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*
Fuzzing and Sanitizers

Test Case Generator

Corpus

Mutated Input

Target Program

Crash
Fuzzing and Sanitizers

Test Case Generator

Corpus

Mutated Input

Sanitizers

Target Program

Crash
Fuzzing and Sanitizers

- Test Case Generator
- Corpus
- Mutated Input
- Sanitizers
- Target Program
- Crash
Sanitizers and Memory Safety Violations

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

![Diagram showing memory access and violation]

- memory access
- memory violation
- mem_obj_A
- redzone
- mem_obj_B
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
Limitations of Existing Binary Sanitizers

2. Redzone-based approaches do not apply on binaries

Cannot add redzones without **changing memory layouts**
Limitations of Existing Binary Sanitizers

3. High runtime and memory overhead

<table>
<thead>
<tr>
<th>Binary Sanitizer</th>
<th>Bug-finding Techs</th>
<th>Object Coverage</th>
<th>Runtime Overhead*</th>
<th>Memory Overhead*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undangle</td>
<td>pointer-tracking**</td>
<td>yes</td>
<td>no</td>
<td>&gt;10x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>&gt;10x</td>
</tr>
<tr>
<td>Dr. Memory</td>
<td>redzone</td>
<td>yes</td>
<td>no</td>
<td>&gt;10x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>&gt;10x</td>
</tr>
<tr>
<td>Memcheck</td>
<td>redzone</td>
<td>yes</td>
<td>no</td>
<td>&gt;10x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>3-10x</td>
</tr>
<tr>
<td>QASan</td>
<td>redzone</td>
<td>yes</td>
<td>no</td>
<td>&gt;10x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>3-10x</td>
</tr>
<tr>
<td>ASan-Retrowrite</td>
<td>redzone</td>
<td>yes</td>
<td>no</td>
<td>1-3x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>3-10x</td>
</tr>
</tbody>
</table>

* Standalone execution, with no optimization applied.
** Use-after-free violation only.
Limitations of Existing Binary Sanitizers

3. High runtime and memory overhead

<table>
<thead>
<tr>
<th>Binary Sanitizer</th>
<th>Bug-finding Techs</th>
<th>Object Coverage</th>
<th>Runtime Overhead*</th>
<th>Memory Overhead*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Heap</td>
<td>Stack</td>
<td>Global</td>
</tr>
<tr>
<td>Undangle</td>
<td>pointer-tracking**</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Dr. Memory</td>
<td>redzone</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Memcheck</td>
<td>redzone</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>QASan</td>
<td>redzone</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>ASan-Retrowrite</td>
<td>redzone</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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, ":");
    } else 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(xmlValidCtxPtr 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 < xml_strlen(content->prefix) + 10) {
                strcat(buf, " ...");
                return;
            }
            strcat(buf, (char *) content->prefix);
            strcat(buf, ":");
        }
        if (size — len < xml_strlen(content->name) + 10) {
            strcat(buf, " ...");
            return;
        }
        if (content->name != NULL)
            strcat(buf, (char *) content->name);
        /* ... */
}

int xmlValidateElementContent(xmlValidCtxPtr 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, content, 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)
Our Inituation

- Access pattern helps to infer data structures in memory
  - Rewards\textit{(NDSS'10)}, Howard\textit{(NDSS'11)}
Our Intuition

- 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“

![Diagram showing data structures and access patterns with arrows indicating pointers to elements like `expr`, `list`, and `ptr_expr`. The diagram illustrates the flow of pointers and potential conflicts.]
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

Binary Analyzer → instrumented binary → Fuzzer

patches

runtime

Binary Rewriter

instrumented binary

Fuzzer

object metadata

Progressive Object Recovery

Record, Resume, Regression

non-critical violation

critical violation

bug reports
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

```
1001 stk_obj_1 + 0

0011 stk_obj_2 + 0
```

**Instructions**

```
ADD X0, SP, #0x78
ADD X0, SP, #1,LSL#12
ADD X0, X0, #0x400
```

**Identified Pointers**

- stk_obj_1
- stk_obj_2

**Stack Region**

```
expr[5000]
list[5000]
```

**Recovered Boundary**

```
1001 stk_obj_1 + 0
```

```
1001 stk_obj_1 + 3000
```
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

<table>
<thead>
<tr>
<th>Binary</th>
<th>AFL++ Qemu</th>
<th>QASan</th>
<th>ASan-Retrowrite</th>
<th>MTSan (analog*)</th>
<th>MTSan (libMTE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc</td>
<td>56.3</td>
<td>34.67</td>
<td>115.54</td>
<td>323.8</td>
<td>94.1</td>
</tr>
<tr>
<td>bmp2tiff</td>
<td>8.38</td>
<td>21.5</td>
<td>156.1</td>
<td>245.336</td>
<td>169.6</td>
</tr>
<tr>
<td>fig2dev</td>
<td>213.47</td>
<td>224.51</td>
<td>170.91</td>
<td>183.816</td>
<td>101.76</td>
</tr>
<tr>
<td>gif2tiff</td>
<td>6.71</td>
<td>5.74</td>
<td>222.46</td>
<td>133.76</td>
<td>152.25</td>
</tr>
<tr>
<td>lou_translate</td>
<td>2.27</td>
<td>0.61</td>
<td>1.86</td>
<td>2.864</td>
<td>2.42</td>
</tr>
<tr>
<td>img2pixel</td>
<td>15.3</td>
<td>15.29</td>
<td>34.77</td>
<td>79.12</td>
<td>13.99</td>
</tr>
<tr>
<td>xml_read_memory_fuzzer</td>
<td>183.94</td>
<td>67.18</td>
<td>82.64</td>
<td>225.792</td>
<td>61.25</td>
</tr>
<tr>
<td>zipool</td>
<td>134.28</td>
<td>61.68</td>
<td>174.14</td>
<td>353.944</td>
<td>111.18</td>
</tr>
<tr>
<td>mp3gain</td>
<td>23.97</td>
<td>9.42</td>
<td>162.41</td>
<td>134.688</td>
<td>80.46</td>
</tr>
<tr>
<td>xmlldoc</td>
<td>222.61</td>
<td>89.87</td>
<td>159.28</td>
<td>301.896</td>
<td>116.79</td>
</tr>
<tr>
<td>testxml</td>
<td>180.92</td>
<td>151.75</td>
<td>177.47</td>
<td>193.352</td>
<td>115.35</td>
</tr>
<tr>
<td>pciretst</td>
<td>42.31</td>
<td>2.24</td>
<td>70.88</td>
<td>91.192</td>
<td>37.49</td>
</tr>
<tr>
<td>pcre2test</td>
<td>40.78</td>
<td>19.16</td>
<td>64.24</td>
<td>173.072</td>
<td>29.12</td>
</tr>
<tr>
<td>readelf</td>
<td>355.48</td>
<td>181.63</td>
<td>67.2</td>
<td>383.576</td>
<td>80.92</td>
</tr>
<tr>
<td>sndtile-convert</td>
<td>235.61</td>
<td>149.97</td>
<td>185.08</td>
<td>153.888</td>
<td>179.48</td>
</tr>
<tr>
<td>tiff2ps</td>
<td>307.7</td>
<td>15.94</td>
<td>191.48</td>
<td>373.832</td>
<td>214.89</td>
</tr>
<tr>
<td>tiffcp</td>
<td>249.37</td>
<td>38.67</td>
<td>236.66</td>
<td>307.2</td>
<td>214.42</td>
</tr>
<tr>
<td>tiffcrop</td>
<td>231.48</td>
<td>48.65</td>
<td>226.14</td>
<td>307.808</td>
<td>214.01</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>139.49</strong></td>
<td><strong>63.25</strong></td>
<td><strong>138.85</strong></td>
<td><strong>220.50</strong></td>
<td><strong>110.53</strong></td>
</tr>
</tbody>
</table>

- 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 during 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

<table>
<thead>
<tr>
<th>Vulnerability ID</th>
<th>Type</th>
<th>Total</th>
<th>Valgrind</th>
<th>QASan</th>
<th>ASan-Startrewrites</th>
<th>MTSan</th>
<th>MTSan-no-rec</th>
<th>MTSan-no-rav</th>
<th>MTSan-no-stg</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2017-14408</td>
<td>SOF</td>
<td>38</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CVE-2017-14409</td>
<td>GOF</td>
<td>114</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>84</td>
<td>49</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>Bug #2065</td>
<td>GOF</td>
<td>400</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>400</td>
<td>0</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>CVE-2017-8786</td>
<td>HOF</td>
<td>469</td>
<td>469</td>
<td>469</td>
<td>469</td>
<td>469</td>
<td>0</td>
<td>469</td>
<td>0</td>
</tr>
<tr>
<td>CVE-2017-7245</td>
<td>SOF</td>
<td>646</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>248</td>
<td>248</td>
<td>0</td>
<td>248</td>
</tr>
<tr>
<td>CVE-2017-7246</td>
<td>SOF</td>
<td>627</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>262</td>
<td>262</td>
<td>0</td>
<td>262</td>
</tr>
<tr>
<td>Bug #2056</td>
<td>SOF</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>0</td>
<td>102</td>
<td>0</td>
</tr>
<tr>
<td>CVE-2017-9047</td>
<td>SOF</td>
<td>489</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>489</td>
<td>40</td>
<td>449</td>
<td>0</td>
</tr>
<tr>
<td>CVE-2017-8363</td>
<td>HOF</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>22</td>
<td>26</td>
<td>26</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>CVE-2017-8361</td>
<td>GOF</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CVE-2017-8365</td>
<td>GOF</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>CVE-2016-10270</td>
<td>HOF</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>0</td>
<td>89</td>
<td>0</td>
</tr>
<tr>
<td>CVE-2016-10271</td>
<td>HOF</td>
<td>235</td>
<td>235</td>
<td>235</td>
<td>235</td>
<td>235</td>
<td>0</td>
<td>235</td>
<td>0</td>
</tr>
<tr>
<td>CVE-2016-10275</td>
<td>HOF</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>22</td>
<td>23</td>
<td>0</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>CVE-2016-10285</td>
<td>HOF</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>CVE-2016-10295</td>
<td>HOF</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2017-8268</td>
<td>HOF</td>
<td>23</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>0</td>
<td>22</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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.