Prevention of C/C++ Pointer Vulnerability

Zihan An
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
anxxx154@morris.umn.edu

ABSTRACT

Programming languages C and C++ have shown their vulnerability on the safety of memory allocation. Recently, the use-after-free (UAF) error and buffer overflow error are particularly popular among attackers. It is difficult to stop their exploitation using the current techniques. In order to provide a safer development environment for C and C++ users, this paper will discuss three new techniques that are provided by three different research groups in their recent studies.

Keywords

C/C++, use-after-free, buffer overflow, pointer vulnerability, type specification, type state analysis, machine learning, pointer tagging.

1. INTRODUCTION

The importance and value of developing methods that protect a language mostly depends on its usage. As a general purpose programming language, C serves many domains in programming world. The subjects could be for instance: operating system, development of other languages, computational platform, etc. Likewise, C++ supports varieties of the applications in the real-world such as game development, graphic user interface, web browser, etc.[1] These two languages share the same memory allocation system which is the dynamic memory allocation to the heap accessed by an address variable called pointer. This feature allows the programmer to customize the memory space allocation according to the amount of space they need. However, the downside of this memory allocation feature is that the handling of such allocation is fairly complex and the pointer has shown its extreme vulnerability on ensuring the safety of the allocation operations. Furthermore, since the C and C++ are pretty old languages in terms of the time they were invented, the safety issue was not a main concern back then. The languages did not have enough security implementation to counter the pointer vulnerability in default. These lead to many security issues. The most common two vulnerabilities are use-after-free and buffer overflow errors. They are particularly popular targets to attackers since their exploitations are hard to stop using the current techniques. The current approaches of preventing the pointer vulnerability are either expensive on the computer resource cost or lacking effectiveness.

In the Section 2 of this paper introduces the general background information of the terminology within it. Then the paper discusses the three solutions as preventions to the pointer vulnerability. In the Section 3 we discuss the first solution which is called *Type-After-Type Type-Safe Memory Reuse* (TAT)[2]. The TAT system uses type specification on the memory allocation to prevent attackers from using use-after-free error to access object with different type. In the Section 4 we discuss the second solution which is the *Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection*[3]. The focus of this solution is to apply the machine learning techniques to reduce the false positive detection on use-after-free error that constantly appears in existing detection methods. In Section 5 we discuss the third solution *Delta Pointer*[4]. The Delta Pointer on the other hand is focusing on solving the buffer overflow error. The Delta Pointer adds a upper bound check to original pointer and will shut down the program once the data is out of the bound. Therefore it can prevent the program from potential run-time bug and attacks. At the end, the conclusion wraps up the paper.

2. BACKGROUND

Memory allocation and reallocation can be different based on different programming languages. There is a region of memory called **heap** that can be allocated for the programs. Now we should observe the difference between Java, Python, and C/C++ allocating heap memory. For instance, Java has a garbage collection system running in the Java Virtual Machine (JVM) which releases the objects that the program no longer needs from the heap memory. Thus programmer has almost minimal engagement in the process. Likewise, Python has a hierarchy system that controls its private heap called Python Memoryp Manager. The memory manager frees a big chunk of memory that contains many small objects in the private heap at once when all the objects in the chunk are no long needed.

Unlike these programming languages, C and C++ have their unique memory allocation system. They do not require extra applications running in the background such as JVM and Python Memory Manage. However, programmer is responsible for both the creation and the destruction of objects (specifically in heap memory). The advantage of letting programmer having the control of memory allocation is that such operation is very flexible when designing a pro-

UMM CSci Senior Seminar Conference, April 2019 Morris, MN.

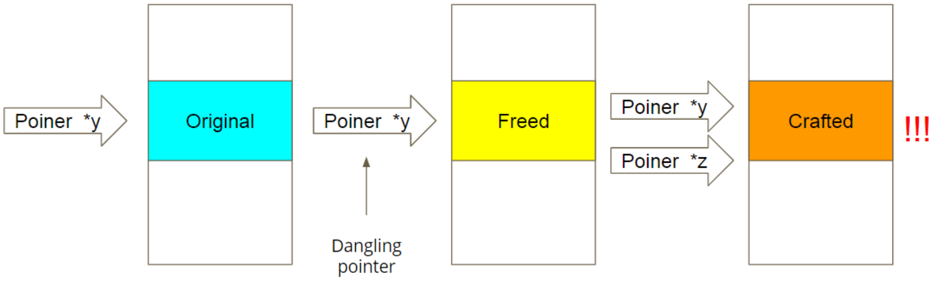


Figure 1: When a pointer is first time assigned to an object, it is a valid pointer. When a memory is "freed" after being used, the pointer still exist but points to an freed object, becoming a dangling pointer. If a new object is created in the same address which the dangling pointer points to, both dangling pointer and the new pointer will point to a same object.

gram. The program developers can always take the amount of memory they actually need instead a fixed value given by the computer. However, the risk of such operation is that it can be very problematic and even potentially harmful. The paper describes attack scenario in Subsection 2.2. The C and C++ memory allocation can be separated into two parts: *stack* and *heap*.

2.1 Stack & Heap

The stack is a special region of the computer memory which stores the local variables in the program. Stack consists of stack frames. Whenever a new function is used, it is pushed to the stack with the variables that were defined in it. When the function is no longer used, the function and its variables are removed from the stack entirely. The computer allocates space for stack in default thus the space is fixed and limited. Normally the variables stored in the stack are temporary, they are pushed in and popped out pretty often.

On the other hand, the heap is the region that the computer does not manage automatically for the user. The size of the heap is almost unlimited comparing to the stack (constrained by how much space available in the system). The heap is where C and C++ allow users to manually allocate and deallocate space for their programs.

2.2 Pointer, Use-After-Free Attack, and Buffer Overflow Attack

Unlike stack which is partitioned into stack frames, the heap almost does not have any ordering for the objects inside. In order to locate and access the object that is stored in the heap, C and C++ provide their unique referencing variable which is the pointer. Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. For instance, `int n;` is a declaration of a variable n, and `int *p;` is a declaration of a pointer that is called p. The actual value of p is an address which looks like `bff5a400` (refers to place on the memory that can be recognized by computer). Unlike the references concept in Java, which is just an alias to an existing variable, the pointer does not necessarily need to to refer an existing object. Figure 1 illustrates when the pointer does not refer to a valid object, or the memory where the pointer points to has been freed (available for reallocation), it becomes a *dangling* pointer. The existence of dangling pointer gives C and C++ memory allocation a great vulnerability. There are two scenarios that dangling pointer can be very harmful element to the program. One is that the object that the dangling pointer points to is not initialized automatically, therefore the new pointer points to the same address is able to read the information of that object. This can cause the leak of the information. Another is that the attacker can use the dangling pointer to craft the data at the object's address, if user uses the new pointer to access the data of that address, it can lead multiple types of data corruption and even cause sensitive information leak.

2.3 LLVM

The LLVM is a project that is a collection of modular and reusable compiler and toolchain technologies. The name LLVM is not an acronym, and LLVM has little to do with traditional virtual machines. [5] In this paper, LLVM is involved as a testing ground for our solutions.

2.4 Machine Learning

"Machine learning (ML) is a category of algorithm that allows software applications to become more accurate in predicting outcomes without being explicitly programmed. The basic premise of machine learning is to build algorithms that can receive input data and use statistical analysis to predict an output while updating outputs as new data becomes available."[6] The specific method of machine learning that this paper includes is the support vector machine, which analyse the training samples with annotated details (features) and provides a template for recognizing new samples.

2.5 Typestate

Typestate is a diagram representation of a machine. It represents the execution of a computer. In this paper, the typesate is used to represent the pointer at different stages. It is used by the Machine-learning-guided detection for the typestate analysis.

3. TYPE-AFTER-TYPE (TAT) TYPE-SAFE MEMORY REUSE

In this section we discuss the solution Type-After-Type

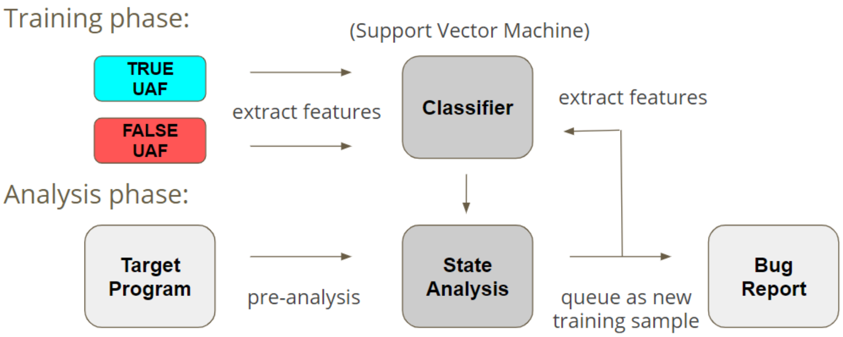


Figure 2: This is the general structure of TAC. In the training phase The Support Vector Machine takes both true and false use-after-free samples with their extracted features, and build the recognition model for state analysis. In the analysis phase the TAC takes target program and analyze it with the model, generates the bug report, and uses it as a new training set of the Support Vector Machine

(TAT) type safe memory reuse. This solution is done by a group of researchers in Vrije Universiteit Amsterdam [2]. The solution to the vulnerability is to provide a temporal memory type (a signature use-after-free error) safety. The temporal memory error happens when the program dereferences a dangling pointer or attempts to deallocate a pointer more than once. Assuming the program is dereferencing a dangling pointer to a dynamically allocated object on the heap, it allows the program to read the data of the object (can be sensitive data) which is not supposed to be read through this dangling pointer. On the other hand, if the program deallocates a pointer more than once, this “undefined behavior” can cause an uncertained error which can crash the program anytime afterward and potentially corrupt a part of the heap. “Temporal memory errors are a major threat to software security. Type-After-Type uses static analysis to determine the types of all heap and stack allocations, and replaces regular allocations with typed allocations that never reuse memory previously used by other types.” [2] Most of the current approaches to the vulnerability are setting an additional temporal metadata that links to a the pointer. Thus the program can check the temporal error by tracking down the metadata information of the pointer. However, the procedure takes fair amount of extra space and additional time. Therefore the resource cost is considered expensive. Comparing to existing approaches dealing with temporal memory safety, TAT has much less space requirement and run-time overhead by using its static method. The procedure is done differently in the heap and the stack.

3.1 Threat Model

The threat model should follow the conditions below: The attacker is seeking type-unsafe exploitation of temporal memory error vulnerability in the victim program. The victim program is considered capable of defending itself from other classes of vulnerabilities such as buffer overflows. The attacker is not able to access our metadata in the memory manager’s data structures (to keep track of multiple typed heaps) and the thread local storage (to store the stack pointers for multiple typed stacks). Therefore we can particularly focus on dealing with temporal type safety without concern of other potential parameters. [2]

3.2 Heap

To ensure the safety on the heap memory, the TAT separates the available memory in heap into several pools for each different type. Everytime after the application frees the memory, Instead of returning the memory to the operation system, TAT returns the memory to the same pool that will only be available for future allocations of the same type. The explicit procedure is scanning through the program and detect the type of the allocated memory. After detecting the type, TAT generates a 64-bits hash code (for security reason) that is unique for the specific type to distinguish with other types. In the future calls, TAT uses an alternative memory allocation function supplied by a custom Malloc Library will take the type hash code as an additional parameter. Therefore TAT can achieve type awareness in the heap. Although in the worst case (many calls from different type of memory) this procedure might cause some memory overhead, it provides a maximum type safety at a lower level cost.

3.3 Stack

On the other hand, “the security issue on the stack is quite different than on the heap. Stack allocations and deallocations are very frequent and must therefore be much more efficient than those operations on the heap.” [2] However, since the stack size is fairly small comparing to the heap and its variables have their type specified during the allocation time, the type analysis is not needed. A different approach to ensure the safety on the stack is to guarantee its initialization. Just like on the heap, TAT has a Stack Initialization Library that creates and initialize a separate stack with a pointer for each type whenever a new thread is created. Whenever the thread is closed or destroyed, its stacks will be freed to prevent memory leak.

3.4 Issues

TAT eliminates a wide range of temporal memory safety errors at a low cost. However, there are several limitations of the system.

First, attack can be wrapped as an allocation call to skip the type detection process. Secondly, the system cannot determine types if the target program is protected by using an arbitrary custom memory allocator. Lastly, the system requires source code as well as compiler managed type infor-

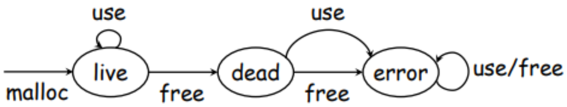
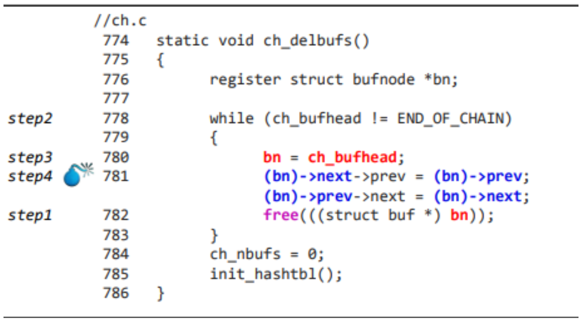


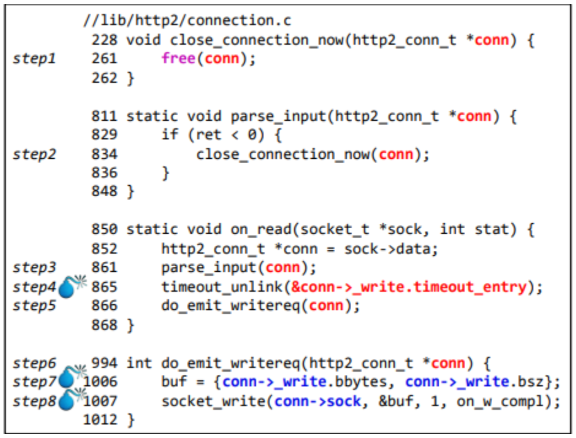
Figure 3: A brief state diagram illustrates the concept of how the analysis is checking its target program.

mation to help the type determination. These issues can be solved applying additional information to the system with the consent of user.



```
            //ch.c
             774   static void ch_delbufs()
             775   {
             776         register struct bufnode *bn;
             777
step2        778         while (ch_bufhead != END_OF_CHAIN)
             779         {
step3        780               bn = ch_bufhead;
step4   💣   781               (bn)->next->prev = (bn)->prev;
                               (bn)->prev->next = (bn)->next;
step1        782               free(((struct buf *) bn));
             783         }
             784         ch_nbufs = 0;
             785         init_hashtbl();
             786   }
```

Figure 4: A use-after-free bug found in less.



```
          //lib/http2/connection.c
           228 void close_connection_now(http2_conn_t *conn) {
step1      261     free(conn);
           262 }

           811 static void parse_input(http2_conn_t *conn) {
           829     if (ret < 0) {
step2      834         close_connection_now(conn);
           836     }
           848 }

           850 static void on_read(socket_t *sock, int stat) {
           852     http2_conn_t *conn = sock->data;
step3      861     parse_input(conn);
step4💣    865     timeout_unlink(&conn->_write.timeout_entry);
step5      866     do_emit_writereq(conn);
           868 }

step6      994 int do_emit_writereq(http2_conn_t *conn) {
step7💣   1006     buf = {conn->_write.bbytes, conn->_write.bsz};
step8💣   1007     socket_write(conn->sock, &buf, 1, on_w_compl);
          1012 }
```

Figure 5: Two use-after-free bugs found in h2o.

4. MACHINE-LEARNING-GUIDED STATIC UAF DETECTION

Another solution to the use-after-free errors is the Machine-Learning-Guided typestate analysis. It is done by a group of researchers in University of New South Wales, Australia. The research of approach is based on the limitation of the existing use-after-free detection programs. Since the over-approximation happens in the large scale target programs, it causes numbers of false detection and has big overhead. This approach is designed for large program with infeasible paths and recursion cycles, loops that cause the hardness to analyze the typestate of the pointer. The machine we use is called TAC. It uses a support vector machine applying to data investigated by the typestate analysis. One of the most important purpose of using the machine-learning techniques and static analysis is to best reduce the overhead of dynamic methods have and the number of false detection.

4.1 Concept

The solution is built using machine learning techniques. The entire process is divided into two phases. The first one is the training phase and the second one is the analysis phase.

Training Phase. The developers set up the TAC prediction model and train it using the both true and false UAF samples in the real world programs. These samples are annotated by the developers for feature extraction. The features are categorized into four categories: Type information (identifying type like array, struct, container, etc), control-flow (keeping track of information of pointer in loops and recursions), common programming practices (memory management in common practice), and points-to information (information of the location the pointer is pointing to)[3]. To actually train the prediction model, the support vector machine will take the features described above as parameters and mark as either harmful or false detection. The expected accuracy is obtained by a self testing during the training phase. To be more explicit, the provided samples are divided into 5 subsets with equal size. Then each subset is used as a test set while using other 4 sets as training set. The average accuracy of these test therefore is the expected accuracy.

Analysis Phase. The analysis is divided to 2 separate phases. The pre-analysis filters out "safe" objects that are determined by the traditional method and left only the candidate objects that may be unsafe for further investigation. The purpose of doing so is to significantly reduce the resource cost to the unnecessary targets. The developers give TAC a set of candidate object whether with use-after-free bugs or not and test its performance. Then the program gives relevant functions for TAC to interact with. The TAC will apply typestate analysis to these functions based on the typestate relation on Figure 3. With the sufficient data and technical analysis, the TAC is able to determine the true bug that contains the use-after-free vulnerability.

4.2 Case Study

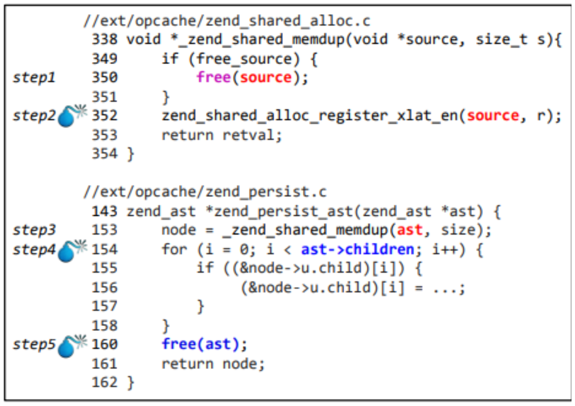
Here are 3 programs that contain use-after-free bugs found by TAC.

Less. Figure 4 shows an unknown use-after-free bug found in the **Less** (version 451) by TAC. TAC detected that the line 781 a same freed object is dereferenced four times in the while loop, causing one distinct use-after-free error.

H2o. Figure 5 shows couple known use-after-free bugs in **h2o** detected by TAC. At line 261, the program frees `conn` which is also involved in the nested call chain at line 834 to 861. However `conn` is used in another function `timeout_unlink` called at line 865. TAC was able to find these multiple different-type bugs.

Php. Figure 6 shows known use-after-free bugs and couple new ones in **php** (version 5.6.7) detected by TAC. These bugs are actually located in separate files. The `source` is freed at line 350 and then accessed at line 154. The freed `source` is further accessed by function of another program and double freed. TAC was able to detect such error related to different files.

There bugs above are all true bugs. The TAC found the



```
      //ext/opcache/zend_shared_alloc.c
      void *_zend_shared_memdup(void *source, size_t s){
          if (free_source) {
step1         free(source);
          }
step2     zend_shared_alloc_register_xlat_en(source, r);
          return retval;
      }

      //ext/opcache/zend_persist.c
      zend_ast *zend_persist_ast(zend_ast *ast) {
step3     node = _zend_shared_memdup(ast, size);
step4     for (i = 0; i < ast->children; i++) {
              if ((&node->u.child)[i]) {
                  (&node->u.child)[i] = ...;
              }
          }
step5     free(ast);
          return node;
      }
```

Figure 6: Two bugs found in php.

exact bugs in these program without giving any false detection.

4.3 Implementation

"The implementation of the TAC in the LLVM-3.8.0 showed a significant result. The evaluation used eight open-source C/C++ programs. TAC finds 109 bugs out of 266 warnings by suppressing 19083 warnings reported by TAC-NML."[3]

5. DELTA POINTER

Assuming an attacker is able to exploit a buffer overflow by feeding malicious inputs to a given vulnerable user program, he can repeatedly interact with the program, and the program is automatically restarted in case of crashes caused by failed exploitation attempts. A concept of the Delta Pointer is designed for the situation.

"The Delta Pointer provides an efficient pointer tagging to prevent such buffer overflow attack without checking explicit memory access operation. It encodes the out-of-bounds states in the pointer itself." [4] When using the Delta Pointer to access the memory, the program will read the metadata and if it detects that the pointer was overflown, the pointer will be recognized as an invalid pointer, causing a run-time error and stop the entire program.

The Delta Pointer provides a solution to replace the null pointers with a value of 0x7fffffff00000000 (a hexadecimal representation of 64 bits whole Delta Pointer). This will cause any dereference of a pointer derived from null to trigger a fault and hence detection.[4]

5.1 Threat Model

The threat model is considered of the following conditions: An attacker able to exploit a buffer overflow by giving malicious inputs to a given vulnerable user program. The attacker can repeatedly interact with the program, and the program is automatically restarted in case of crashes caused by failed exploitation attempts. The goal of the study is to detect the exploitation of arbitrary buffer overflows when memory is either written to or read from, protecting both integrity and confidentiality.[4]

5.2 Delta Pointer Structure

The key feature of the Delta Pointer is its structure. As it is described in Figure 7, a Delta Pointer consist of 64 bits value. The first bit is the overflow key which represent whether the pointer is overflown. The next 31 bits are the Delta tag which consists with the size of the buffer when it is initialized. The last 32 bits are the original address information of the pointer. As shown in the Figure 8, the Delta tag will keep track on the size information of the buffer. While we adding constants into the buffer, the value of delta tag increases respectively. Whenever the value reaches the upper bound 7fffffff (hexadecimal representation of 31 bits of "1"s), the overflow tag will be triggered. Any further addition to the buffer will cause the delta pointer to be "invalid" and shutdown the program to prevent the system from further damage. If anything keeps adding into the buffer, it would trigger the overflow tag, and further addition to the buffer would cause a run-time error, shutting down the program. Figure 9 illustrates when we need to retrieve the original address of the pointer, we use bitwise arithmetic to mask out the delta pointer tag and only keep the original address contained in the pointer. This also prevents the metadata of Delta Pointer itself to be revealed to outside.

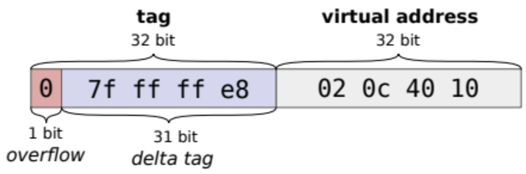


Figure 7: The first 32 bits are the tag which is added to the last 32 bits original pointer address. The very first bit is a overflow tag which determines whether the pointer is overflown or not. The following 31 bits are the Delta tag which represents the size of the buffer.

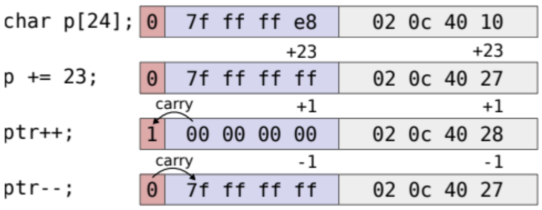


Figure 8: Delta tag increases as the size of the constants in the buffer increases

5.3 Coverage

Many existing pointer tagging applications only consid-

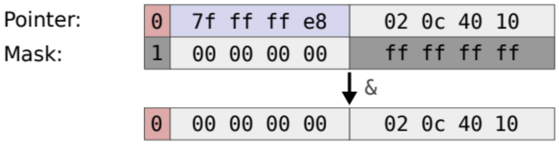


Figure 9: Masking out the Delta Pointer information

ered the situation which the pointer is tagged with metadata, because a null pointer is dereferenced and cause the program crush otherwise. However, this consideration ignored that the fact pointers can be unsafely used by the attackers and therefore we can not assume the presence of metadata in the pointer. According to this, a robust pointer tagging-based defense is designed to deal with the potential missing metadata. Delta Pointer is robust by design against missing metadata since the zeroed metadata can not be dereferenced as a valid pointer.

Delta Pointer focus on heap part of the memory allocation. This solution currently support all allocation functions in C/C++ standard libraries. It does not automatically support most of the custom allocators unless they have the similar structure for the implementation of Delta Pointer.

5.4 Performance and Implementation

The tagging operation is bitwise, which means it is very fast. However, the actual procedure could still cause the temporary memory pressure especially when there is conflict with the operating system. Delta pointer is also good at dealing against metadata corruption, since the pointer overflow will also overflow the delta tag, immediately invalidating the pointer.

The implementation is a Delta Pointer prototype for Linux using LLVM compiler infrastructure. "The code consists of 3,749 lines of code of LLVM C++ passes, which add the instrumentation described previously. An additional 846 lines of code make up run-time and helper libraries, including a static library that shrinks the address space of the process to make room for tags in pointers. The code is open source." [4]

6. CONCLUSION

As two of the most widely used programming languages, C and C++ appear in applications over many different fields. Their safety issues threaten significant amount of users information. Thus the solutions to prevent these threats are very essential. To overcome the challenges that affect the existing prevention methods, these 3 new solutions contribute a significant improvement of different aspects to the issue. Type-After-Type uses a static method to categorize the memory and specify its types to ensure the safety to use-after-free error. Machine-Learning-Guided Typestate Analysis implements machine learning to resolve the false detection issue within previous use-after-free detection. The dynamic Delta Pointer design provides an efficient solution to disallow the buffer overflow error, preventing both run-time bug and potential attacks. Although above solutions still have some levels of overhead and limitation issues, every cloud has a silver lining, and the better solutions will take their place in the future.

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Elena Machkasova for the patient and deep guidance of the paper, as well as Professor Nic McPhee and UMM alumnus Stephen Adams for their valuable feedback.

7. REFERENCES

[1] "Applications of c / c++ in the real world," Apr 2018. [Online]. Available: https://www.invensis.net/blog/it/applications-of-c-c-plus-plus-in-the-real-world/

[2] E. van der Kouwe, T. Kroes, C. Ouwehand, H. Bos, and C. Giuffrida, "Type-after-type: Practical and complete type-safe memory reuse," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 17–27.

[3] H. Yan, Y. Sui, S. Chen, and J. Xue, "Machine-learning-guided typestate analysis for static use-after-free detection," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC 2017. New York, NY, USA: ACM, 2017, pp. 42–54.

[4] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 22:1–22:14.

[5] "The LLVM compiler infrastructure project." [Online]. Available: http://llvm.org/

[6] "What is machine learning (ML)? - definition from whatis.com." [Online]. Available: https://searchenterpriseai.techtarget.com/definition/machine-learning-ML