Beyond Spatial and Temporal Memory Safety

Zhe Chen, Chuanqi Tao, Zhiyi Zhang, and Zhibin Yang
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
Nanjing, Jiangsu, China
zhechen@nuaa.edu.cn

ABSTRACT

The unsafe features of C often lead to memory errors that can result in vulnerabilities. Dynamic analysis tools are widely used to detect such errors at runtime and enforce memory safety. It is widely believed that memory safety exactly consists of spatial and temporal safety, thus all existing analysis tools aim at detecting spatial or temporal errors. In this paper, we introduce another class of memory safety, namely *segment safety*, which has been neglected in previous work. Indeed, state-of-the-art analysis tools cannot detect segment errors. Thus we propose and implement a new approach to detect segment errors at runtime.

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Software and its engineering → Software reliability**; **Software safety**; **Software testing and debugging**;

KEYWORDS

memory safety, spatial safety, temporal safety, dynamic analysis, C programs, segment errors

ACM Reference Format:
Zhe Chen, Chuanqi Tao, Zhiyi Zhang, and Zhibin Yang. 2018. Beyond Spatial and Temporal Memory Safety. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195090

1 INTRODUCTION

C is widely used for implementing systems software and embedded software which are usually security or safety critical systems. Unfortunately, the unsafe language features, such as the low-level control of memory, often lead to memory errors that can result in vulnerabilities. Dynamic analysis tools are widely used to detect such errors at runtime and enforce memory safety because of their precision (e.g., no false positives). For the state-of-the-art tools, Google's AddressSanitizer (ASan) [10] uses a combination of guard-based [4, 5] and object-based approaches [1, 3], whereas SoftboundCets (SoCets) [6–8] uses a pointer-based approach [9, 11].

Memory safety is traditionally divided into *spatial safety* and *temporal safety*. A spatial error is caused by dereferencing a pointer outside the bounds of its referent, such as buffer overflows. A temporal error is caused by accessing a referent that no longer exists, such as dangling pointer dereferences (i.e., use-after-free of heap or stack objects). The following code shows an example of spatial and temporal errors.

```
1 int *p = (int*)malloc(5*sizeof(int));
2 *(p+5) = 0;          //spatial error
3 free(p);     *p = 0; //temporal error
```

It is widely believed that memory safety exactly consists of spatial and temporal safety, thus all existing analysis tools aim at detecting spatial or temporal errors [6–8, 11]. However, we find that there is another class of memory safety, namely *segment safety*, which has been neglected in previous work. A consequence is that, existing tools are not able to detect segment errors, as these errors are neither spatial nor temporal. Although existing tools can accidentally detect some segment error as it simultaneously causes a spatial or temporal error occasionally, they cannot ensure complete segment safety, and cannot pinpoint the real cause and source location of the error.

Therefore, we propose to formally address segment safety as an independent safety class. We introduce segment safety by examples in Section 2, and show how to detect segment errors at runtime in Section 3.

2 SEGMENT MEMORY SAFETY

A program memory is logically divided into a text segment for storing program instructions and several segments for storing program data. In particular, the *text segment* (code segment) stores machine code of the compiled program, and is often read-only to prevent the program from being accidentally modified. The *data segment* (initialized data segment) stores all constant, initialized global, static, and external variables. The *BSS segment* (uninitialized data segment) stores all uninitialized global, static, and external variables, which are by default initialized to zero. The *stack segment* stores all local variables, and the parameters and return address of function calls. The *heap segment* stores all dynamically allocated variables.

The use of these segments should respect some restrictions to ensure memory safety. For example, the text segment should not be read or written as program data, while data in the other segments should not be interpreted as program instructions. Besides, objects in the heap segment can be freed, while explicit deallocation is forbidden for the other segments. Ironically, although the hazards of violating segment safety policies are clear, programs containing segment errors can successfully escape from compilers and existing analysis tools.

The instances of segment errors include invalid dereferences caused by arbitrary casts (e.g., dereferencing function pointers

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3195090

```
1  void foo() { /*omitted*/ }
2  int main() {
3    char ch;
4    void (*fp)() = foo;
5    char *p = (char *)foo;
6
7    (*fp)();
8    ch = *p; /*segment error*/
9    /* or *p = ch; */
10   return 0;
11 }
```

Figure 1: Invalid Dereference 1

```
1  int main()
2  {
3    char ch;
4    char *p = &ch;
5    void (*fp)() = (void (*)())p;
6
7    (*fp)(); /*segment error*/
8
9    return 0;
10
11 }
```

Figure 2: Invalid Dereference 2

```
1  void f_stack(int **p_addr) {
2    int x;
3    *p_addr = &x;
4  }
5  int main() {
6    int *p1, *p2;
7    f_stack(&p1);
8    p2 = p1;
9    free(p2); /*segment error*/
10   return 0;
11 }
```

Figure 3: Invalid Free

as data pointers, and vice versa) and invalid frees (e.g., explicitly deallocating non-heap objects). For example, Figures 1, 2 and 3 show three programs with segment errors. In Fig. 1, we first convert a function pointer into a data pointer (Line 5), and then access the text segment that stores the function code as program data (Line 8). The memory read does not cause any symptom, while the memory write (Line 9) results in a segmentation fault at runtime. In Fig. 2, we first convert a data pointer into a function pointer (Line 5), and then interpret the data in the stack segment as a function (Line 7). The function call also results in a segmentation fault. In Fig. 3, we free an object in the stack segment (Line 9), again resulting in a segmentation fault. Note that these errors are neither spatial nor temporal, as the accessed memory could be in the bounds of referent and still alive.

Although the above errors seem obvious in the simplified examples, debugging segment errors in complex programs could be much harder. Indeed, all these programs can be successfully compiled without any error or warning, e.g., using gcc. Furthermore, state-of-the-art analysis tools, such as ASan and SoCets, cannot detect invalid dereferences either. However, ASan can detect the invalid free thanks to the side effects of their object-based analysis approach, while SoCets can detect the invalid free thanks to the simultaneous temporal error. That is, they are not designed to detect segment errors, thus they cannot pinpoint the real causes and source locations of errors.

3 APPROACH AND IMPLEMENTATION

We have proposed a new approach to detect segment errors at runtime. The general idea is to track the segment type of every memory object. When a pointer is used, we check whether the way of using the pointer conforms to the segment type of the referent. For example, when a pointer is dereferenced as program data or a function, we check whether the accessed object is in a program data segment or the text segment, respectively. When a pointer is passed to `free`, we check whether the deallocated object is in the heap segment.

We have developed a prototype tool, namely RVMS, which implements the approach by instrumenting source code. By running the instrumented program, segment errors can be automatically detected. According to our preliminary evaluation, RVMS can effectively detect all mentioned segment errors. For example, it reports the following message for the program in Fig. 1.

```
1.c: In function 'main':
1.c:8:9: error: dereferenced pointer 'p' points to a function,
     not a data block. [segment violation]
1 error generated.
```

4 CONCLUSION AND FUTURE WORK

We proposed a new class of memory safety, and showed how to detect segment errors at runtime. However, adding this feature to existing tools can cause extra runtime overhead, because of tracking the additional segment type. Thus, one future work is how to integrate our method into the existing approaches for detecting spatial and temporal errors, without significant efficiency losses. Then we need further evaluation of the integrated approaches. Besides, we will also try to integrate RVMS into our runtime verification tool MOVEC [2].

ACKNOWLEDGMENTS

This work is supported by the Joint Research Funds of National Natural Science Foundation of China and Civil Aviation Administration of China (No. U1533130).

REFERENCES

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *Proceedings of the 18th USENIX Security Symposium*. USENIX Association, 51–66.

[2] Zhe Chen, Zhemin Wang, Yunlong Zhu, Hongwei Xi, and Zhibin Yang. 2016. Parametric Runtime Verification of C Programs. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016) (Lecture Notes in Computer Science)*, Vol. 9636. Springer, 299–315.

[3] Dinakar Dhurjati and Vikram S. Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*. ACM, 162–171.

[4] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-weight bounds checking. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012*. ACM, 135–144.

[5] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 911–922.

[6] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages, SNAPL 2015 (LIPIcs)*, Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 190–208.

[7] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*. ACM, 245–258.

[8] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010*. ACM, 31–40.

[9] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (2005), 477–526.

[10] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA*. USENIX Association, 309–318.

[11] Matthew S. Simpson and Rajeev Barua. 2013. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Software - Practice and Experience* 43, 1 (2013), 93–128.