Heriot-Watt University
Research Gateway

Tag-Protector: An Effective and Dynamic Detection of Illegal Memory Accesses Through Compile-time Code Instrumentation

Citation for published version:
Saeed, A, Ahmadinia, A & Just, M 2016, 'Tag-Protector: An Effective and Dynamic Detection of Illegal Memory Accesses Through Compile-time Code Instrumentation', *Advances in Software Engineering*. <http://downloads.hindawi.com/journals/ase/aip/345056.pdf>

Link:
Link to publication record in Heriot-Watt Research Portal

Document Version:
Peer reviewed version

Published In:
Advances in Software Engineering

General rights
Copyright for the publications made accessible via Heriot-Watt Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
Heriot-Watt University has made every reasonable effort to ensure that the content in Heriot-Watt Research Portal complies with UK legislation. If you believe that the public display of this file breaches copyright please contact open.access@hw.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 22. May. 2025

Tag-Protector: An Effective and Dynamic Detection of Illegal Memory Accesses Through Compile-time Code Instrumentation

Ahmed Saeed, Glasgow Caledonian University, Glasgow, UK
Ali Ahmadinia[1], Department of Computer Science, California State University San Marcos, CA, USA
Mike Just, School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK

Abstract

Programming languages permitting immediate memory accesses through pointers often result in applications having memory-related errors, which may lead to unpredictable failures and security vulnerabilities. A light-weight solution is presented in this paper to tackle such illegal memory accesses dynamically in C/C++ based applications. We propose a new and effective method of instrumenting an application's source code at compile time in order to detect illegal spatial and temporal memory accesses. It is based on creating tags, to be coupled with each memory allocation and then placing additional tag checking instructions for each access made to the memory. The proposed solution is evaluated by instrumenting applications from the BugBench benchmark suite and publicly available benchmark software, Runtime Intrusion Prevention Evaluator (RIPE), detecting all the bugs successfully. The performance and memory overheads are further analyzed by instrumenting and executing real world applications from various renowned benchmark suites. In addition, the proposed solution is also tested to analyze the performance overhead for multi-threaded applications in multi-core environments. Overall our technique can detect a wide range of memory bugs and attacks with reduced performance overhead and higher detection rate as compared to the similar existing countermeasures when tested under the same experimental set-up.

General Terms: Security, Reliability, Languages, Code instrumentation

Keywords: Dynamic memory accesses checking, compile-time code instrumentation, spatial and temporal memory safety, buffer overflows, dangling pointer dereferences

1. INTRODUCTION

Illegal memory accesses (IMAs) such as out-of-bound buffer read/write operations and dangling pointer dereferences are major concerns in applications written with programming languages like C/C++. These languages provide a powerful set of low-level features to software developers such as direct memory accesses and arithmetic operations on pointers. Normally in such languages, the starting address is assigned to a pointer when a memory area of required size is allocated, whereas an access is considered legal only when either its actual pointer or a pointer derived from it is used between the allocation and deallocation of a specific memory area. A pointer directing to a memory location that has already been deallocated is called a dangling pointer. A spatial IMA, which is more commonly known as buffer overflow or underflow, may occur when a pointer accesses memory outside the range of its allocated memory object. A temporal IMA, also known as dangling pointer dereference, occurs when a dangling pointer is used at the time of the access.

Typical programming errors, such as out-of-bound array indexing and dangling pointer dereferences, are common and cause indeterministic behavior because these operations can write to memory locations in ways not defined by the designer. For example, memory accesses can happen outside the intended range if the index calculation of an array is based on an erroneous formula. It is a difficult and tedious job to detect and diagnose such behavior using static analysis-based tools. Even when an application is tested intensively through these tools, such bugs can still exist as it is practically impossible to create all of the input combinations for an error to occur in the development or test phase. Furthermore, without required protection, many security threats like viruses, Trojans and worms can modify appli-

[1]Corresponding author: Ali Ahmadinia (aahmadinia@csusm.edu)

cation data by gaining illegitimate access to secured blocks of the memory such as through buffer overflow attacks [1]. Software-based attacks have become increasingly widespread and buffer overflow is one of the major causes of such security outbreaks. According to a report published by Sourcefire [2], buffer overflow based attacks are responsible for 14% of all and 35% of critical vulnerabilities over the past 25 years. Stack smashing [3] is a classic example of such an attack where an attacker can simply replace the return address of a function on the stack through a buffer overflow. In the case of unprotected execution, on function return, the control may be switched to the specific location where malicious code is placed. Similarly, return-oriented programming (ROP) [4] is a relatively new way to accomplish security exploits, after gaining control of the execution flow through a buffer overflow. ROP and Return-into-libc based attacks execute a group of instructions from the existing code to create new functionality. Different solutions already exist to detect such attacks that are either based on removing vulnerabilities statically through safe languages (e.g., [5], [6], [7], [8]) or inserting run-time checks for the detection of out-of-bounds memory accesses (e.g., [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]). Moreover, some of the existing dynamic memory checkers do not cover all bugs while others provide complete protection at the cost of notable performance overhead.

In this paper, a fast and effective memory protection technique is presented to detect a wide range of IMA bugs and attacks dynamically such as out-of-bound (OOB) buffer and dangling pointer based memory accesses, which can be sub-categorized as buffer over-reads and over-writes, buffer under-reads and under-writes, OOB accesses through direct indexing and dereferencing dangling pointers. Our tag-protection solution is built upon the well-known Jones and Kelly approach [10] of tracking each memory object pointed to by its referent pointer. Ruwase et al. carried forward this work and presented an enhanced solution called CRED [11]. Their solution fails to detect overflows when buffers are allocated inside `struct` type data variables and for many library functions such as `snprintf() and fscanf()` under specific cases for memory objects allocated on stack/bss/data-segments. On the other hand, our proposed solution is generalized and traverses through all the memory allocations, linking the memory objects with tag marks and tracking the memory accesses through these tag marks. The base and end addresses of each memory object are calculated and stored in the corresponding tag marks. The address comparisons instructions are inserted for each instruction that either read or write to that particular memory object. Whenever a reserved memory area is accessed through an allocated pointer, our solution verifies the access by comparing the address being accessed with the bound addresses that are stored in the tag marks linked with that particular memory object. If any underflow or overflow occurs, the address of the accessed memory block would either fall behind or exceed the address values stored in the associated tag marks, then an IMA bug alert will be generated. The access is considered legal only if the address is found within the bounds. Furthermore, to detect dangling pointer dereferences our tag-protection solution creates a dedicated tag address and initializes the corresponding tag mark with it when a memory object is deallocated. On each memory access, to detect any dangling pointer dereference, the tag marks are also compared with that dedicated tag address.

The proposed solution does not require modifications to the application's source code and it is based on the automatic code instrumentation at compile time by adding new instructions without affecting the actual flow of data. We have placed the run-time checks through compile-time code instrumentation. The LLVM v3.4 [19] compiler infrastructure has been used for implementation purposes. Our proposed solution is different from existing solutions [15; 20; 17; 16] in a way that it does not perform table-up searches to load start and end addresses of a particular memory object. The proposed solution effectiveness and performance overhead is measured on real-world applications from several benchmark suites [21; 22; 23; 24] and test-bed codes [25]. Based on the experimental results, it is shown that our approach has lower performance overhead when compared with the existing

solutions and hence it is a good choice for testing applications that are written in C/C++. Our solution can detect various memory bugs and security attacks including out-of-bound read and write accesses, stack-overflows, stack-underflows, heap-overflows, heap-underflows, overflows and underflows in globally defined variables (data and bss segments), direct-indexed overflows/underflows and dangling pointer dereferences.

The paper is organized as follows: The current work related to the existing memory bug detectors is summarized in section 2. The general approach and implementation details of the proposed tag-protection solution are described in section 3. Section 4 presents the effectiveness, performance evaluation, and comparison with the similar existing solutions while section 5 concludes this paper.

2. RELATED WORK

Different solutions have been proposed to detect memory errors in C/C++ based applications. Memory errors are typically grouped into spatial errors and temporal errors. Various Hardware-based memory safety solutions [26],[27], [28] have been presented in the literature with reduced performance overhead as compared to software-based solutions. Such solutions are not generic and require either dedicated hardware modules or specific modifications in the processor pipeline or cache architecture. Similarly, dynamic information flow tracking (DIFT) based hardware-assisted solutions such as Secure Program Execution [29], Dynamic Tainting [30] and libdft [31] are based on tagging data coming from untrusted sources and then tracking their usage as the application executes. These techniques require modifications in application's data, processor architecture, and memory layout. Therefore, they are not feasible for such systems that use more common hardware design approaches.

As mentioned earlier, static analysis tools do not provide 100% guarantee, and hence, they have been substituted with dynamic techniques. Many software-based dynamic memory bug detection techniques have been proposed in the literature that vary in implementation level, memory utilization, run-time overhead, types of bugs detected, the probability of detecting bugs, supported architectures and many other features. Based on the implementation level, these tools can be classified into two types. For instance, solutions such as Purify [32], Valgrind [33], Dr. Memory [34], and Dynamic Tainting [30] operate at binary level while other techniques like Backwards-Compatible Bounds Checking [10], Dynamic Buffer Overflow Detector [11], Backwards-compatible Array Bounds Checking [12], SAFECode [13], BaggyBounds [16], SoftBound [15], PAriChecks [17], LBC [18] and AddressSanitizer [35] require source code to insert run-time checks through compile-time instrumentation. Some dynamic IMA bug detectors such as StackGuard [9], LibSafe [36], ProPolice [37] and StackShield [38] provide protection for stack memory only, while WIT [14] detects invalid writes only. Other dynamic tools can detect a wide range of bugs successfully, but at the expense of large performance overhead. In this section, we have discussed only those solutions that are similar to our proposed solution.

Referent-object based approaches such as Backwards-Compatible Bounds Checking [10], CRED [11], PAriCheck [17] and BaggyBound [16] work at source-code level by maintaining a separate table, using different data structures, to record bounds of each memory allocation. This table is then used to verify memory accesses by performing table lookups at run-time. These techniques differ in the implementation and handling of record tables. Avijit et al. extended the solution of LibSafe [36] by implementing LibsafePlus and TIED [39; 40]. Static allocations are handled by TIED whereas LibsafePlus deals with dynamic information about the stack size and heap allocations. These details are then used at run-time to detect any overflow. Furthermore, these solutions do not provide dangling pointer dereference detection capabilities and also require customized memory allocator libraries to achieve reduced performance overhead. SAFECode [13] which is also an object-based approach, operates at the source-code level. It instruments loads and stores to prevent illegal memory accesses, uses points-to analysis and type-inference to find type-safe regions of the heap,

and partitions the heap into regions to eliminate load/store checks on type-safe heap regions. They have also presented a solution [41] that provides protection against dangling pointer dereferences. Similarly, PAriCheck [17] computes bounds and assigns label to each fixed-size memory block and stores these labels in a separate table at run-time. On each pointer arithmetic operation, the label is compared by values in the table. This solution has reported an average overhead of 9.5% for SPEC CINT2000 benchmark suite but does not detect dangling pointer dereferences and overflows in structures. There are also object-based approaches that have been applied on the whole operating system [42].

On the other hand, the pointer-based approaches such as MSCC [43], CCured [6], SoftBound [15], LBC [18] and others [44; 26; 45] associate base and size metadata with every pointer and insert run-time checks manipulating metadata information during load/store operations of pointer values. CCured [6] combines instrumentation with static analysis to insert run-time checks and removes redundant checks at compile time. CCured fails to work when un-instrumented pre-compiled libraries are in use. Unlike prior pointer-based (also known as fat pointer) approaches that modify pointer representations and object layouts [44; 46; 18], SoftBound records the bounds information in a disjoint metadata which is accessed via explicit table lookups on loads and stores of pointer values only. Nagarakatte et al. extend their SoftBound technique and presented another solution called CETS [20] to detect temporal safety errors.

AddressSanitizer [35] verifies whether each allocated memory block is safe to access by creating shadow memory around stack and global objects to detect overflows. The shadow memory is checked on each load and store request. The only drawback is that it also requires modified run-time library to create shadow memory around allocated heap regions. The current implementation is based on the LLVM compiler infrastructure. AddressSanitizer might fail to detect dangling pointer dereferences when a large amount of memory is allocated and deallocated (between the deallocation and its next use).

Among different techniques that operate at the binary level, Purify [32] is one of the early solutions in this area. It enforces the insertion of extra checking instructions directly into the application's object code and verifies every memory read and write operation performed by the application under execution. Valgrind [33] is an openly available instrumentation framework for building dynamic analysis tools. The Valgrind's MemCheck tool uses shadow memory to keep track of which memory areas have been allocated and pinpoints illegal accesses to uninitialized memory. Another approach presented by Doudalis et al. [30] associates unique taint marks with each pointer and its allocated memory block. These taint marks are then propagated and verified through taint checking instructions, whenever memory is accessed. This approach works at binary level but requires modification in run-time libraries to generate taint marks properly. Such tools do not require source code and recompilation but they increase the memory utilization and execution time overhead largely as the tool is first loaded into the memory and main application runs on the top of it.

Each of the above mentioned memory bug detection tools has its own strengths and limitations. For example, tools that operate at binary level detect bugs effectively without any source code requirement but at the cost of more execution time overhead. Moreover, solutions such as [30; 32] are not open source and therefore not available for performance comparison. As discussed earlier, the tools that operate at source-code level also require either modified run-time memory allocators or a dedicated compiler driver as it is the case in several existing solutions [30; 13; 35; 16; 17]. Other software based techniques such as LBC [18] have presented much lower performance overhead but they require modifications in the source code, thus presenting compatibility issues.

Unlike SoftBound, our tag-protection solution is a combination of object and pointer-based approach. We create tag marks for each memory object and propagate these tag marks to all the pointers that are associated with that memory object. Moreover, we do not perform

any table-lookup search at run-time which makes our solution more efficient. SoftBound with the CETS [20] extension is capable of detecting both temporal and spatial safety errors but with a much higher performance overhead. On the contrary, lower performance and memory overheads have been reported by our solution when it is compared with the openly available tools under the same experimental set-up. While our solution requires source code for instrumentation, it does not need customized run-time libraries and static analysis. Furthermore, our technique has presented higher detection rate by instrumenting all buffer allocations including buffers allocated inside `struct` type data variables which are left undetected by the existing solutions excluding SoftBound.

3. TAG-PROTECTION

The key concept behind our proposed tag-protection solution, detection of dangling pointer dereferences, current limitations and how it is implemented at compile time have been discussed in this section.

3.1. General Approach and Design

Tag-protection is a compile-time code instrumentation approach and it is based on storing memory objects bounds information in separately allocated tag marks and then inserting tag checking instructions to detect IMAs at run-time. For example, when a memory object created and a certain memory area is reserved for it, our technique creates two tag marks *tag_start* and *tag_end*. The base address and the end address of that memory object are calculated and stored in the *tag_start* mark and the*tag_end* mark respectively. Finally, when that memory object is accessed, the run-time check instructions compare the address being accessed with the bounds stored in its associated tag marks. In the case of any spatial illegal memory access, either buffer underflow or overflow, an alert signal will be generated before terminating the program execution.

Our solution is compatible with C/C++ source code as it does not change the memory layout and tag marks are generated and maintained separately. The source code of `C/C++` based applications is converted into an intermediate representation form which is used to detect each memory allocation, create tag marks and insert run-time checks. The typical memory layout of a C/C++ program is shown in Fig. 1(a). For each memory object, whether it is globally, statically or dynamically allocated, the tag marks are created and memory object bounds are assigned to these tag marks as shown in Fig. 1(b). The record of tag marks along with their associated memory objects is kept separately in a specialized record table as depicted in Fig. 1(c). The tag marks are also created for sub-objects that are defined inside a structure memory object. In that case, our solution stores the sub-object information along with the main memory object in the third column of the record table. It should be noted that the record table is used at the time of code instrumentation only to place relevant check instructions and it is not part of the final executable file.

Unlike other compile-time code instrumentation solutions (such as SoftBound), which use record tables for bounds lookup at runtime, our solution utilizes this record table at compile time only and is deleted before generating final instrumented executable. The record table is created at the time of instrumentation and stores the address bounds. Using this record table, the tag checking instructions are inserted before each load or store instruction detected for the corresponding memory object. This approach has resulted in lower execution time as bound lookup step is performed at compile time and tags are accessed directly at run-time . During tag address comparison, the given memory access will be considered legal only if the address used to access the memory area is greater than the address stored in its *tag_end* mark and less than the address contained by its *tag_end* mark. In the case of any spatial IMA bug, the addresses stored in tag marks will be surpassed and tag check instructions will raise bug alarm and abort the application.

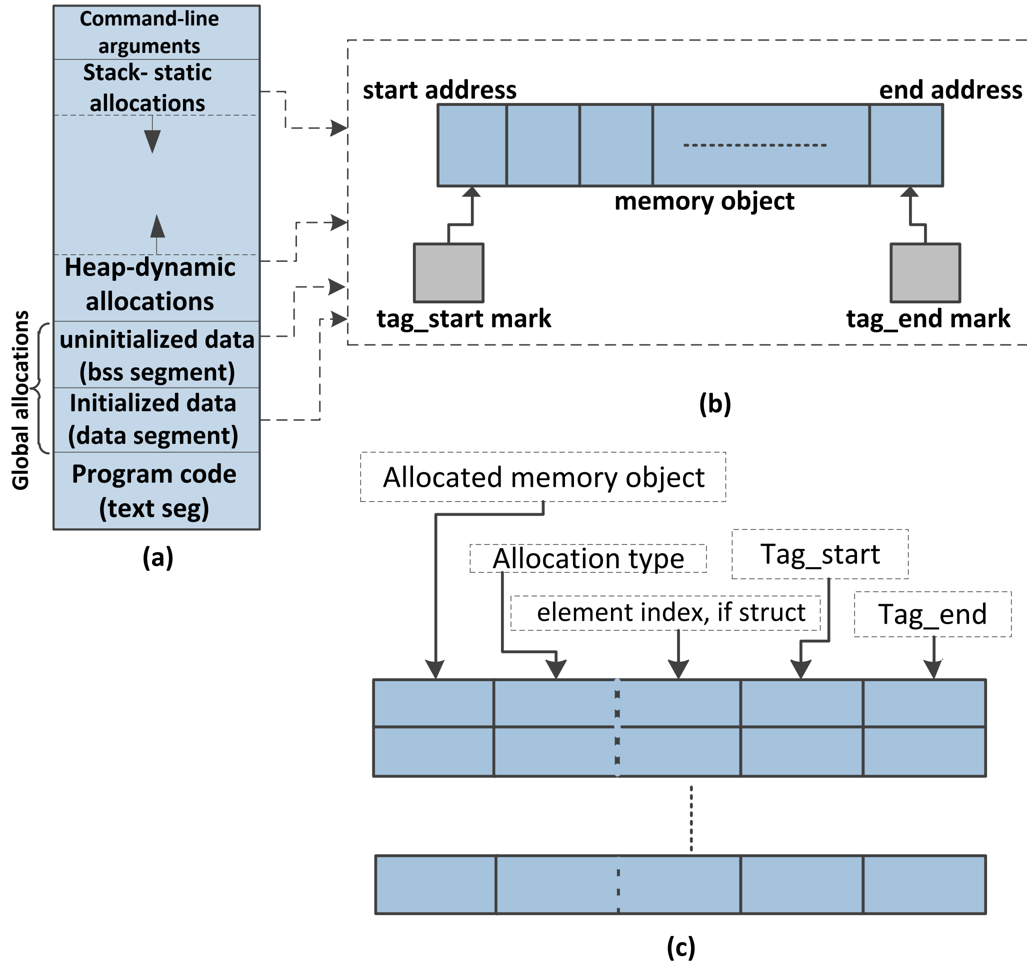


Fig. 1: (a) Typical memory layout of a C program. (b) Memory objects coupled with tag_start and tag_end marks. (c) Record table layout used by tag-protection at the time of code instrumentation.

In order to handle temporal IMA bugs, run-time checks are also inserted to detect dangling pointer dereferences. This is achieved by detecting memory deallocation function calls and assigning a dedicated tag address to the respective tag marks. The memory access will be considered dangling pointer dereference if the tag mark address matches with that dedicated tag address. The design of tag-protection solution is based on the following steps which are also presented in Fig. 2.

3.1.1. Function Duplication Enabling Inter-procedural Tag propagation. When memory objects are accessed by the pointers and these pointers are passed as function arguments then the corresponding tag marks must also be propagated with them as well. One solution is to allocate all the tag marks globally so that the tag marks can be accessed anywhere in the program without modifying the function arguments. However during testing, this approach failed when functions are called recursively and in multi-threaded applications where multiple threads call the same function with different arguments which in turn generate false

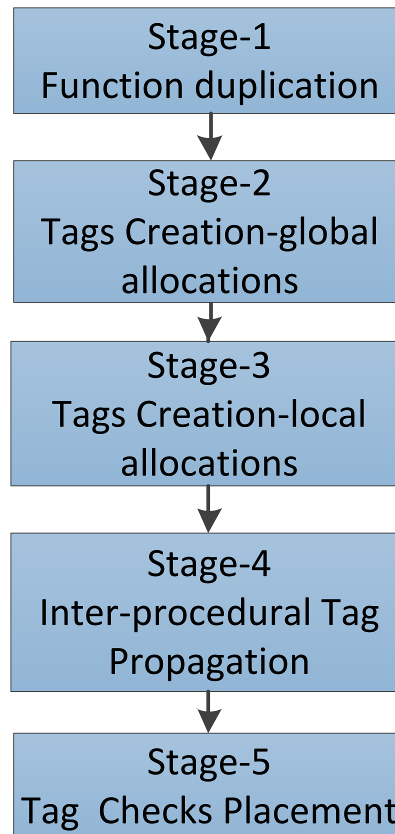


Fig. 2: Flow chart used to implement the tag-protection solution

IMA bug alarm. To handle this problem, the tag marks are created in the same memory segment where corresponding memory object is being allocated. Moreover, the function duplication technique is used to create a copy of the function and additional arguments are added to pass tag_start and tag_end marks for each pointer argument. In order to mark the function being duplicated the actual function name is used along with a unique attribute as presented in Algorithm 1. The details of each function being duplicated are saved in a separate table that is used in later stages of tag-protection pass to replace function calls.

3.1.2. Tag Creation for Global Memory Objects. For globally defined memory objects that are defined statically, such as buffers and structures, the memory is allocated directly at program entry level in *data* and *bss* segments as shown in Fig. 1(a). In order to instrument such global allocations the tag marks are created globally and a dedicated tag address is also allocated to handle dangling pointers dereferences as illustrated in Algorithm 2.

3.1.3. Tag Creation for Local Memory Objects. For the local memory objects that are defined at function level statically, the memory is reserved explicitly on stack and tag marks are also allocated on the stack for such memory objects. Our tag-protection computes the start and end address of such memory objects after which tag_start and tag_end marks are initialized respectively with these addresses. For each memory object pointer that is used to allocate memory dynamically, tag_start and tag_end mark pointers are also allocated

ALGORITHM 1: Stage-1:Function duplication with modified argument list to enable tag marks for functions that take pointers as arguments

Input: Uninstrumented LLVM-IR code α generated through LLVM *llvm-link* command
Output: Instrumented LLVM-IR code β_1 generated through LLVM *opt* command using stage-1 of tag-protection pass
Create fun_map_table
for *each function* fun_def *declared in* α **do**
    if fun_def *has definition in place and arguments contain pointer memory objects* **then**
        Create a function clone fun_def_clone of fun_def. Create two new pointer arguments(*tag_start_arg* and *tag_end_arg*) for each pointer argument in fun_call_clone argument list. Store function name of fun_def_clone and fun_def in fun_map_table
    end
end
Save modified LLVM-IR code as semi instrumented LLVM-IR code β_1

ALGORITHM 2: Stage-2:Tag creation for globally declared memory objects by tag-protection pass

Input: Instrumented LLVM-IR codeβ_1 generated in stage-1 of tag-protection pass
Output: Instrumented LLVM-IR code β_2 generated through LLVM *opt* command using stage-2 of tag-protection pass
Create dedicated tag address *globaltag*; Create Tag_map_table.
for *each global memory object global_var in* α **do**
    if *global_var is not a pointer memory object* **then**
        if *global_var is an array memory object* **then**
            Find start and end address, create *tag_start* and *tag_end* mark pointers and assign them start and end addresses.
            Store memory object *global_var*, its type and tag marks information in Tag_map_table
        end
    else
        Create tag_start and tag_end mark pointers and initialize them with NULL value.
        Store memory object *global_var*,its type and tag information in Tag_map_table
    end
    if *global_var is a structure memory object struct* **then**
        for *each variable struct_var in struct* **do**
            if *struct_var is not a pointer memory object* **then**
                if *struct_var is an array memory object* **then**
                    Find start and end address, create *tag_start* and *tag_end* mark pointers and assign them start and end addresses.
                    Store memory object *global_var*, its type, *struct_var* index and tag marks information in Tag_map_table
                end
            else
                Create *tag_start* and *tag_end* mark pointers and initialize them with NULL value.
                Store memory object *global_var*, its type, *struct_var* index and tag marks information in Tag_map_table
            end
        end
    end
end
Save modified LLVM-IR code saved as semi instrumented LLVM-IR code β_2

and initialized with *NULL*. For such dynamically created objects, the memory is reserved on heap implicitly by calling special memory allocation functions (e.g, `malloc, calloc,`

`realloc, xcalloc` etc.) and starting address is returned to a pointer variable. Our proposed solution intercept such function calls and the start address is assigned to its tag_start mark pointer. The end address of the allocated object is also determined by our pass and it is assigned to its corresponding tag_end mark pointer. To ensure that the tag marks are written atomically, the tag update instructions are placed in the same *basic block*[1] where memory objects are actually allocated. In this way, all the tag marks will be thread-safe for multi-threaded applications. The record of tag marks, along with their initialized values and memory objects, is kept in a separate table. This step of tag creation is achieved by implementing the steps as defined in Algorithm 3.

ALGORITHM 3: Stage-3:Tag creation for memory objects that are declared locally inside each function

---

Input: Instrumented LLVM-IR code β_2 generated in stage-2 of tag-protection pass ; memory map table Tag_map_table; Dedicated tag address *globaltag* ;
Output: Instrumented LLVM-IR code β_3 generated through LLVM *opt* command using stage-3 of tag-protection pass

for *each function definition* fun_def *in* β_3 **do**
    for *each instruction* fun_inst *in* fun_def **do**
        if fun_inst *is a memory object allocation instruction and does not create pointer object* **then**
            if fun_inst *creates an array memory object* **then**
                Find start and end address, create *tag_start* and *tag_end* mark pointers and assign them start and end addresses.
                Store memory object fun_inst, its type and tag marks information in Tag_map_table
            end
        end
        if fun_inst *is memory allocation instruction and creates pointer object* **then**
            Create local *tag_start* and *tag_endmark* pointers and initialize them with NULL value.
            Store memory object instruction fun_inst,its type and tag information in Tag_map_table
        end
        if fun_inst *is heap memory allocation function call instruction* **then**
            Find start and end address of heap allocation. Find respective memory object and retrieve *tag_start* and *tag_end* marks from Tag_map_table.
            Create new STORE instructions to assign start and end addresses to the tag marks.
        end
        if fun_inst *is heap memory deallocation function call instruction* **then**
            Retrieve corresponding tag marks from Tag_map_table and initialize it with *globaltag*
        end
        if fun_inst *is a STORE instruction and updates an allocated memory object pointer address from source operand.* **then**
            Retrieve respective *tag_start* and *tag_end* marks from Tag_map_table for source memory object .
            Retrieve respective *tag_start* and *tag_end* marks from Tag_map_table for destination memory object .
            Create store instructions to copy address values from source to destination tag marks.
        end
    end
end
Save modified LLVM-IR code as an instrumented LLVM-IR code β_3

---

[1]In LLVM-IR code a basic block is simply a container of instructions that execute sequentially. Each basic block can be referenced by instructions such as branches

3.1.4. Inter-procedural tag propagation. As discussed earlier, it is very critical to allocate tag marks in the same memory segment(e.g., heap, stack, bss, data) where memory object is being created. The memory objects can be accessed inside the body of another function through pointers that are passed as arguments at the function call. To handle inter-procedural tag marks propagation, functions containing pointers as arguments are duplicated as shown in Algorithm 4. In order to update function calls for these newly created functions, the function call instructions are detected by our pass and replaced with new instructions so that tag marks can be propagated separately without changing the data-flow of the application under instrumentation.

ALGORITHM 4: Stage-4:Inter-procedural tag propagation

Input: Instrumented LLVM-IR code β_3 generated in stage-3 of tag-protection pass ; memory map table Tag_map_table and fun_map_table ;
Output: Instrumented LLVM-IR code β_4 generated through LLVM *opt* command using stage-4 of tag-protection pass
for *each function definition fun_def in β_3* **do**
  for *each instruction fun_inst in fun_def* **do**
    if *fun_inst is a function call instruction* **then**
      Get function name fun_call_name being called by fun_inst instruction
      if *function called by fun_inst is present in fun_map_table.* **then**
        Retrieve respective fun_def_clone from fun_map_table.
        Create new function call instruction fun_inst_new pointing to fun_def_clone.
        for *each pointer function argument ptr_arg in fun_inst* **do**
          Retrieve respective *tag_start* and *tag_end* marks from Tag_map_table and add them to fun_inst_new argument list.
        end
        Remove fun_inst instruction and replace it with fun_inst_new.
      end
    end
  end
end
Delete memory map table *fun_map_table* Save modified LLVM-IR code as an instrumented LLVM-IR code β_4

3.1.5. Tag Checks Placements. In final step, the tag checks are created by following the steps as shown in Algorithm 5. Memory read and write accesses are performed through *LOAD* and *STORE* operations respectively at LLVM-IR level. Our tag-protection pass detects such instructions and uses record table to locate the memory object pointer and tag marks to be accessed. Tag check instructions, to compare the start and end address of memory object with its associated tag marks, are then inserted before each load and store instruction to detect spatial IMA bug. Memory accessed through LLVM intrinsic functions (e.g., memset, memcpy) are also instrumented and accordingly by detecting memory objects being accessed and interesting tag checking instructions accordingly. Furthermore, the tag marks are also compared with the dedicated tag address to detect any dangling pointer deference. In order to explain the threat model clearly, an example C code with possible IMAs bugs is presented in Fig. 1. The code consists of a function which simply copies strings into two local buffers and prints the results before exiting the function. Here at line 2, a *buffer2* of length *MAX_size* is allocated on stack and at line 3, a heap memory of the same size is allocated for *buffer*. The function calls on line 6 and 7 may initiate spatial IMA bugs through heap and stack overflow if the length of input strings, passed to the main function at run time, exceeds *MAX_size*. The *memcpy* function at line 9 also causes dangling pointer dereference error as the respective memory area is deallocated at line 8 and hence can no

ALGORITHM 5: Stage-5:Tag checks placement.

Input: Instrumented LLVM-IR code β_4 generated in stage-4 of tag-protection pass ; 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 pass
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 *fun_inst* instruction and store the read value in *before_fun*.
                Read address location next to tag_end address after *fun_inst* instruction and store the read value in *after_fun*.
                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 instruction and updates a memory object* **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 dangling pointer dereference check. compare *tag_end* with the *globaltag*.
            Perform address comparison checks: *address_tobe_accessed* with the *tag_start* and *tag_end*.
        end
        if *fun_inst is a LOAD instruction and read from allocated memory object* **then**
            Retrieve respective *tag_start* and *tag_end* marks from *Tag_map_table* and get address to be accessed by the *fun_inst* instruction.
            Perform dangling pointer dereference check. compare *tag_end* with the *globaltag*.
            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 a final instrumented LLVM-IR code γ

longer be used. It is not possible to detect these kinds of IMAs through static analysis as the user inputs are not known at compile time. The details of instrumenting this example code are presented in the following subsection.

3.2. Handling Pointer Operations:

If a pointer is derived from another pointer, the tag mark pointer associated with the actual object pointer must also be propagated. Our proposed technique detects store instructions, at LLVM-IR level, that are used to pass address values from one pointer object to another pointer object. To illustrate this, consider the line 5 in Fig. 1, where pointer object *ptr* passes its contained address value to another pointer object *buffer*. In that case, the extra instructions are inserted by our pass to copy the tag mark pointer of *ptr* to the tag mark pointer of *buffer*.

Our solution instruments main memory objects and the sub-memory objects that are being allocated inside the *structs* type memory objects and it also instruments the sub-

Listing 1: An example C code with illegal memory accesses

```
1 :int funcall(int argc , char **argv){
2 :   char *buffer,*ptr,buffer2[MAX_size];//stack alloc
3 :   ptr=(char *)malloc(MAX_size);//heap alloc
4 :   if(ptr==NULL) exit(1);
5 :   buffer=ptr;
6 :   strcpy(buffer,argv[1]);/*possible heap overflow*/
7 :   strcpy(buffer2,argv[2]);/*possible stack overflow*/
8 :   free(buffer);
9 :   memcpy(ptr,buffer2,MAX_size) /*dangling pointer dereference*/
10:   printf("String one :%s\n,buffer")/*dangling pointer deref*/
11:   printf("String two :%s\n,buffer2")
12: }
```

memory object that is being allocated inside another sub-memory object(such as linked lists).

3.3. Detecting Dangling Pointer Dereferences

To handle dangling pointer dereferences, our solution detects calls to dedicated memory deallocating functions such as *free*. The memory object to be deallocated is identified along with its tag mark pointer. Extra instructions are then inserted to assign dedicated tag address, *globaltag*, to the corresponding tag mark pointer of the memory object to be deallocated as mentioned in Algorithm 3. The check instructions, being inserted for each $LOAD$ or $STORE$ operation as mentioned in Algorithm 5, measure the address contained by the tag mark pointer and perform address comparison. If the memory object pointer has not been reallocated and it is used to access the deallocated memory, the tag mark pointer still points to the dedicated tag address,*globaltag*. In that case, check instructions being inserted by our tag-protection pass will generate the dangling pointer dereferences signal to terminate the application.

For example, consider a call to *memcpy* function at line 9 in Listing 1 where pointer object *ptr* is used. This memory object is deallocated at line 8 through pointer *buffer*. Our tag-protection solution will initialize its tag mark pointer with the dedicated tag address, *globaltag*. Check instructions inserted for the tag mark pointer of *ptr* before the function call *memcpy* will raise dangling pointer dereference bug alarm and terminate the application before this function call executes.

3.4. Implementation

The proposed tag-protection approach operates at the source-code level and it is loaded as an instrumentation pass at compile time. The current implementation is based on the LLVM v3.4 compiler infrastructure as shown in Fig. 3. The *Clang* compiler is used to compile C/C++ source file and generate Intermediate Representation (LLVM-IR) code. The LLVM Linker (llvm-link) is then used to link and generate a single LLVM-IR code file before running the tag-protection pass. In order to have minimum overhead, this pass is placed at the end of optimization pipeline and instruments only those memory operations that sustains other optimizations implemented by the LLVM Optimizer (opt). For instance, the memory operations such as accesses to local stack variables and objects created through LLVM code generator (e.g., debug information and metadata) will not be instrumented by our pass as these will be optimized out by the LLVM during pre-processing at compile time. After the LLVM-IR code has been instrumented by our tag-protection pass, it is passed again through LLVM optimization pipeline in order to simplify the tag marks propagation and checks. Furthermore, our tag-protection pass is independent of any specific Instruction Set

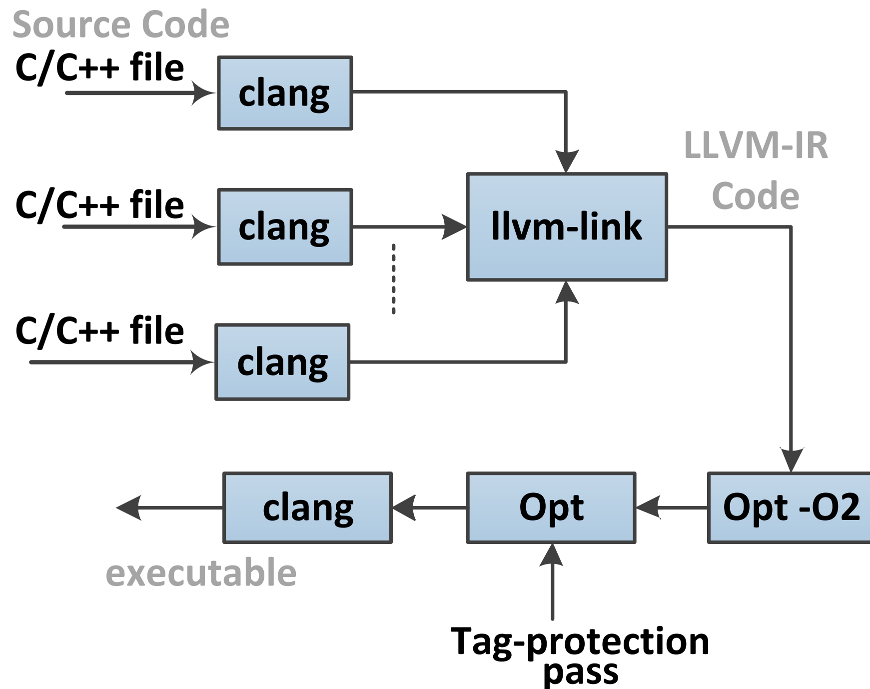


Fig. 3: Tag-Protection implementation block diagram based on LLVM v3.4 compiler framework

Architecture (ISA) as it is executed on LLVM's target-independent intermediate representation form. In Appendix A different C codes, taken from actual applications, have been instrumented and presented in LLVM-IR form.

3.5. Limitations

Our proposed solution works primarily as an Intrusion Detection System (IDS) and prevents further damage by terminating the application's execution. The alert signal is generated followed by an exit call, whenever an overflow occurs either due to unintentional error or as a result of deliberate attack. Moreover, the current implementation of tag-protection pass requires C/C++ based applications source code and does not support instrumentation of code generated at run-time by dynamically interpreted languages (such as Python, Ruby, Perl, etc.). This means that the type of each memory object must be known at compile-time in order to detect its start and end address.

Calls to pre-compiled library functions where source code is not available (e.g., `memcpy, strcpy, sscanf` etc.) are also identified by our tag-protection pass. In such cases, it is not possible to insert *tag address check instructions*. Alternatively, our tag-protection pass inserts one *tag value check instructions* after such function calls by detecting memory objects passed as function arguments and loading tag mark values as defined in Algorithm 5. Any overflow that occurs as a result of the sequential write operation will overwrite the memory pointed by the tag mark. Such overflows will be eventually detected, on function return, by tag value check instructions placed after function call instruction. If the overflow attack occurs at the same memory location with different values over a period of time (such as brute-force or adaptive attacks), the tag mark can be overwritten with its initial value. In

such cases, our address checking mechanism can detect these kinds of attacks only if the complete source code is provided.

Our tag-protection pass will not be able to detect any read overflow that occurs during execution of the un-instrumented pre-compiled library functions. Furthermore, our solution currently does not instrument functions variable argument functions, therefore any memory safety violations occurring inside such functions will not be reported by our solution.

4. EVALUATION

To evaluate the effectiveness and overhead of our tag-protection pass, we have instrumented applications from various benchmark suites with our tag-protection pass. All applications are compiled and instrumented using *Clang* with *-O2* optimization level. The instrumented applications are executed in 64-bit mode on a DELL OPTIPLEX 780 machine with Intel core i5-2400 CPUs and 4GB RAM running Ubuntu 12.04 with kernel 3.11.0.26.

Please note that, for any solution that performs compile-time transformations, the changes made to the source code must not affect the actual flow of data and the generated binary executable should produce the expected output. In our case, the tag marks and the check instructions inserted by the tag-protector pass should not and will not generate any false alarms. In addition, inserting and executing extra checks always result in performance and memory overhead. In order to achieve an efficient solution, these overheads should be manageable.

4.1. Effectiveness

To measure the effectiveness of our technique, we have created different test programs, initializing various IMA bugs caused by illegal array indexing and invalid pointer arithmetic operations. These programs are instrumented with our tag-protection pass and all the bugs detected successfully. To test our solution on real-world applications that have been reported with buffer overflow vulnerabilities, C language based applications from BugBench benchmark suite [22] are compiled and instrumented with tag-protection pass. This benchmark suite [22] has a set of applications that contain various known software defects including buffer overflows, stack smashing, double frees, uninitialized reads, data races and atomic violations. We have selected only those applications that have buffer overflows as our solution targets these kinds of memory vulnerabilities. These applications are then executed using input sets, triggering each known IMA bug. Our proposed solution detected all the bugs successfully as presented in table I. The second column of this table represents the total lines of code compiled for each application and the third column provides the line number of the bug location in the given file. Many open-source tools provide a wide range of test programs. To further check the validity of our solution, it is also tested successfully on a set of diverse test-bed programs that come along with SAFECode source files [25].

We have also assessed our tag-protection pass using publicly available Wilander and Nikiforakis' benchmark software, runtime intrusion prevention evaluator (RIPE) [47]. Various buffer-overflow vulnerabilities depending on the technique used to overflow the buffer, the kinds of attacks performed and the location of the buffer to be overwritten, have been covered by RIPE. For instance, RIPE covers four memory locations:Stack, Heap, BSS, and Data segment to allocate a buffer to be overflowed and uses a return address, old base pointer, function pointer, *longjmp* buffers and buffers inside the *structs* to as code target pointers. Our solution provides 100% accuracy by successfully detecting all the overflows. The execution of the attack code is prevented by terminating the program execution. This software is currently supported for 32-bit architecture so we have made some modifications to enable its execution on 64-bit architecture We have tested our tag-protection pass and other publicly available solutions on 64-bit Ubuntu 12.4 in order to measure detection rates. The comprehensive comparison with the existing countermeasures that are available openly, is presented in Table II. CRED and SAFECode failed to prevent direct stack/B-

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

Application	Lines of code (LoC)	Bug location	Bug type	Detected
bc-1.06	14.4k	storage.c:177	heap overflow	yes
bc-1.06	14.4k	util.c:577	heap overflow	yes
bc-1.06	14.4k	bc.c:1425	global overflow	yes
gzip-1.2.4	8.1k	gzip.c:457	global overflow	yes
man-1.5h1	4.1k	man.c:978	global overflow	yes
ncompress	1.9k	compress.c:896	stack overflow	yes
polymorph-0.40	0.7k	polymorph.c:120	global overflow	yes
polymorph-0.40	0.7k	polymorph.c:193	stack overflow	yes
squid-2.3	93.5k	ftp.c:1024	heap overflow	yes

Table II: Comprehensive Comparison with existing publicly available countermeasures when tested on Ubuntu 12.4 64-bit architecture

Technique used	Buffer Overflow Location					Dangling Pointer Detection	Detection rate for RIPE [47]
	Stack	Heap	Data segment	Bss segment	Structs[2]		
gcc compiled no protection	×	×	×	×	×	×	0%
Libsafe [36]	✓	×	×	×	×	×	7%
StackShield [38]	✓	×	×	×	×	×	36%
ProPolice [37]	✓	×	×	×	×	×	40%
LibsafePlus+ TIED [39; 40]	✓	✓	×	×	×	×	70%
CRED [11]	✓	✓	✓	✓	×	×	60%
SoftBound [20]	✓	✓	✓	✓	✓	✓	36.26%
SAFECode [13]	✓	✓	✓	✓	✓	✓	68.89%
AddressSanitizer [35]	✓	✓	✓	✓	✓	✓	85.5%
Proposed Tag-Protector	✓	✓	✓	✓	✓	✓	100%

SS/databased overflows toward function pointers, longjmp buffers, and static arrays defined within structs[2] for many library functions such as `sprintf(), snprintf(), sscanf(), and fscanf()`.

SoftBound has presented the detection rate of 36% only and on close examination of the RIPE source code and SoftBound transformation pass, it is found that the SoftBound fails to insert overflow checks for arrays that are being defined inside structs[2] based memory objects. Under specific circumstances, where direct out-of-bound access is made through absolute indexing, our proposed tag-protection solution can detect such IMA bugs whereas other existing tools like Valgrind and AddressSanitizer leave such bugs undetected.

4.2. Performance Overhead

The performance overhead is defined as the percentage increase in the execution time of instrumented binaries as compared to the execution time of the binaries built using the standard *Clang* compiler with *-O2* optimization level. As our solution terminates the application's execution as soon as it detects any overflow or dangling pointer dereference, the

[2]Buffers allocated within a struct data type declaration e.g

```
typedef struct struct_data{
    char buffer1[256];
    int buffer2[128];
     }buffdata;
```

performance overhead can not be measured using applications from BugBench benchmark suite [22] or RIPE [47]. To measure the actual performance overhead of our solution, we have instrumented all the C/C++ based applications from SPEC CPU2006 benchmark suite [23] comprising over 1.11M lines of code. The applications in this benchmark are real-world applications and are assumed to be virtually bug-free due to their general usage and no bugs have been reported so far in the community. All the applications instrumented by tag-protection pass have been executed successfully without generating any IMA bug alert which further proves the effectiveness of our solution.

The applications are also instrumented with similar openly available solutions (such as AddressSanitizer and SAFECode) and executed on the same machine under similar experimental set-up. Our tag-protection pass on average incurs 26.42% increase in execution time which is still far lower than other solutions. Each application has been executed five times to get the average execution time. The complete performance overhead results along with the error bars, presenting standard deviation in the execution time, are shown in Fig. 4.

To compare the performance overhead of our solution with the SoftBound+CETS [20], the latest available source code [48] is downloaded and installed as per instructions provided. Various benchmark applications from SPEC CPU2006 and SPEC CPU2000 are instrumented and execution time overhead comparison is presented in Fig. 5. To have a fair comparison, this figure only presents execution time overhead for those applications as reported in the SoftBound+CETS [20] and as well as the percentage increase in the execution time of tag-protection enabled applications is shown in this figure. Our solution has presented lower performance overhead except two applications (*lbm, crafty*) where the difference in percentage increased is still less than 3%.

In order to compare execution time overhead with SAFECode, the latest available source code [25] is downloaded and installed as per instructions. The whole-program analysis feature of SAFECode is also enabled through *libLTO* plug-in that performs these analyses and transformations. Fig. 6 presents performance overhead comparison in terms of percentage increase in execution time when C/C++ based benchmark applications from SPEC CPU2006 are executed after instrumenting with our proposed tag-protection pass, SAFECode and AddressSanitizer respectively. The SAFECode failed to instrument two applications, *403.gcc* and *473.omnetpp*, which are not included in this figure. Furthermore, two applications, *477.dealll* and *483.xalancbmk*, have generated false bug alarms when executed after instrumenting with SAFECode. This is probably due to the fact that certain optimizations such as type-safe load/store check elimination, static array bounds checking, automatic pool allocation and fast pool run-time checks for loads and stores, and the use of multiple splay trees, have not been enabled in the source code that is available online [25].

Intel Pointer Checker [45] tool has also been evaluated by installing the trail version as it is not available publicly. Applications from SPEC CPU2006 benchmark suite are instrumented using this tool by following the instructions carefully but it failed to instrument many applications completely and has presented very large performance overhead.

To make a fair comparison with similar existing solutions which are not available publicly such as PAriCheck [17], BaggyBound [16] and WIT [14], we have instrumented applications from the Olden [21] and SPEC CINT2000 [49] benchmarks as the indicated solutions have reported the performance overhead for these benchmark suites. Furthermore, our solution has detected a heap overflow bug in application *"em3d"* from Olden benchmark, when executed with same input set as used by PAriCheck. The same bug is also detected by AddressSanitizer and Valgrind's MemCheck tool whereas PAriCheck and other solutions have not reported this bug. Our solution has presented an average overhead of 26.64% for SPEC CINT2000 and 19.98% for Olden applications when compiled and executed with similar settings whereas the BaggyBound has reported 60% overhead for SPEC CINT2000 applications. Only PAriCheck [17] has reported lower execution time overhead than our technique but their solution fails to detect dangling pointer dereferences and overflows in

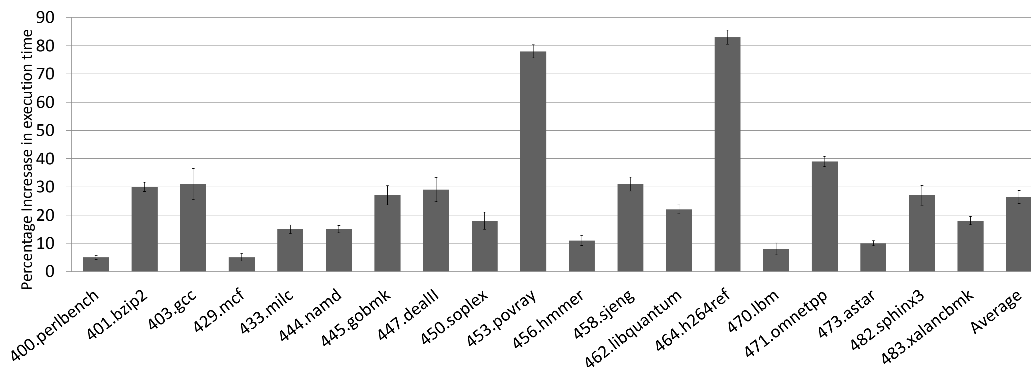


Fig. 4: Performance overhead for SPEC CPU2006 Benchmark applications

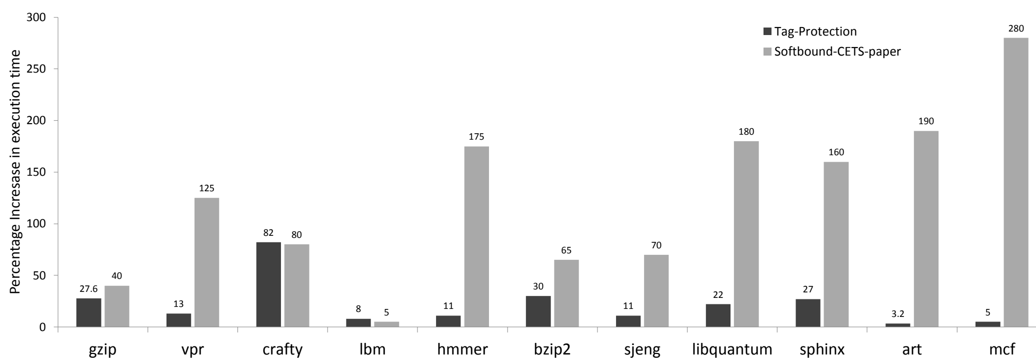


Fig. 5: Performance overhead comparison with SoftBound-CETS for various applications from SPEC CPU2006 and SPEC CPU2000 benchmark suite.

pointer buffers declared inside structure objects. WIT [14] has also reported lower execution time but their solution has limited functionality as they provide protection against illegal writes only.

4.3. Memory Overhead

To measure peak memory usage for the instrumented applications we have examined the *VmPeak* field from */proc/(pid)/status* file. From the table III, it is clear that our tag-protection solution has presented minimal memory utilization overhead.

Under specific conditions, where memory usage is the main constraint, this overhead can be further reduced by disabling tag error reporting and allocating tag marks for array memory objects only. Table IV summarizes the increase in binary size when applications from SPEC CPU2006 benchmark suite are instrumented with tag-protection pass and other existing solutions under similar experiment setup as explained earlier. On average, the binary size is increased by 5.09x. This increase is still lower than the existing solutions as presented in this table.

4.4. Multi-threaded Applications

To evaluate our solution for multi-threaded applications in a multi-core environment, the PARSEC v2.1 benchmark suite [24] is used. Through successful instrumentation and execution of benchmark applications, it is shown that the our proposed solution is thread-safe

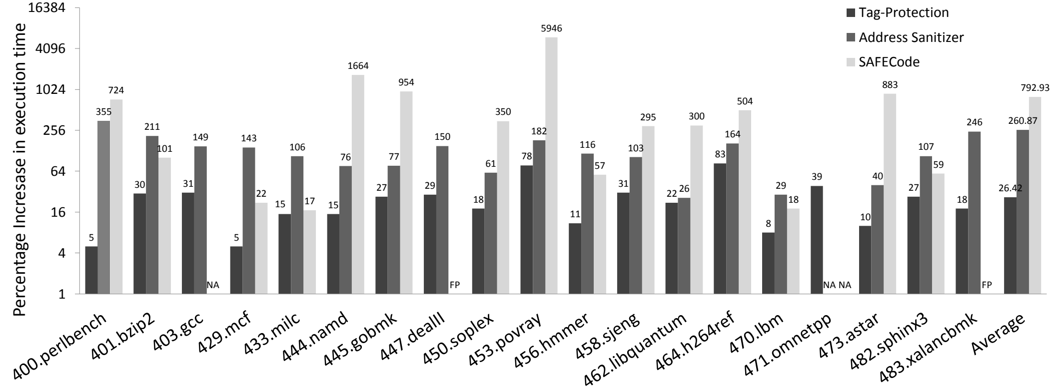


Fig. 6: Performance overhead comparison for SPEC CPU2006 Benchmark applications with existing solutions

Table III: Increase in memory utilization for instrumented SPEC CPU2006 Benchmark applications

Application	Uninstrumented (KB)	Instrumented (KB)	Increase (KB)
400.perlbench	9752	14304	4552
401.bzip2	69536	70168	632
403.gcc	25272	43332	18060
429.mcf	390756	39040	84
433.milc	16344	17160	816
444.namd	59060	59444	384
445.gobmk	36176	40596	4420
447.dealII	31060	50260	19200
450.soplex	49516	51524	2008
453.povray	16488	26168	9680
456.hmmer	8176	10236	2060
458.sjeng	186212	187252	1040
462.libquantum	2984	8944	5960
464.h264ref	35672	39592	3920
470.lbm	426132	426824	692
471.omnetpp	20936	25412	4476
473.astar	21304	21568	264
482.sphinx3	40512	46440	5928
483.xalancbmk	26216	47164	20948
Total	**1472104**	**1577228**	**105124**

and it is suitable for multi-core systems. So far seven applications from this benchmark are instrumented successfully with our prototype solution whereas other applications require changes in the compiler drivers to integrate our tag-protection pass completely in order to achieve complete instrumentation. These issues will be addressed in our future work. The simulation results in terms of percentage increase in execution time are presented in Fig. 7.

In order to get performance comparison with the SAFECode and AddressSanitizer, for the multi-threaded applications, the instrumented binaries are thus executed using the same machine configuration as defined earlier. The performance comparison and execution time overhead is presented in Fig. 8. From these results, it is clear that even in the worst-case scenario, the tag-protection presents 25.4% performance overhead for which is the least as compared to the existing solutions. As shown in this figure,

Table IV: Increase in binary size for instrumented SPEC CPU2006 Benchmark applications.

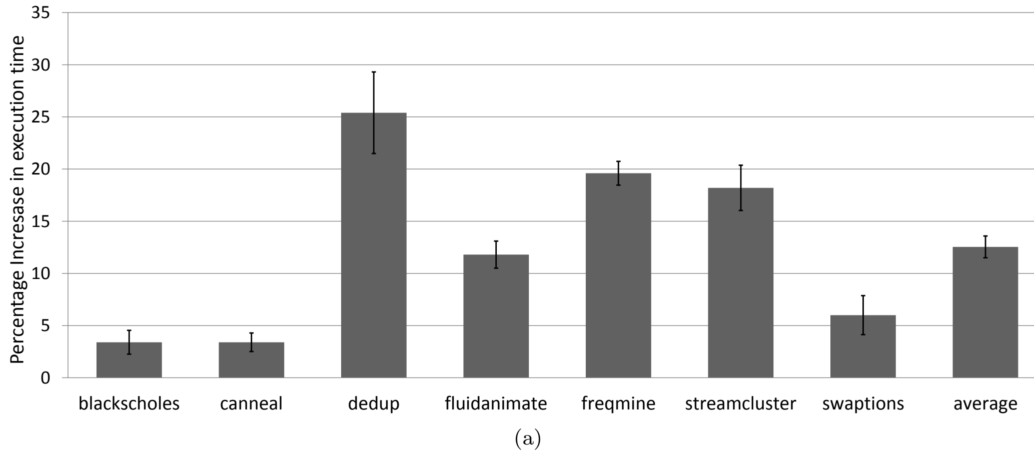
Application	Tag-Protection	Address-Sanitizer	SAFECode
400.perlbench	5.08x	10x	7.59x
401.bzip2	2.26x	19.35x	5.98x
403.gcc	6.2x	4.28x	NA
429.mcf	2.05x	60.65x	20.82x
433.milc	2.76	11.96x	4.89x
444.namd	5.94x	9.12x	7.15x
445.gobmk	1.5x	3.01x	2.51x
447.dealII	6.36x	10.22x	14.74x
450.soplex	16.07x	12.6x	17.79x
453.povray	6.12x	5.17x	4.81x
456.hmmer	3.13x	8x	3.6x
458.sjeng	3.23x	11.63x	4.02x
462.libquantum	2.35x	25.93x	7.96x
464.h264ref	3.38x	6.25x	2.55x
470.lbm	12.7x	56.65x	14.39x
471.omnetpp	5.65x	NA	NA
473.astar	2.75x	26.4x	10.2x
482.sphinx3	3.12x	4.05x	9.52x
483.xalancbmk	6.02x	7.86x	14.39x
Average	**5.09x**	**16.27x**	**8.99x**

one application, *dedup*, has generated false bug alarm while when an application, *swaptions*, failed to complete its execution in the correct manner as required when compiled with SAFECode. One application, *freqmine*, fails to complete its execution when instrumented with AddressSanitizer. On the other hand, our technique does not result in any false alarms and all the benchmark applications generated outputs as expected.

Unlike AddressSanitizer, which uses compact shadow mapping and customized run-time libraries, the tag-protection allocates 8-bit tag mark against each memory allocation which relatively uses less memory. Contrary to SAFECode, our approach does not require static analysis and customized compiler driver which increases the compilation time. The instrumentation done by SAFECode has resulted in slower execution time. This effect is visible from our results where binaries instrumented through SAFECode bear huge performance overhead. Furthermore, in our technique the tag checks are inserted only for those store instructions that write to the allocated memory areas, skipping pointer copy instructions, which result in less execution time overhead.

5. FUTURE WORK

The proposed solution has been designed for C/C++ based applications targeting embedded systems as these languages provide a powerful set of features such as low memory footprint, little run-time support, low-level direct memory accesses and arithmetic operations through pointers. Real-time embedded systems having hard deadlines can not afford the luxury of executing extra instructions along with the actual code. For such systems, the proposed solution can be improved further by exploring the possibility of executing run-time checks, being inserted by the tag-protection pass, through dedicated hardware module. This hardware module can be designed to run in parallel within the main processing core. In this way, the performance overhead will be reduced approximately to zero. For example, for FPGA-based embedded systems, such hardware module can be implemented by designing customized instruction-set architecture in order to differentiate between run-time checks and application instructions.



(a)

Fig. 7: Performance overhead in terms of percentage increase in execution time for PARSEC v2.1 Benchmark applications

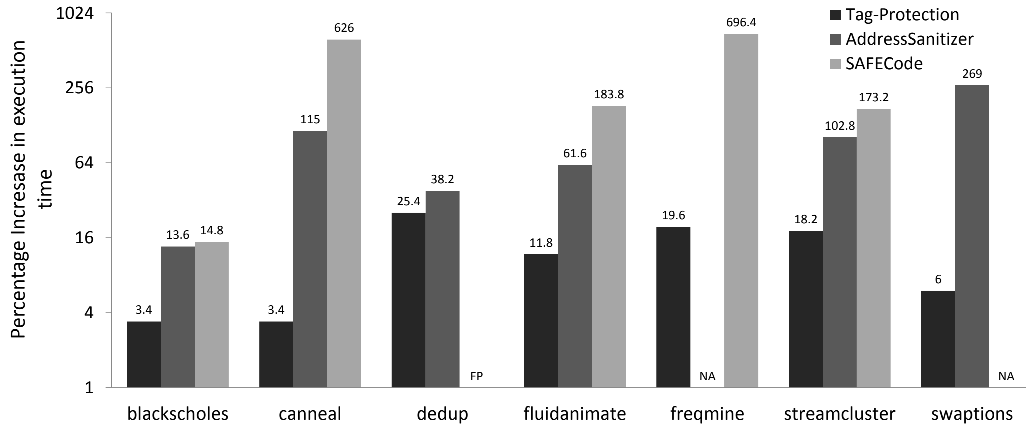


Fig. 8: Performance overhead comparison for PARSEC v2.1 Benchmark applications with existing solutions

The current implementation of the tag-protection pass requires single LLVM-IR code file, as shown in Fig. 3. The LLVM linker (*llvm-link*) has failed to generate LLVM-IR code file for remaining applications. In our future work, the TPP will be integrated completely within the compiler framework in order to compile and generate instrumented executable files.

6. CONCLUSION

In this paper, a fast and effective tag-protection solution is presented to detect illegal memory accesses in the applications that are written in C/C++. It is implemented as an instrumentation pass using LLVM and operates at the source-code level. The effectiveness of the proposed solution is tested using several benchmarks and test applications. Through various experimental results, it is shown that our solution has less performance overhead when compared with the publicly available tools. The applications instrumented with the tag-protection pass incur only 26.42% and 12.48% performance overhead on average for the SPEC CPU2006 and the multi-threaded PARSEC v2.1 benchmark suites, respectively. Fur-

thermore, through the execution of instrumented multi-threaded applications, it is shown that our proposed solutions are thread-safe and the performance overhead is minimal when these applications are executed with a higher number of threads in a multi-core system. This demonstrates that our proposed technique is a scalable solution for multi-core environments as well.

REFERENCES

[1] Eric Chien and Péter Ször. Blended attacks exploits, vulnerabilities and buffer-overflow techniques in computer viruses. *Virus*, 1, 2002.

[2] Yves Younan. 25 Years of Vulnerabilities: 1988-2012, 2013.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[4] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[5] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[6] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[7] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 69–80, New York, NY, USA, 2003. ACM.

[8] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, volume 32. Washington DC, 2001.

[9] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, volume 98, pages 63–78, 1998.

[10] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, pages 13–26. Citeseer, 1997.

[11] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[12] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. ACM.

[13] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 144–157, New York, NY, USA, 2006. ACM.

[14] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE, 2008.

[15] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *ACM Sigplan Notices*, volume 44, pages 245–258. ACM, 2009.

[16] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[17] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R Sekar, Frank Piessens, and Wouter Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 145–156. ACM, 2010.

[18] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 135–144, New York, NY, USA, 2012. ACM.

[19] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[20] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[21] Martin Christopher Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, Princeton University, 1996.

[22] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, pages 1–5, 2005.

[23] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[24] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[25] SAFECode. Download:SAFECode FOR LLVM 3.2, 2006.

[26] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 103–114, New York, NY, USA, 2008. ACM.

[27] David Chisnall, Colin Rothwell, Robert NM Watson, Jonathan Woodruff, Munraj Vadera, Simon W Moore, Michael Roe, Brooks Davis, and Peter G Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–130. ACM, 2015.

[28] Intel's_MPX. Intel Memory Protection Extensions (Intel MPX) Enabling Guide.

[29] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGARCH Comput. Archit. News*, 32(5):85–96, October 2004.

[30] Ioannis Doudalis, James Clause, Guru Venkataramani, Milos Prvulovic, and Alessandro Orso. Effective and efficient memory protection using dynamic tainting. *Computers, IEEE Transactions on*, 61(1):87–100, 2012.

[31] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. *SIGPLAN Not.*, 47(7):121–132, March 2012.

[32] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. Citeseer, 1991.

[33] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[34] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC*, volume 2012, 2012.

[36] Arash Baratloo, Navjot Singh, and Timothy Tsai. Libsafe: Protecting critical elements of stacks. *White Paper http://www. research. avayalabs. com/project/libsafe*, 1999.

[37] Hiroaki Etoh and Kunikazu Yoda. Propolice: Improved stack-smashing attack detection. *IPSJ SIGNotes Computer Security (CSEC)*, 14:25, 2001.

[38] Vendicator. Stackshield, 2001.

[39] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *USENIX Security Symposium*, pages 45–56, 2004.

[40] Kumar Avijit and Prateek Gupta. Binary rewriting and call interception for efficient runtime protection against buffer overflows. *Software: Practice and Experience*, 36(9):971–998, 2006.

[41] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 269–280, June 2006.

[42] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of Twenty-first ACM*

SIGOPS Symposium on Operating Systems Principles, SOSP '07, pages 351–366, New York, NY, USA, 2007. ACM.

[43] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 117–126, New York, NY, USA, 2004. ACM.

[44] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. ACM.

[45] PointerChecker. Pointer Checker:Easily Catch Out-of-Bounds Memory Accesses, 2012.

[46] Yutaka Oiwa. Implementation of the memory-safe full ansi-c compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 259–269, New York, NY, USA, 2009. ACM.

[47] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime intrusion prevention evaluator. In *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*. ACM, 2011.

[48] SoftBound+CETS. SoftBound+CETS: sourcecode, 2014.

[49] SPEC. SPEC CPU2000: CINT200, 2000.

Appendix:

A. LLVM INTERMEDIATE REPRESENTATION (LLVM-IR) CODE

A simplified section of a C function extracted from one of the Bugbench [22] applications is shown in Listing 2. Here, for some user inputs, the value of *count* exceeds the *index* value resulting in heap overflow error. More details about this overflow bug are available in [22]. The uninstrumented LLVM-IR code generated by *Clang* is shown in Listing 3. After generating this LLVM-IR code, the un-instrumented code is processed through tag-protection pass, based on Algorithms (1-5), to generate final instrumented LLVM-IR code with necessary run-time checks as underlined in Listing 4. The tag address comparison checks, inserted at lines 22 to 37, by the tag-protection pass will detect and report any overflow dynamically.

The record table as shown in Fig. 1-c has been to create and store tag marks and generate tag checking instructions. For instance, to place tag check instruction at lines 24 in Listing 4, the instructions at line 5,6,11 and 14 will be stored in the first, second, fourth and fifth columns of record table respectively. As the memory object is not of *struct* data type, the third column of record table for this object will not be used. On reaching line 13 of un-instrumented code as shown in Listing 3, the tag-protection pass will use the given entries of the record table to create all the overflow check instructions (as presented by lines 38-49 of Listing 4). After processing the last line of the un-instrumented code, the tag-protection pass will delete the record table and generate the final executable file.

Calls to pre-compiled library functions where source code is not available (e.g., `memcpy`, `strcpy`, `sscanf` etc.) are also identified by our tag-protection pass. In such cases, it is not possible to insert tag address checking instructions. Alternatively, our tag protection pass inserts tag value check instructions after such function calls by detecting memory objects passed as function arguments and loading their respective tag values. For example, consider a very simple code, as shown in Listing 5, that copies user input to a global buffer. The un-instrumented LLVM-IR code generated by *Clang* is shown in Listing 6 whereas the transformed LLVM-IR version as instrumented by the tag-protection pass is shown in Listing 7. Any overflow that occurs during the execution of these functions will overwrite the tag mark which will be eventually detected on function return by tag value check instruction placed at lines 28 to 42.

Listing 2: Heap overflow example from one of the Bugbench applications.

```
//buffer is declared as global array of pointers
1:  buffer = malloc(MAX_size*sizeof(char *));
2:  for(int index=0;index<count;index++)
3:          buffer[index]=NULL;//store instruction
```

Listing 3: LLVM-IR code (Un-instrumented) for C code presented in Listing 2

```
   //Lines 1-3 represent LLV-IR code for line 1 of C code in Listing 2
1 :  %call = call noalias i8* @malloc(i64 800) #2
2 :  %0 = bitcast i8* %call to i8**
3 :  store i8** %0, i8*** @buffer, align 8
4 :  br label %for.cond
   //Lines 5-8 represent LLV-IR code for line 2 of C code in Listing 2
5 : for.cond:     ; preds = %for.inc, %entry
6 :  %index.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
7 :  %cmp = icmp slt i32 %index.0, 101
```

```
8 :  br i1 %cmp, label %for.body, label %for.end
   //Lines 9-14 represent LLV-IR code for line 3 of C code in Listing 2
9 : for.body:                     ; preds = %for.cond
10:  %idxprom = sext i32 %index.0 to i64
11:  %1 = load i8*** @buffer, align 8
12:  %arrayidx = getelementptr inbounds i8** %1, i64 %idxprom
13:  store i8* null, i8** %arrayidx, align 8
14:  br label %for.inc
15: for.inc:
; preds = %for.body
16:  %inc = add nsw i32 %index.0, 1
17:  br label %for.cond
18: for.end:
; preds = %for.cond
```

Listing 4: LLVM-IR code (Instrumented with tag-protection pass) for C code presented in Listing 2

```
   //Dedicated Tag address creation for dangling pointer checks
1 :  %0 = tail call noalias i8* @malloc(i64 2)
2 :  store volatile i8* %0, i8** @globaltag, align 8
3 :  %1 = load volatile i8** @globaltag, align 8
4 :  store volatile i8 107, i8* %1, align 1
5 :  %call = tail call noalias i8* @malloc(i64 800) #1
6 :  %2 = bitcast i8* %call to i8**
7 :  store i8** %2, i8*** @buffer, align 8
   //Tag marks creation
8 :  %3 = ptrtoint i8* %call to i64
9 :  %add3 = add i64 %3, 800
10:  %t_tag4 = inttoptr i64 %add3 to i8*
11:  store volatile i8* %t_tag4, i8** @buffer_glb_tag_end, align 8
12:  %sub5 = add i64 %3, -1
13:  %t_a6 = inttoptr i64 %sub5 to i8*
14:  store volatile i8* %t_a6, i8** @buffer_glb_tag_start, align 8
15:  br label %for.body
16: for.body:  ; preds = %t_bf.exit.for.body_crit_edge, %entry
17:  %load_tag_end = phi i8* [%load_tag_end.pr,
     %t_bf.exit.for.body_crit_edge],[%t_tag4, %entry]
18:  %load_tag_start = phi i8* [%load_tag_start.pre,
     %t_bf.exit.for.body_crit_edge],[%t_a6, %entry]
19:  %indvars.iv = phi i64
     [%indvars.iv.next,%t_bf.exit.for.body_crit_edge],[0,%entry ]
20:  %4 = load i8*** @buffer, align 8
21:  %arrayidx = getelementptr inbounds i8** %4, i64 %indvars.iv
   //Tag marks check instructions before STORE instruction
22:  %5 = ptrtoint i8* %load_tag_start to i64
23:  %6 = ptrtoint i8** %arrayidx to i64
24:  %cmp_null_chk.i = icmp eq i8* %load_tag_end, null
25:  br i1 %cmp_null_chk.i, label %tag_check_storeinst_bf.exit,label %entry.i
26: entry.i: ; preds = %for.body
27:  %7 = load i8** @globaltag, align 8
```

```
28:  %cmp.i = icmp eq i8* %7, %load_tag_end
29:  br i1 %cmp.i, label %abortBB_funheap.i, label %if.else.i
30: if.else.i: ; preds = %entry.i
31:  %8 = bitcast i8* %load_tag_end to i8**
32:  %cmp2.i = icmp ult i8** %arrayidx, %8
33:  br i1 %cmp2.i, label %if.else5.i, label %abortBB_heap_overwrite.i
34: if.else5.i: ; preds = %if.else.i
35:  %9 = bitcast i8* %load_tag_start to i8**
36:  %cmp6.i = icmp ule i8** %arrayidx, %9
37:  br i1 %cmp6.i, label %abortBB_heap_underwrite.i,
     label %tag_check_storeinst_bf.exit
38: tag_check_storeinst_bf.exit:; preds = %if.else5.i, %for.body
39:  store i8* null, i8** %arrayidx, align 8
40:  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
41:  %10 = trunc i64 %indvars.iv.next to i32
42:  %cmp = icmp slt i32 %10, 101
43:  br i1 %cmp, label %t_bf.exit.for.body_crit_edge, label %for.end
44: t_bf.exit.for.body_crit_edge:   ; preds = %tag_check_storeinst_bf.exit
45:  %load_tag_start.pre = load i8** @buffer_glb_tag_start, align 8
46:  %load_tag_end.pr = load i8** @buffer_glb_tag_end, align 8
47:  br label %for.body
48: for.end:  ; preds = %tag_check_storeinst_bf.exit
49:  ret i32 0
```

Listing 5: A small section of C code using library function "strcpy"

```
1:   char *buffer;
2:   buffer = (char *)malloc(MAX_size);
3:   strcpy(buffer,argv[1]);// strcpy function call
```

Listing 6: LLVM-IR code (Un-instrumented) for C code presented in Listing 5

```
  //Lines 1-2 represent LLV-IR code for lines 1-2 of C code in Listing 5
1 :  %call = call noalias i8* @malloc(i64 80) #2
2 :  store i8* %call, i8** @buffer, align 8
  //Lines 3-7 represent LLV-IR code for line 3 of C code in Listing 5
3 :  %0 = load i8** @buffer, align 8
4 :  %arrayidx = getelementptr inbounds i8** %argv, i64 1
5 :  %1 = load i8** %arrayidx, align 8
6 :  %call1 = call i8* @strcpy(i8* %0, i8* %1) #2
7 :  ret i32 0
```

Listing 7: LLVM-IR code (Instrumented with tag-protection pass) for C code presented in Listing 5.

```
  //Dedicated Tag address creation for dangling pointer checks
1 :  %0 = tail call noalias i8* @malloc(i64 2)
2 :  store volatile i8* %0, i8** @globaltag, align 8
3 :  %1 = load volatile i8** @globaltag, align 8
4 :  store volatile i8 107, i8* %1, align 1
```

```
5 :  %call = tail call noalias i8* @malloc(i64 80) #1
6 :  %2 = bitcast i8* %call to i8**
7 :  store i8** %2, i8*** @buffer, align 8
   //Tag marks creation
8 :  %3 = ptrtoint i8* %call to i64
9 :  %add3 = add i64 %3, 80
10:  %t_tag4 = inttoptr i64 %add3 to i8*
11:  store volatile i8* %t_tag4, i8** @buffer_glb_tag_end, align 8
12:  %sub5 = add i64 %3, -1
13:  %t_a6 = inttoptr i64 %sub5 to i8*
14:  store volatile i8* %t_a6, i8** @buffer_glb_tag_start, align 8
15:  %3 = load i8** @buffer, align 8
16:  %arrayidx = getelementptr inbounds i8** %argv, i64 1
17:  %4 = load i8** %arrayidx, align 8
   //Reading tag mark before "strcpy" function call
18:  %fun_load_bf = load volatile i8** @buffer_glb_tag_end, align 8
19:  %cmpchek_null= icmp eq i8* %fun_load_bf, null
20:  br i1 %cmpchek_null, label %5, label %if.bf_funcall
21: ; <label>:5        ; preds = %if.notUAF, %entry
22:  %call1 = tail call i8* @strcpy(i8* %3, i8* %4) #1
   //Tag mark value check instructions after "strcpy" function call
23:  %load_tagchk_en = load volatile i8* @tagchk_en, align 1
24:  %cmp = icmp eq i8 %load_tagchk_en, 1
25:  br i1 %cmp, label %if.TagChkEn, label %6
26: ; <label>:6           ; preds = %if.TagChkEn, %5
27:  ret i32 0
28: if.bf_funcall: ; preds = %entry
29:  %load_glbtag = load volatile i8** @globaltag, align 8
30:  %cmpcheck= icmp eq i8* %fun_load_bf, %load_glbtag
31:  br i1 %cmpcheck, label %abortBB_funheap, label %if.notUAF
32: if.notUAF: ; preds = %if.bf_funcall
33:  store volatile i8 1, i8* @tagchk_en, align 1
34:  %load_t = load volatile i8* %fun_load_bf, align 1
35:  store volatile i8 %load_t, i8* @tagval_bfFuncall, align 1
36:  br label %5
37: if.TagChkEn: ; preds = %5
38:  store volatile i8 0, i8* @tagchk_en, align 1
39:  %load_bf= load volatile i8* @tagval_bfFuncall, align 1
40:  %load_af = load volatile i8* %fun_load_bf, align 1
41:  %cmpcheck_tagval = icmp eq i8 %load_bf, %load_af
42:  br i1 %cmpcheck_tagval, label %6, label %abortBB_funheap
```