Tag-Protector: An Effective and Dynamic Detection of Out-of-bound Memory Accesses

Ahmed Saeed, **Ali Ahmadinia**

School of Engineering and Built Environment
Glasgow Caledonian University, United Kingdom

Mike Just

School of Mathematics and Computer Sciences,
Heriot-watt University, United Kingdom

Outline

- Introduction
- Problem Statement
- Proposed solution
- Methodology
- Implementation
- Results and Discussion
- Conclusion

Introduction

- **Illegal memory accesses (IMAs) are major concerns in applications written with programming languages like C/C++.**
  - Typical programming errors: out-of-bound array indexing and dangling pointer dereferences
  - Spatial IMA :more commonly known as buffer overflow
  - Temporal IMA: also known as use-after-free access

```
int funcall(int argc , char **argv){
  char *buffer,*ptr,buffer2[MAX_size];//stack alloc
  ptr=(char *)malloc(Max_size);//heap alloc
  if(ptr==NULL) exit(1);
  buffer=ptr;
  strcpy(buffer,argv[1]);/*possible heap overflow*/
  strcpy(buffer2,argv[2]);/*possible stack ovrflow*/
  free(buffer);
  memcpy(ptr,buffer2,Max_size) /*use-after-free */
  printf("String one : %s\n,buffer") /*use-after-free */
  printf("String two : %s\n,buffer2")} /*use-after-free */
```

Ahmed Saeed (ahmed.saeed@gcu.ac.uk)
School of Engineering and Computing
Glasgow Caledonian University

Problem Statement

- **Increase in software content and network connectivity.**
- **Software is not fully trustable.**
  - Software-based attacks: Stack smashing through buffer overflows
  - Illegal memory reads and writes
- **Protect System/Data / Programs against**
  - Extraction of secret information: Data confidentiality
  - Modification in the behavior: Data integrity
  - Denial of service: Availability

Proposed Solution

- Detect IMAs dynamically through tag based protection
- Based on source code instrumentation through LLVM compiler framework
- Targets data confidentiality and integrity attacks.
- Effectiveness evaluated through various benchmark suites and testbed codes
- Presented lower memory and performance overhead

Methodology

- **Require application source code**
- **Implementation is based on following steps.**
  - Convert code in to Intermediate Representation(IR)
  - Detect memory allocations instructions
  - Link each memory objects with a special tag
  - Detect memory access instructions.
  - Insert tag address and value check instructions

Methodology

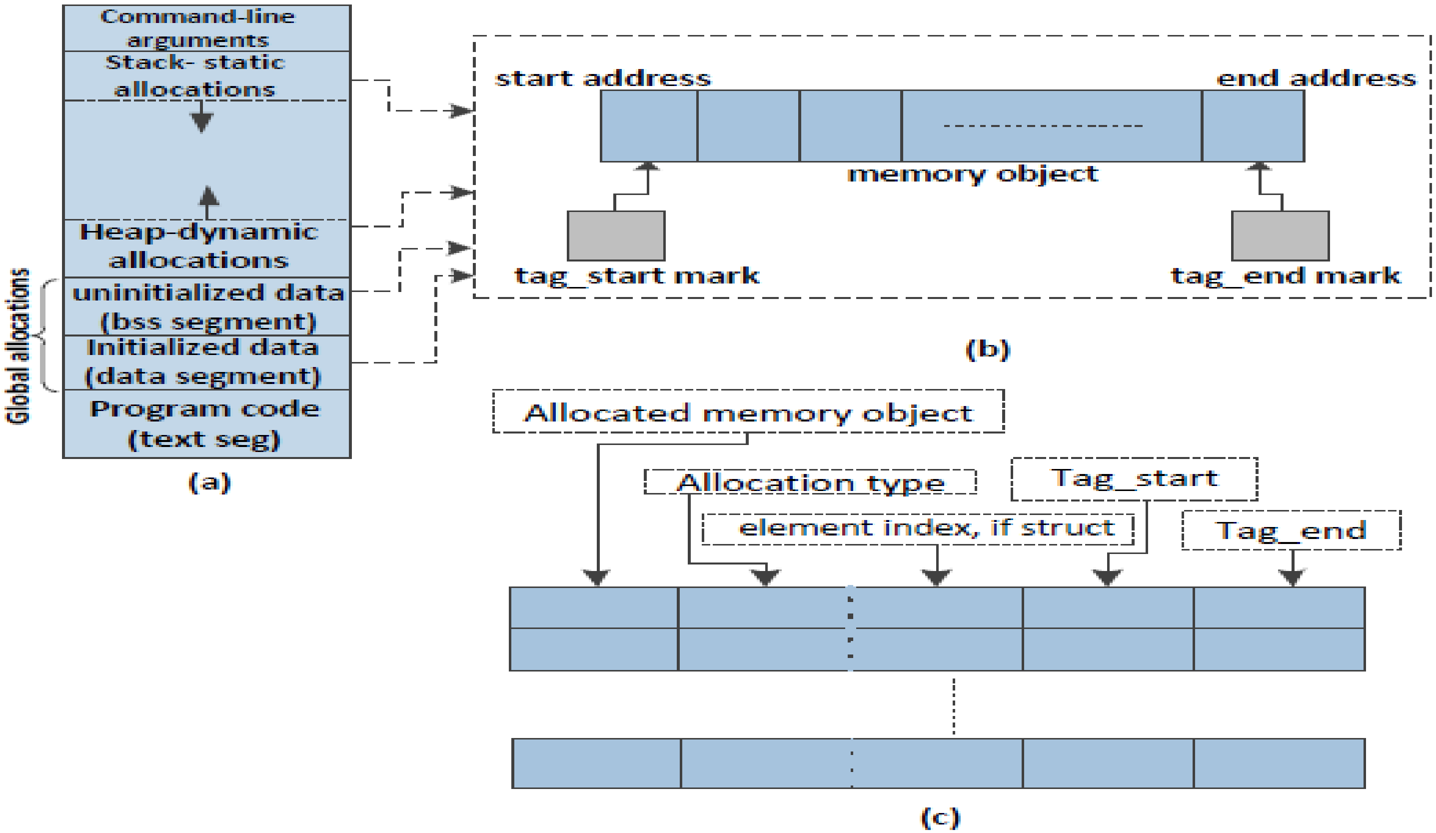

Figure 1: (a) Typical memory layout of a C program. (b) Memory objects coupled with tag_start and tag_end marks. (c) Record table layout used by tag-protection at the time of code instrumentation.

Ahmed Saeed (ahmed.saeed@gcu.ac.uk)
School of Engineering and Computing
Glasgow Caledonian University

Implementation

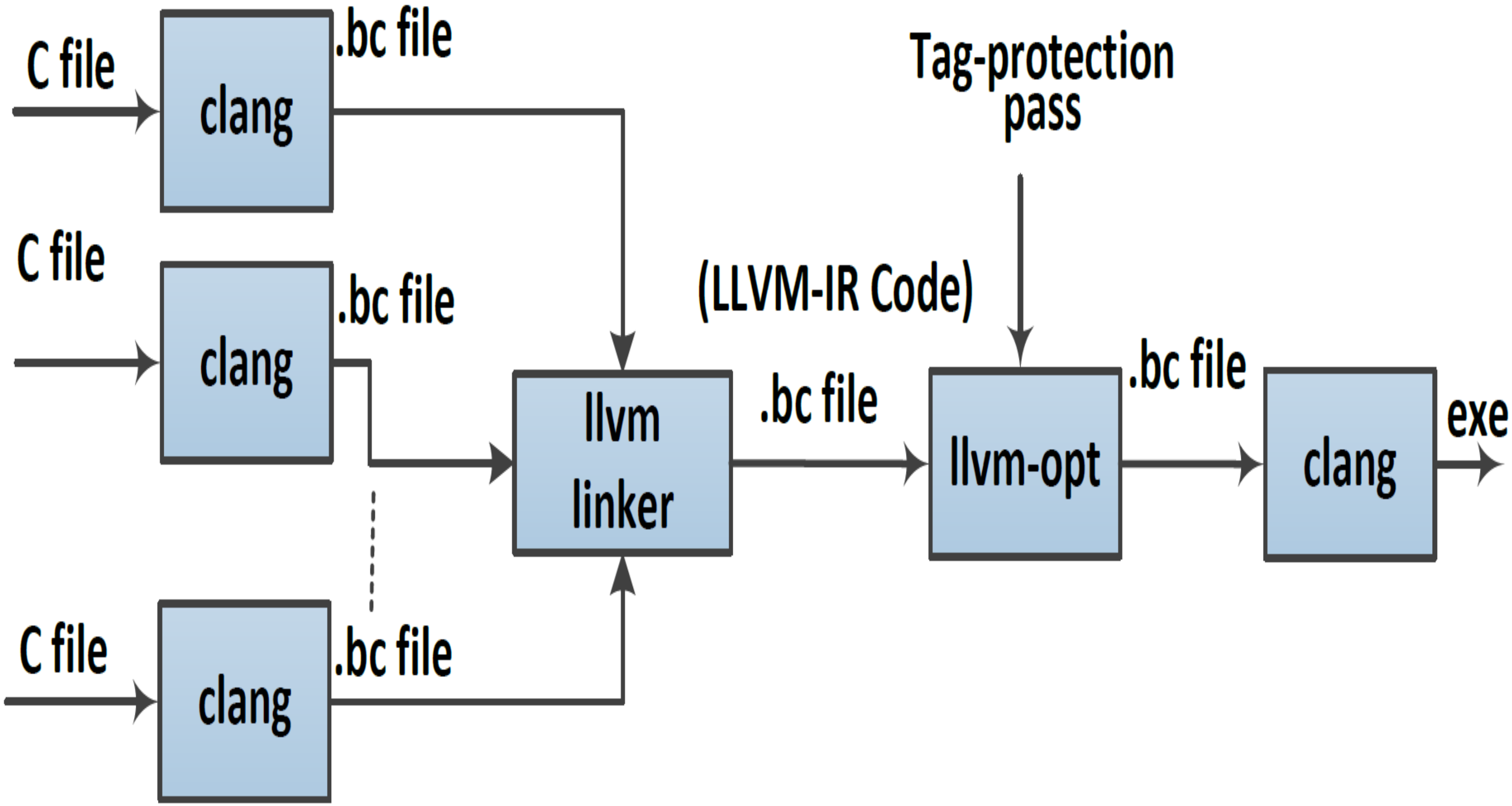

Figure 2: Tag-Protection implementation block diagram

Ahmed Saeed (ahmed.saeed@gcu.ac.uk)
School of Engineering and Computing
Glasgow Caledonian University

Implementation

Algorithm 1: Stage-5:Tag checks placement.

Input: Instrumented LLVM-IR code β_4 generated in stage-4 of tag-protection solution ; memory map table Tag_map_table;Dedicated tag address $globaltag$

Output: Final Instrumented LLVM-IR code γ generated through LLVM *opt* command using stage-5 of tag-protection solution

```
for each function definition fun_def in β_3 do
    for each instruction fun_inst in fun_def do
        if  fun_inst is function call without definition
        and not a memory allocation or deallocation call
        then
            for each function argument fun_arg in
            fun_inst do
                Create two memory objects before_fun
                and after_fun. Retrieve respective
                tag_start and tag_end marks from
                Tag_map_table.
                Read address location next to tag_end
                address before and after fun_inst
                instruction and store the values in
                before_fun and after_fun respectively.
                Place tag check instruction after function
                call fun_inst comparing before_fun and
                after_fun memory objects.
            end
        end
        if  fun_inst is a STORE/LOAD instruction
        then
            Retrieve respective tag_start and tag_end
            marks from Tag_map_table and get address
            to be accessed address_tobe_accessed by the
            fun_inst instruction.
            Perform address comparison checks:
            address_tobe_accessed with the tag_start
            and tag_end.
        end
    end
end
Delete memory map table Tag_map_table.
Save modified LLVM-IR code as an final instrumented
LLVM-IR code γ
```

Ahmed Saeed (ahmed.saeed@gcu.ac.uk)
School of Engineering and Computing
Glasgow Caledonian University

Results and Discussion

Table 1: Effectiveness of the proposed tag-protection solution on different applications from BugBench benchmark suite

Application	Bug location	Bug type	Detected
bc-1.06	storage.c:177	heap	yes
bc-1.06	util.c:577	heap	yes
bc-1.06	bc.c:1425	global	yes
gzip-1.2.4	gzip.c:457	global	yes
man-1.5h1	man.c:978	global	yes
ncompress	compress.c:896	stack	yes
polymorph-0.40	polymorph.c:120	global	yes
polymorph-0.40	polymorph.c:193	stack	yes
squid-2.3	ftp.c:1024	heap	yes

Results and Discussion

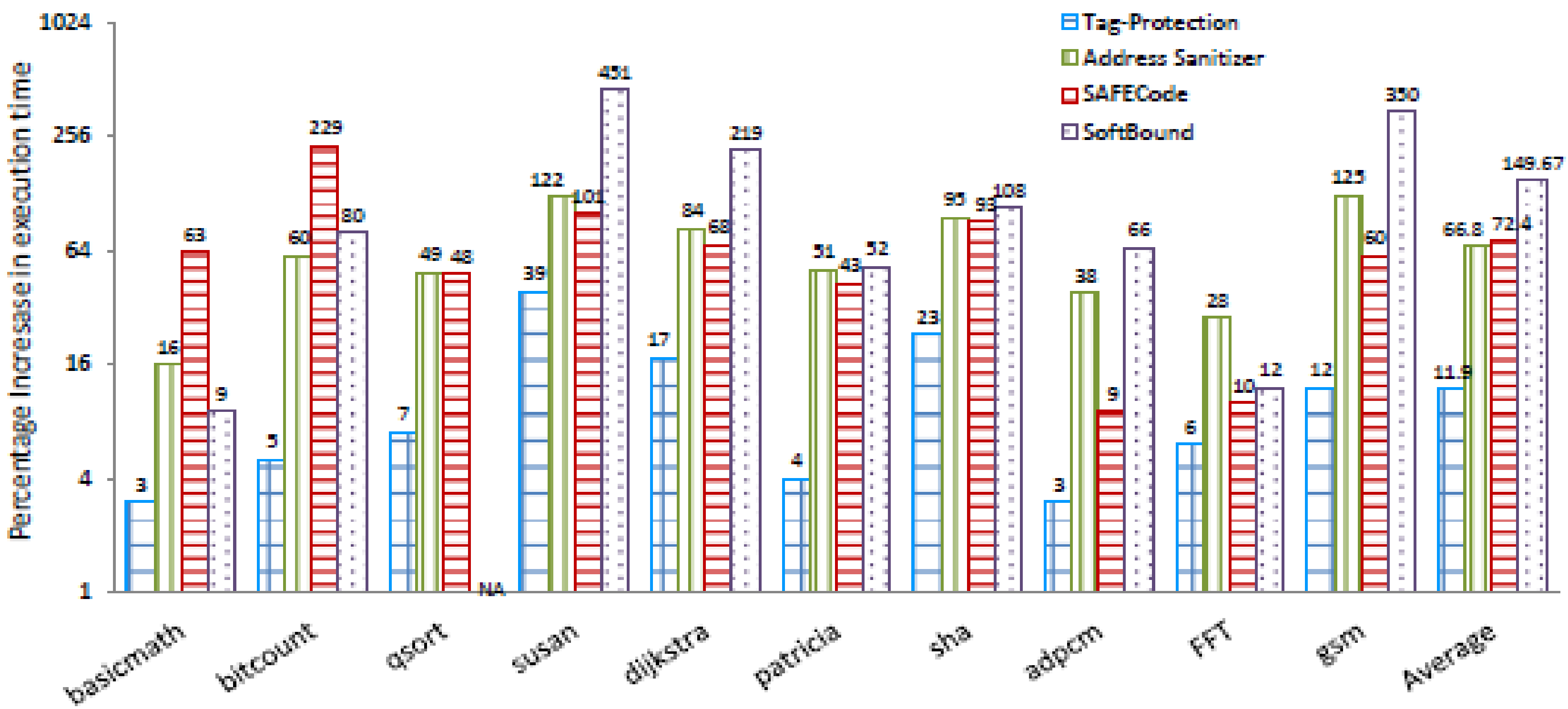

Figure 3: Performance overhead comparison for MiBench Benchmark applications with existing solutions

Results and Discussion

Table 2: Increase in memory utilization for instrumented MiBench Benchmark applications

Application	Non-instrume-nted (KB)	Instrumen-ted(KB)	Increase (KB)
basicmath	7308	7352	44
bitcount	7308	7352	44
qsort	10056	10058	2
susan	7844	8556	712
dijkstra	7396	7404	8
patricia	14088	14104	16
sha	7356	7372	16
adpcm	7304	7348	44
FFT	7772	7880	108
gsm	7412	7468	56
Total	**83844**	**84894**	**1050**

Ahmed Saeed (ahmed.saeed@gcu.ac.uk)
School of Engineering and Computing
Glasgow Caledonian University

Results and Discussion

Table 3: Increase in binary size for instrumented applications from MiBench embedded benchmark suite.

Application	TPP	AS§	SC†	SB∂
basicmath	1.32x	91.41x	19.27x	6.18x
bitcount	1.19x	98.71x	26.21x	7.2x
qsort	1.79x	157.32x	37.56x	9.68x
susan	19.91x	33.32x	8.83x	5.58x
dijkstra	2.09x	100.62x	26.2x	6.8x
patricia	2.6x	103.1x	26.11x	7.72x
sha	2.63x	104.68x	25.5x	7.85x
adpcm	1.83x	155.54x	32.41x	9.78x
FFT	3.28x	101.56x	26.77x	7.15x
gsm	2.05x	20.19x	6.24x	4.03x
Average	**3.87x**	**96.65x**	**21.95x**	**7.2x**

§AddressSanitizer, †SafeCode, ∂SoftBound,

Conclusion

- **A fast and effective tag-protection solution to detect illegal memory accesses.**
- **Implemented as an instrumentation pass using LLVM and operates at source-code level.**
- **Less performance overhead when compared with the publicly available tools.**

Ahmed Saeed (ahmed.saeed@gcu.ac.uk)
School of Engineering and Computing
Glasgow Caledonian University

Any Questions?

Ahmed Saeed (ahmed.saeed@gcu.ac.uk)
School of Engineering and Computing
Glasgow Caledonian University