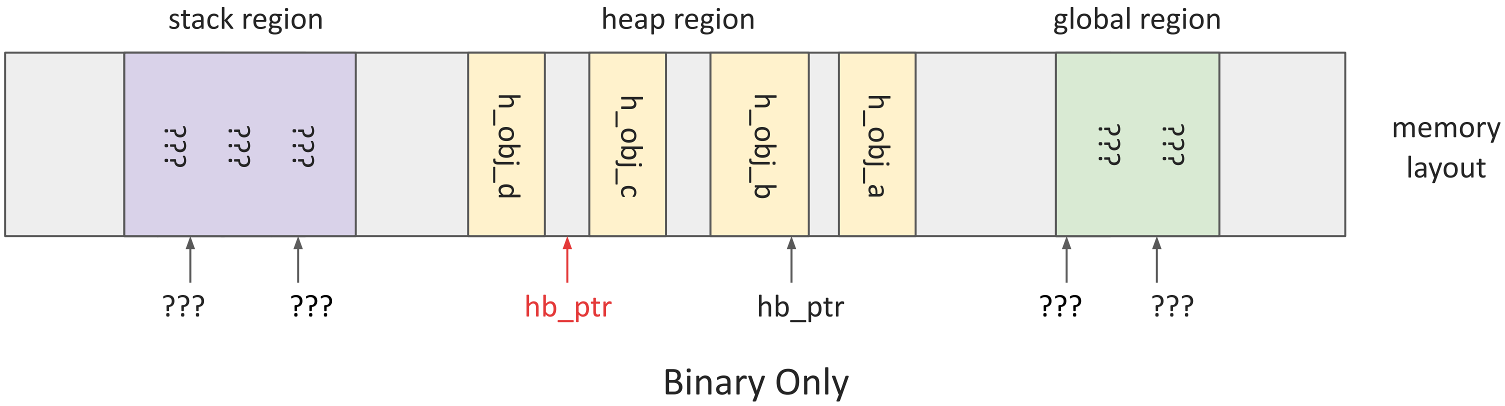
Limitations of Existing Binary Sanitizers

1. They only support heap objects, neglecting memory errors in stack and global regions.



Limitations of Existing Binary Sanitizers

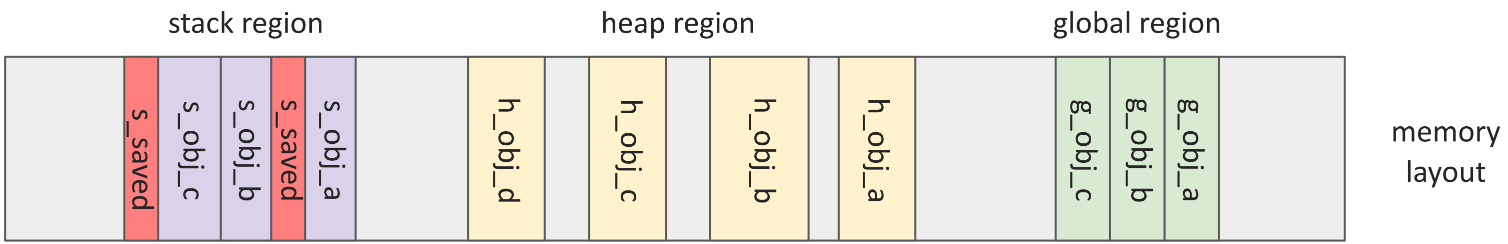
1. They only support heap objects, neglecting memory errors in stack and global regions.



Type info is lost during compilation -> **boundary info is unavailable**

Limitations of Existing Binary Sanitizers

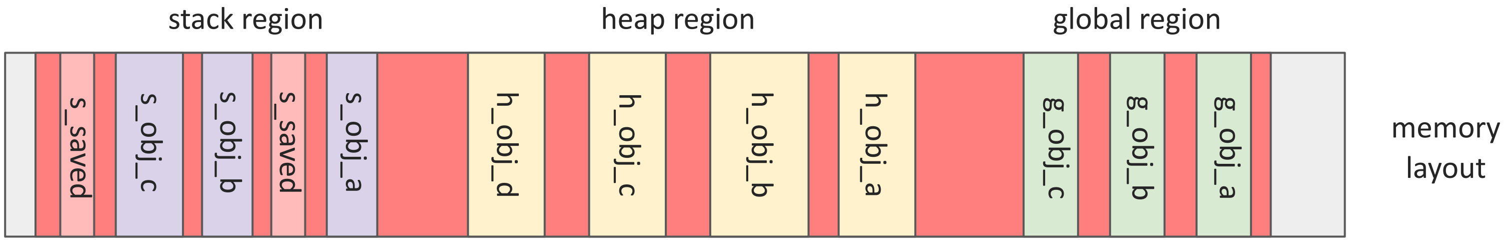
2. Redzone-based approaches do not apply on binaries



Source Code Available (w/o redzone)

Limitations of Existing Binary Sanitizers

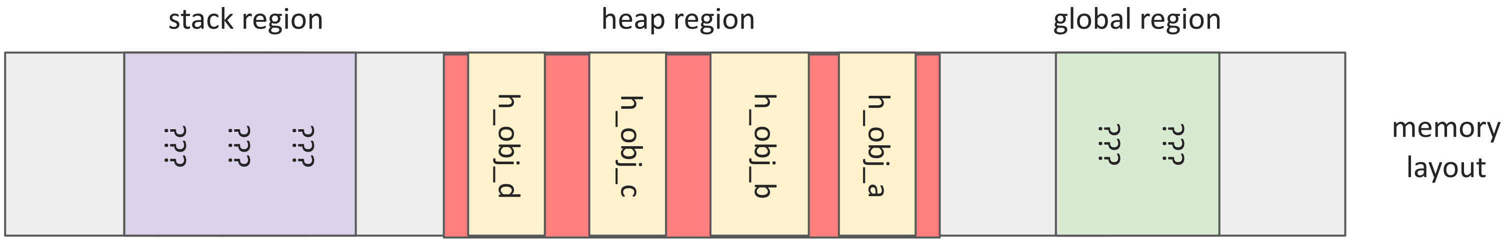
2. Redzone-based approaches do not apply on binaries



Source Code Available (w/redzone)

Limitations of Existing Binary Sanitizers

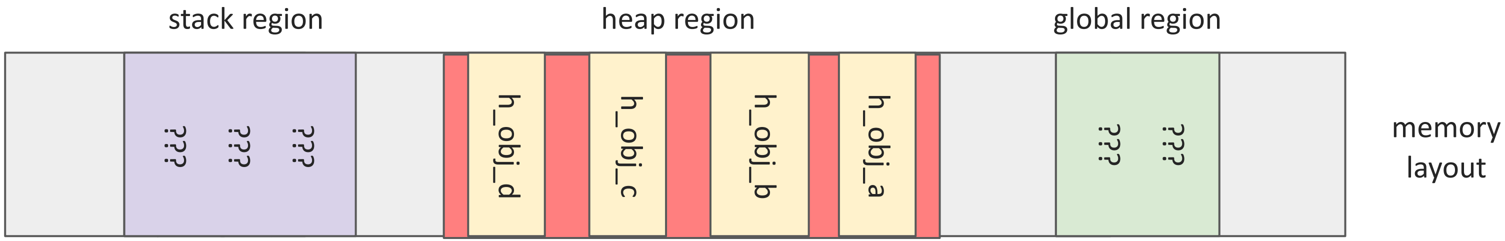
2. Redzone-based approaches do not apply on binaries



Binary Only (w/redzone)

Limitations of Existing Binary Sanitizers

2. Redzone-based approaches do not apply on binaries



Binary Only (w/redzone)

Cannot add redzones without **changing memory layouts**

Limitations of Existing Binary Sanitizers

3. High runtime and memory overhead

Binary Sanitizer	Bug-finding Techs	Object Coverage			Runtime Overhead*	Memory Overhead*
		Heap	Stack	Global		
Undangle	pointer-tracking**	yes	no	no	>10x	>10x
Dr. Memory	redzone	yes	no	no	>10x	>10x
Memcheck	redzone	yes	no	no	>10x	3-10x
QASan	redzone	yes	no	no	>10x	3-10x
ASan-Retrowrite	redzone	yes	no	no	1-3x	3-10x

* Standalone execution, with no optimization applied.
** Use-after-free violation only.

Limitations of Existing Binary Sanitizers

3. High runtime and memory overhead

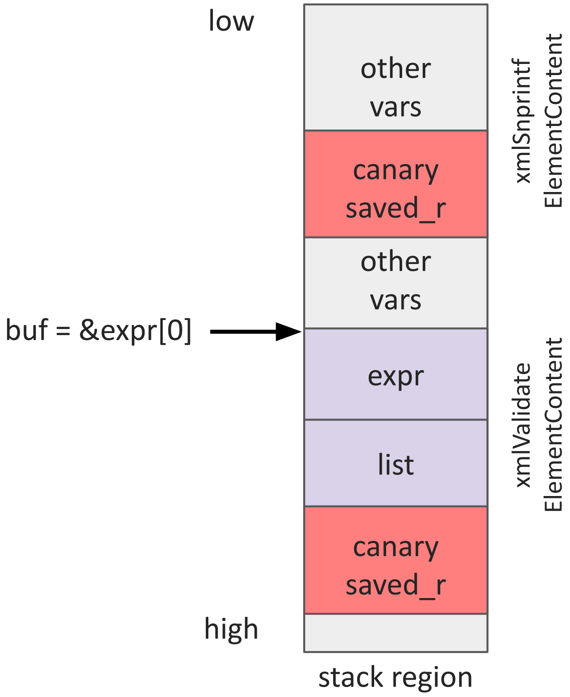
Binary Sanitizer	Bug-finding Techs	Object Coverage			Runtime Overhead*	Memory Overhead*
		Heap	Stack	Global		
Undangle	pointer-tracking**	yes	no	no	>10x	>10x
Dr. Memory	redzone	yes	no	no	>10x	>10x
Memcheck	redzone	yes	no	no	>10x	3-10x
QASan	redzone	yes	no	no	>10x	3-10x
ASan-Retrowrite	redzone	yes	no	no	1-3x	3-10x

High overhead **reduces fuzzing efficiency** and **curtails their application**

* Standalone execution, with no optimization applied.
** Use-after-free violation only.

Motivating Example

CVE-2017-9047

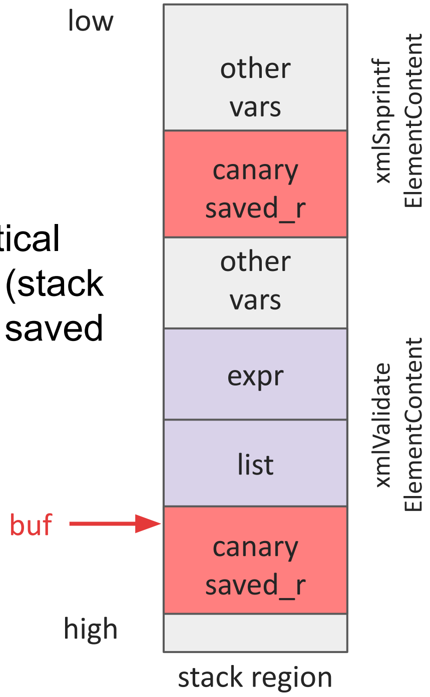



```c
void xmlSnprintfElementContent(char *buf, int size,
    xmlElementContentPtr content, int englob) {
    /* ... */
    len = strlen(buf);
        /* ... */
        if (content->prefix != NULL) {
            if (size - len < xmlStrlen(content->prefix) + 10) {
                strcat(buf, " ...");
                return;
            }
            strcat(buf, (char *) content->prefix);
            strcat(buf, ":");
        }
        if (size - len < xmlStrlen(content->name) + 10) {
            strcat(buf, " ...");
            return;
        }
        if (content->name != NULL)
            strcat(buf, (char *) content->name);
        /* ... */
}
int xmlValidateElementContent(xmlValidCtxtPtr ctxt, xmlNodePtr
    child, xmlElementPtr elemDecl, int warn, xmlNodePtr parent){
    /* ... */
    if (ctxt != NULL) {
        char expr[5000]; // vulnerable buffer
        char list[5000]; // victim buffer
        expr[0] = 0;
        xmlSnprintfElementContent(&expr[0], 5000, cont, 1);
    /* ... */
}
```

Motivating Example

CVE-2017-9047

Overflowing critical data structures (stack canary and the saved return address)

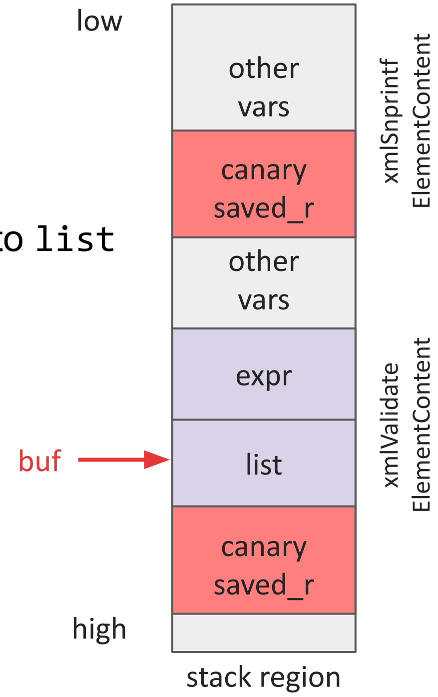



```c
void xmlSnprintfElementContent(char *buf, int size,
    xmlElementContentPtr content, int englob) {
    /* ... */
    len = strlen(buf);
        /* ... */
        if (content->prefix != NULL) {
            if (size - len < xmlStrlen(content->prefix) + 10) {
                strcat(buf, " ...");
                return;
            }
            strcat(buf, (char *) content->prefix);
            strcat(buf, ":");
        }
        if (size - len < xmlStrlen(content->name) + 10) {
            strcat(buf, " ...");
            return;
        }
        if (content->name != NULL)
            strcat(buf, (char *) content->name);
        /* ... */
}
int xmlValidateElementContent(xmlValidCtxtPtr ctxt, xmlNodePtr
    child, xmlElementPtr elemDecl, int warn, xmlNodePtr parent){
    /* ... */
    if (ctxt != NULL) {
        char expr[5000];  // vulnerable buffer
        char list[5000];  // victim buffer
        expr[0] = 0;
        xmlSnprintfElementContent(&expr[0], 5000, cont, 1);
    /* ... */
}
```

Motivating Example

CVE-2017-9047

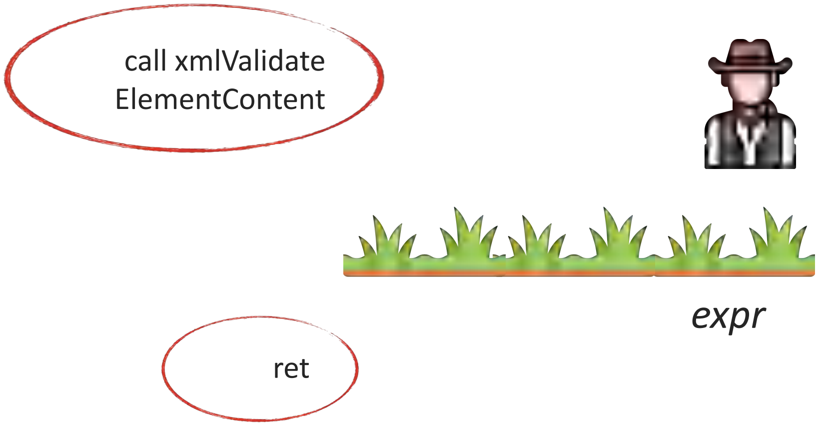
Overflowing into `list`




```c
void xmlSnprintfElementContent(char *buf, int size,
    xmlElementContentPtr content, int englob) {
    /* ... */
    len = strlen(buf);
        /* ... */
        if (content->prefix != NULL) {
            if (size - len < xmlStrlen(content->prefix) + 10) {
                strcat(buf, " ...");
                return;
            }
            strcat(buf, (char *) content->prefix);
            strcat(buf, ":");
        }
        if (size - len < xmlStrlen(content->name) + 10) {
            strcat(buf, " ...");
            return;
        }
        if (content->name != NULL)
            strcat(buf, (char *) content->name);
        /* ... */
}
int xmlValidateElementContent(xmlValidCtxtPtr ctxt, xmlNodePtr
    child, xmlElementPtr elemDecl, int warn, xmlNodePtr parent){
    /* ... */
    if (ctxt != NULL) {
        char expr[5000];  // vulnerable buffer
        char list[5000];  // victim buffer
        expr[0] = 0;
        xmlSnprintfElementContent(&expr[0], 5000, cont, 1);
    /* ... */
}
```

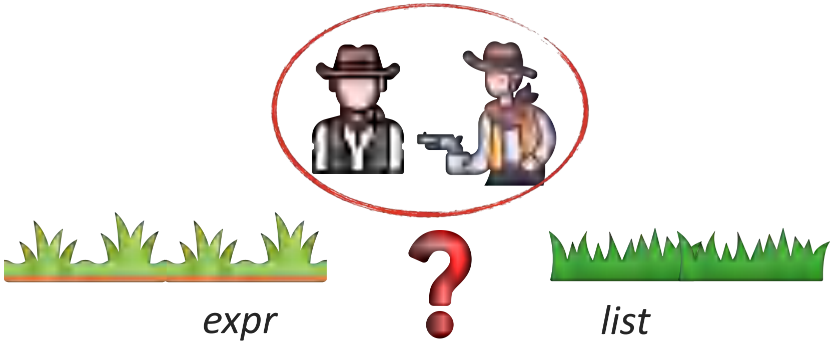
Challenges

1. How to recover memory objects in target binary?
   a. pointers
   b. boundary
   c. lifetime



Challenges

1. How to recover memory objects in target binary?
    a. pointers
    b. boundary
    c. lifetime
2. How to detect memory violations?



Our Initution

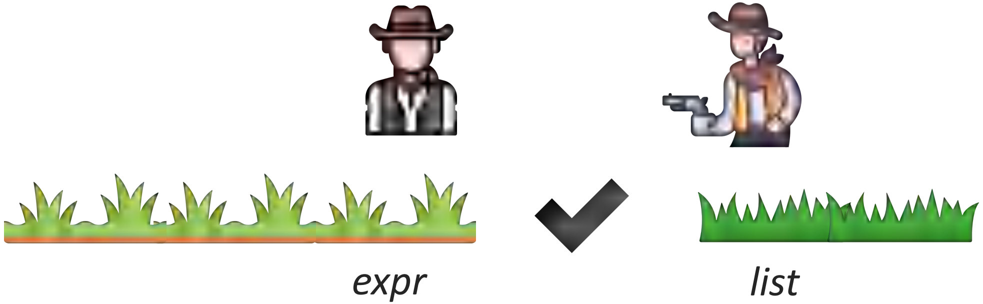
- Access pattern helps to infer data structures in memory
  - Rewards(NDSS'10), Howard(NDSS'11)

expr

list

Our Initution

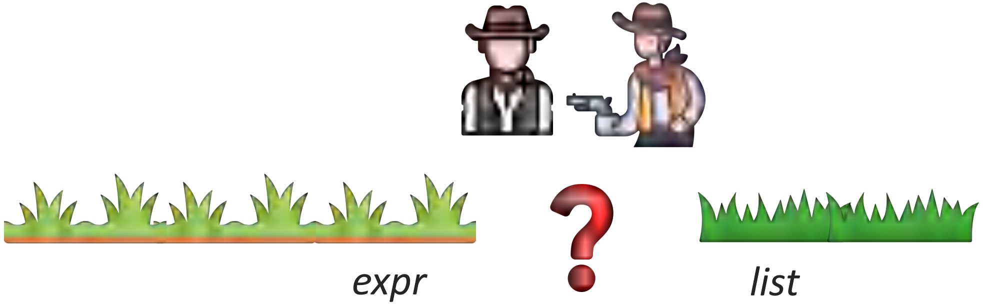
- Access pattern helps to infer data structures in memory
  - Rewards(NDSS'10), Howard(NDSS'11)



Our Initution

- Access pattern helps to infer data structures in memory
  - Rewards(NDSS'10), Howard(NDSS'11)
- Our insight

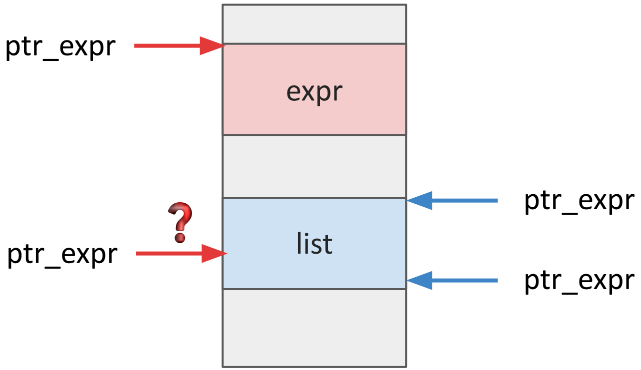
"***Conflicts*** *among inferred object boundaries —— caused by inferencing from both benign and bug-triggering input —— are* ***indicators for memory errors***"



Our Initution

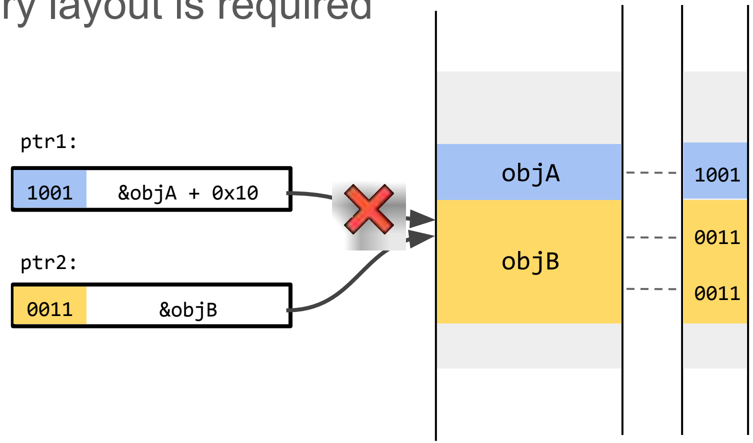
- Access pattern helps to infer data structures in memory
  - Rewards(NDSS'10), Howard(NDSS'11)
- Our insight

"***Conflicts*** *among inferred object boundaries —— caused by inferencing from both benign and bug-triggering input —— are* ***indicators for memory errors***"



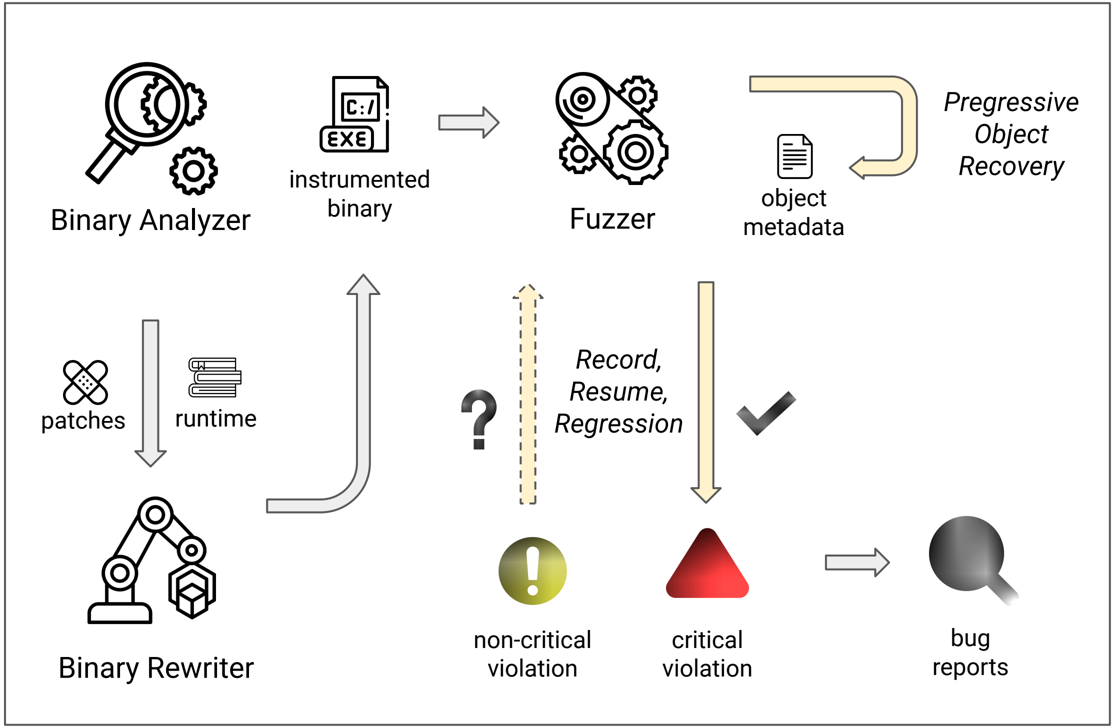
Memory Tagging

- Add unique tags to both pointers and memory space
- Checked at every memory access by hardware and crashes the program if not match
- **No change** to memory layout is required

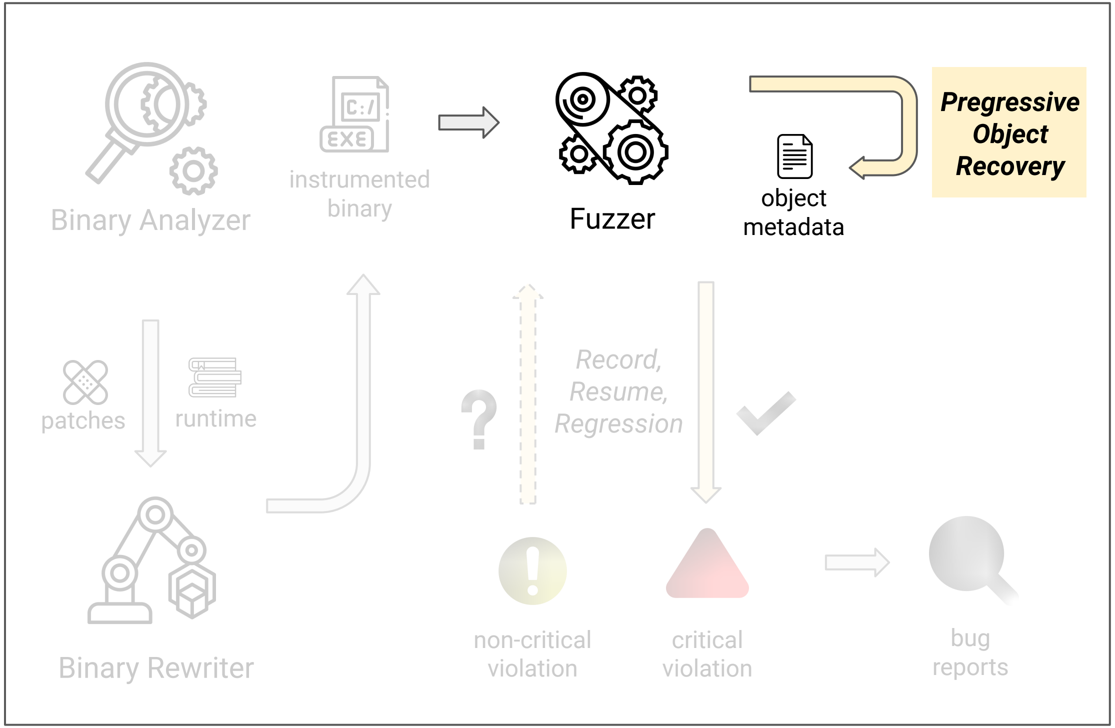


- 64-bit architectures only
- Every aligned 16 bytes of memory have a 4-bit tag
- ARM introduced Memory Tagging Extension in **ARMv8.5-A**

Our Approach: MTSan

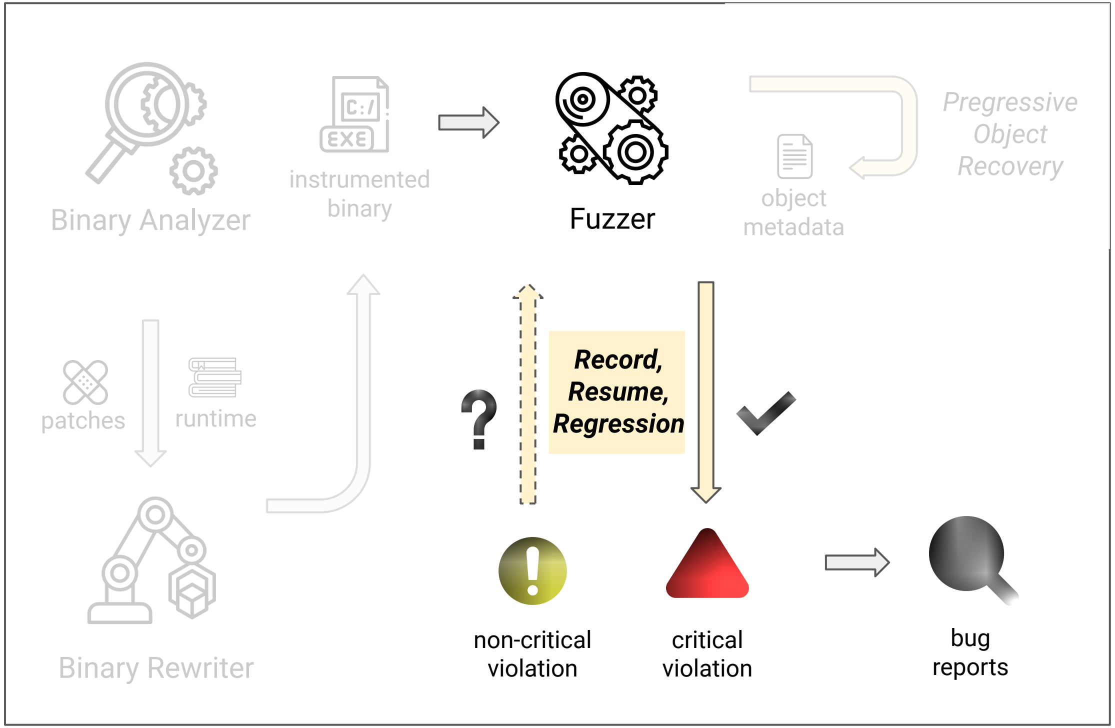


Our Approach: MTSan



Challenge 1. **Recovering memory objects** during fuzzing

Our Approach: MTSan

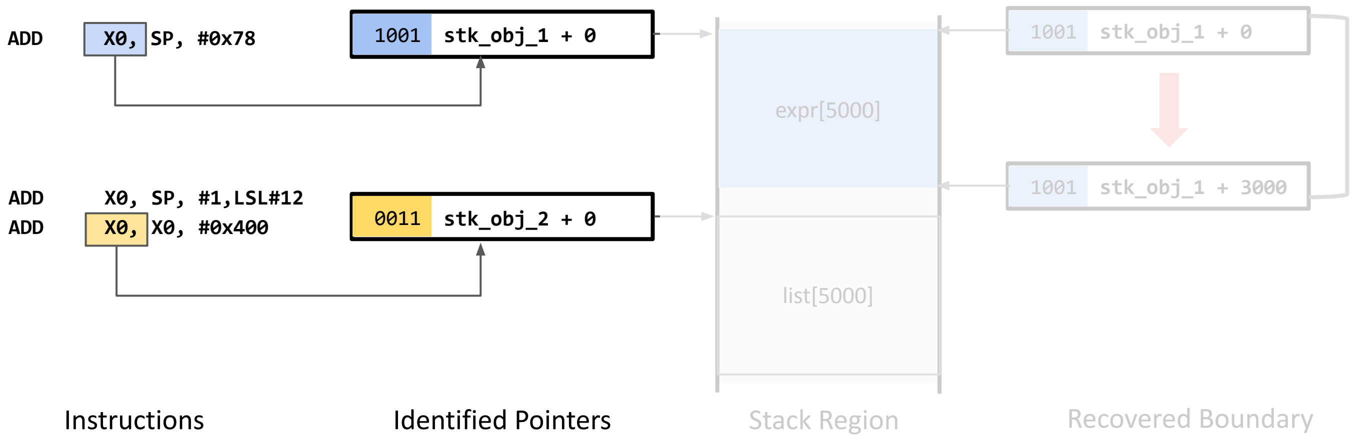


Challenge 1. **Recovering memory objects** during fuzzing

Challenge 2. **Detecting memory violations** during fuzzing

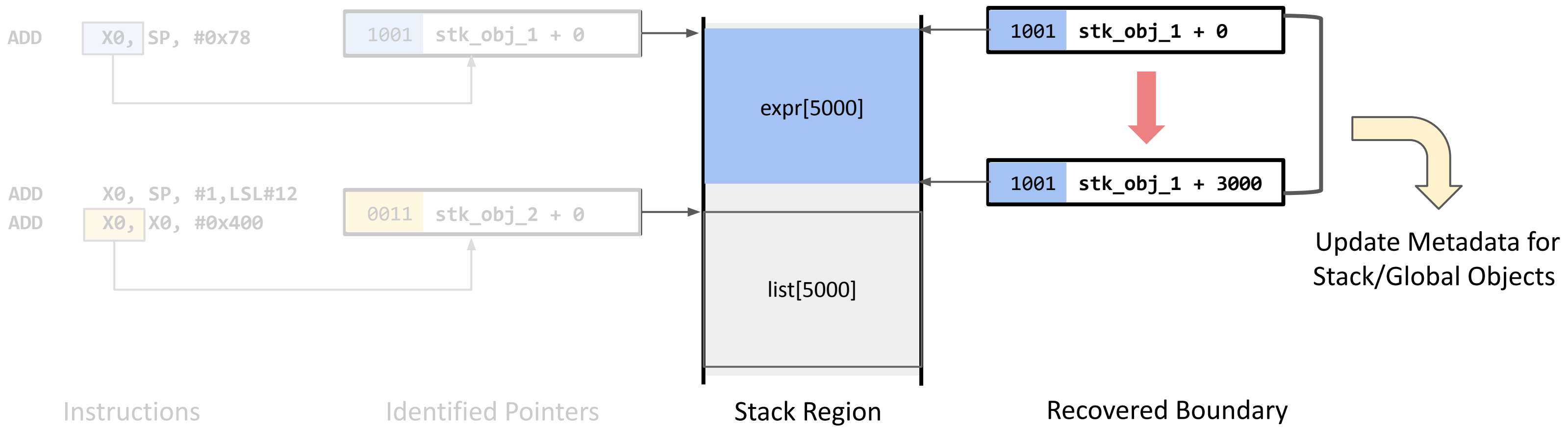
Progressive Object Recovery

1. Identifying object pointers based on how the pointer is derived
   a. for heap regions: hook memory allocators
   b. for stack and global regions: values derived out of the stack pointer and global addresses



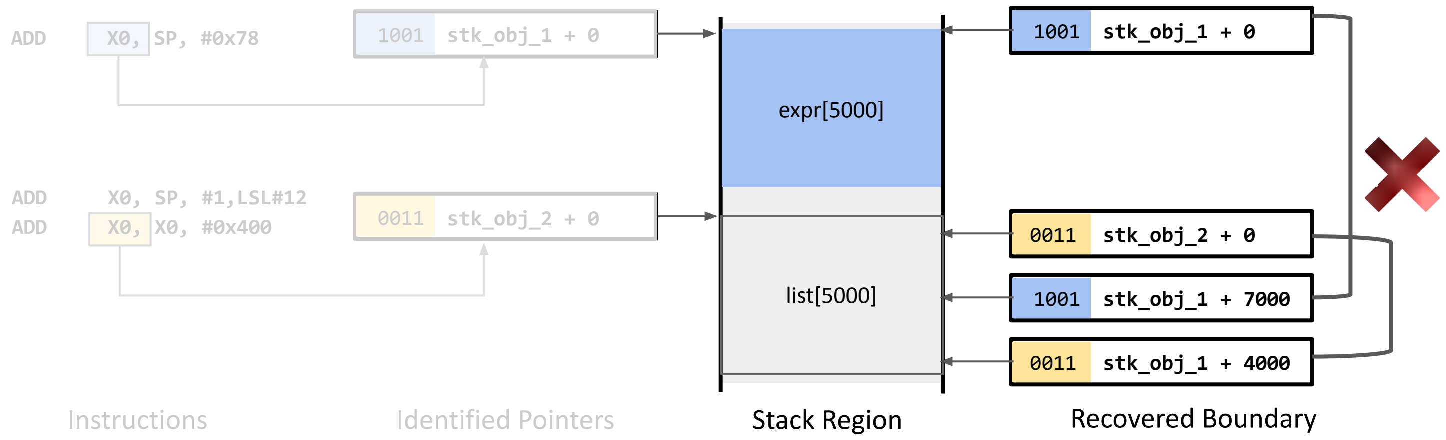
Progressive Object Recovery

2. Inferring object boundaries based on the use patterns of identified pointers
   a. `deref(addr, size)` -> loading `size` bytes from `addr`
   b. `deref(A, 8)` and `deref(A+24, 8)` -> boundary info `[A, A+32)`



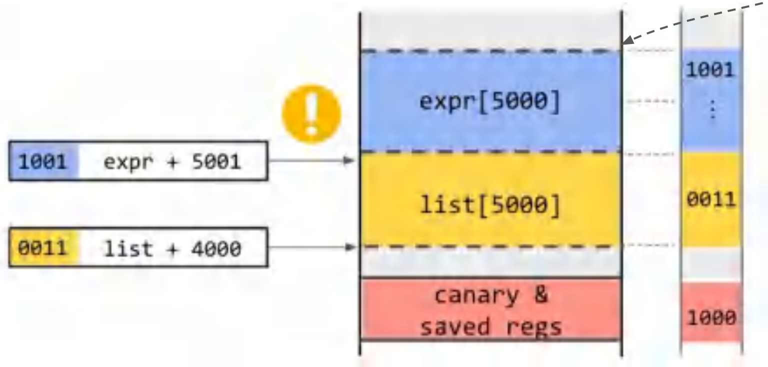
Progressive Object Recovery

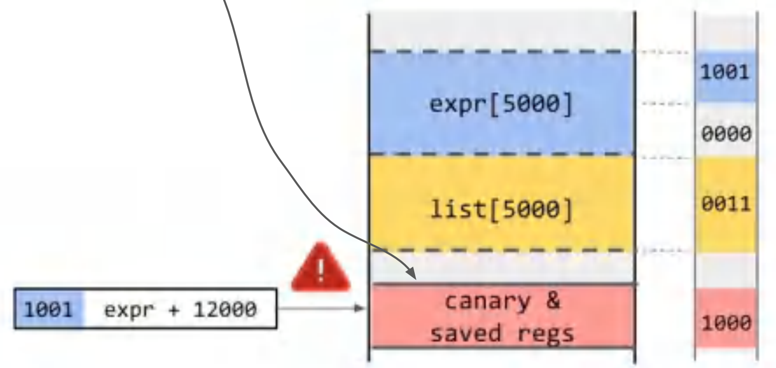
3. Progressively refining object properties using unique executions during fuzzing
   Conflicts among inferred object boundaries are indicators for memory errors



Adaptive Sanitization

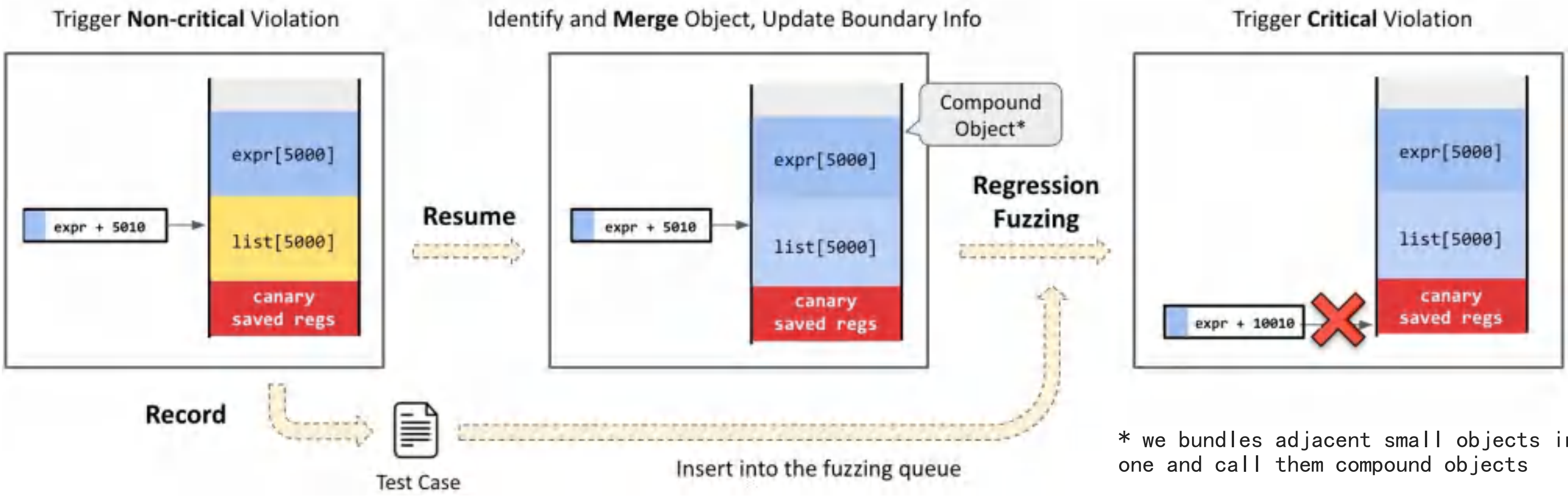
- False alarms may stall fuzzing
  - E.g., compilers may emit multiple pointers to access the same object
- Sanitization policy
  - **Non-critical** violations: relies on checks of *presumptive* properties
  - **Critical** violations: only relies on check on *deterministic* properties





Adaptive Sanitization

- Record - Resume - Regression
  - Intuition: Given enough time, fuzzers will likely expose true positives and filter away false positives.



* we bundles adjacent small objects into one and call them compound objects

Fuzzing Efficiency

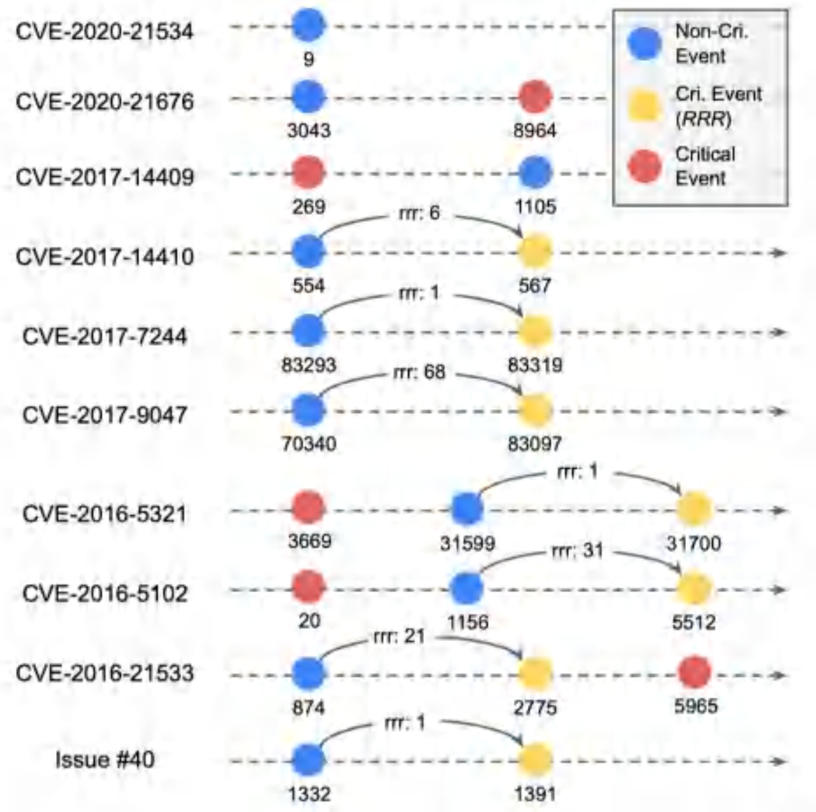
Binary	AFL++ Qemu	QASan	ASan-Retrowrite	MTSan (analog)	MTSan (libMTE)
bc	56.3	34.67	115.54	323.8	94.1
bmp2tiff	8.38	21.5	156.1	245.336	169.6
fig2dev	213.47	224.51	170.91	183.816	101.76
gif2tiff	6.71	5.74	222.46	133.76	152.25
lou_translate	2.27	0.61	1.86	2.864	2.42
img2sixel	15.3	15.29	34.77	79.12	13.99
xml_read_memory_fuzzer	183.94	67.18	82.64	225.792	61.25
ziptool	134.28	61.68	174.14	353.944	111.18
mp3gain	23.97	9.42	162.41	134.688	80.46
mxmldoc	222.61	89.87	159.28	301.896	116.79
testmxml	180.92	151.75	177.47	193.352	115.35
pcretest	42.31	2.24	70.88	91.192	37.49
pcre2test	40.78	19.16	64.24	173.072	29.12
readelf	355.48	181.63	67.2	383.576	80.92
sndfile-convert	235.61	149.97	185.08	153.888	179.48
tiff2ps	307.7	15.94	191.48	373.832	214.89
tiffcp	249.37	38.67	236.66	307.2	214.42
tiffcrop	231.48	48.65	226.14	307.808	214.01
Average	139.49	63.25 (-54.66%)	138.85 (-0.46%)	220.50 (+58.07%)	110.53 (-20.77%)

Vulnerability ID	QASan	Asan-Retro.	MTSan		MTSan-no-rec	MTSan-no-rrr	MTSan-no-rsv	MTSan-no-stg
			Cri.	Non-C.				
CVE-2017-14408			✓			✓	✓	✓
CVE-2017-14409	✓	✓	✓	✓	✓	✓	✓	✓
Bug #2065 [49]	✓	✓	✓		✓	✓	✓	✓
CVE-2017-9047			✓	✓				
CVE-2017-8361	✓				✓			
CVE-2016-10270	✓		✓		✓	✓	✓	✓
CVE-2016-10271	✓		✓		✓	✓	✓	✓
CVE-2013-4243	✓	✓	✓		✓	✓	✓	✓
CVE-2015-8668	✓	✓	✓		✓	✓	✓	✓
CVE-2017-12858	✓		✓		✓	✓	✓	
CVE-2020-21675	✓	✓	✓			✓	✓	✓
CVE-2020-21050	✓	✓	✓		✓	✓	✓	✓
CVE-2018-20005					✓			
CVE-2018-20592*	✓	✓	✓			✓	✓	✓
Issue #237 [50]*	✓	✓	✓		✓	✓		✓
Issue #5 [51]*	✓	✓					✓	
CVE-2016-5321*	✓		✓	✓	✓	✓	✓	✓
CVE-2017-7244*		✓	✓	✓			✓	✓
CVE-2016-5102*	✓	✓	✓	✓	✓	✓	✓	✓
CVE-2020-21533*	✓	✓	✓	✓	✓	✓	✓	✓
CVE-2020-21534*	✓	✓		✓	✓			
CVE-2020-21676*		✓	✓	✓	✓	✓	✓	✓
CVE-2017-14410*			✓	✓			✓	✓
Issue #40 [52]*			✓	✓			✓	✓
Total	**17**	**14**	**20**	**10**	**16**	**16**	**19**	**18**

- MTSan (analog*) yields the **highest number of executions**, following ASan-Retrowrite and MTSan (libMTE). +
- MTSan (libMTE*) reported **most bugs** during fuzzing evaluation. +

* We used instruction analogs and implemented libMTE for evaluation, please check our paper for details.

Fuzzing Efficiency - *RRR*



Time-to-Discovery of vulnerabilities (in seconds) detected duiring the fuzzing evaluation

- *RRR* escalated **seven** non-critical violations to **critical** violations
- *For more internal statistics, please refer to our paper :)*

Security Evaluation - Real-world Vulnerabilities

CVE-2017-14408	SOF	38	0	0	0	19	19	0	0	0	19	0	19	0
CVE-2017-14409	GOF	114	0	0	0	84	49	35	0	0	49	34	49	22
Bug #2065	GOF	400	0	0	0	400	0	400	0	0	0	400	0	400
CVE-2017-8786	HOF	469	469	469	469	469	469	0	469	0	469	0	469	0
CVE-2017-7245	SOF	646	0	0	0	248	248	0	0	0	248	0	248	0
CVE-2017-7246	SOF	627	0	0	0	262	262	0	0	0	262	0	262	0
Bug #2056	SOF	102	0	0	0	102	0	102	0	0	0	102	0	102
CVE-2017-9047	SOF	489	0	0	0	489	40	449	0	0	40	449	40	449
CVE-2017-8363	HOF	26	26	26	22	26	26	0	26	0	26	0	26	0
CVE-2017-8361	GOF	13	0	0	0	0	0	0	0	0	0	0	0	0
CVE-2017-8365	GOF	2	0	0	0	2	2	0	0	0	2	0	2	0
CVE-2016-10270	HOF	89	89	89	89	89	89	0	89	0	89	0	89	0
CVE-2016-10271	HOF	235	235	231	200	235	235	0	235	0	235	0	235	0
CVE-2009-2285	HOF	32	31	0	0	32	32	0	32	0	32	0	32	0
CVE-2013-4243	HOF	4	4	4	4	4	4	0	4	0	4	0	4	0
[illegible]	HOF	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
CVE-2018-20004	SOF	10	0	0	0	8	8	0	0	0	8	0	8	0
CVE-2018-20005	UAF	19	19	19	19	19	19	0	19	0	19	0	19	0
CVE-2021-20294	SOF	5	0	0	0	4	4	0	0	0	4	0	4	0
Total	27	3440	941	910	875	2589	1595	994	945	0	1595	993	1595	981

- MTSan is **more effective** than existing binary sanitizers.
- MTSan detected **most stack and global violations** with **low FP rate**.
- Performance optimizations and Compiler optimizations has **limited effect**.

Conclusion

- A feasible and practical hardware- assisted memory sanitizer, MTSan, for binary fuzzing on AArch64
  - A novel **progressive object recovery** scheme to infer object properties in binaries, including stack and global objects
  - Using **ARM MTE** to sanitize based on memory tagging
  - **Low runtime overhead**

Xingman Chen

Email: cxm16@mails.tsinghua.edu.cn

MTSan and libMTE will soon be open sourced! We are working on documentation and patenting.