HA^{2}lloc: Hardware-Assisted Secure Allocator

Orlando Arias
University of Central Florida
oarias@knights.ucf.edu

Dean Sullivan
University of Central Florida
dean.sullivan@knights.ucf.edu

Yier Jin
University of Central Florida
yier.jin@eecs.ucf.edu

ABSTRACT

With ever-increasing complexity of software systems, the number of reported security issues increases as well. Among them, memory corruption attacks are a prevalent vector used against today's software stacks. These attacks are repeatedly leveraged to compromise common application software, such as web browsers or document viewers. However, previous work to mitigate memory corruption attacks either suffer from high overhead or can be bypassed by a knowledgeable attacker.

In this work, we introduce HA^{2}lloc, a hardware-assisted allocator that is capable of leveraging an extended memory management unit to detect memory errors in the heap. We also perform some preliminary testing using HA^{2}lloc in a simulation environment and find that the approach is capable of detecting and preventing common memory vulnerabilities.

ACM Reference format:
Orlando Arias, Dean Sullivan, and Yier Jin. 2017. HA^{2}lloc: Hardware-Assisted Secure Allocator. In *Proceedings of HASP '17, Toronto, ON, Canada, June 25, 2017,* 7 pages.
https://doi.org/http://dx.doi.org/10.1145/3092627.3092635

1 INTRODUCTION

As the complexity of modern software increases, the possibility of encountering vulnerabilities that affect platform security increases. These vulnerabilities are estimated to cost the industry billions of dollars every year [1]. For this reason, companies such as Google, Microsoft, and Mozilla have implemented bug bounty programs, where *white hat* hackers are rewarded for finding security issues with their products [2–4]. Likewise, competitions such as Pwn2Own reward *white hat* hackers for their ability to compromise systems. Most of the vulnerabilities reported as part of bug bounty programs and used in competitions like Pwn2Own are memory-related. These vulnerabilities are the result of unsafe usage of languages that allow manual memory management.

Memory errors are prevalent in programs that are written in languages that allow direct access and management of memory. Memory errors can be generalized in two categories: *temporal* and *spatial* [5]. Temporal memory errors occur when the program attempts to utilize an allocation that has already been freed, whereas a spatial error occurs when memory is dereferenced outside valid bounds. At times, memory errors will result in accessing a portion of memory which has not been mapped to the application, resulting in an illegal memory access and a runtime exception being thrown to the application. However, under a sophisticated attacker [6], a memory error can result in security implications for the system such as the possibility to perform code reuse attacks [7] or leak sensitive data [8].

Previous work in academia and industry have used compiler instrumentation or software-based runtime analysis to detect memory errors. However, compiler-based approaches suffer from two issues: the precondition that source code for the application is available, and that the instrumentation is as good as the pointer analysis the compiler performs. Also, software-based runtime analysis introduces large performance penalties and may require a training phase. In this work, we propose a new type of memory allocator which combines both software and hardware elements to provide protection against memory errors while remaining transparent to software running on a platform. We call our memory allocator *HA^{2}lloc*, the *hardware-assisted allocator*. HA^{2}lloc utilizes the facilities of the runtime environment and operating system in combination with an extension to the memory management unit to detect both temporal and spatial memory errors as they occur without the need for compiler instrumentation. We demonstrate the low overhead provided by HA^{2}lloc and how it can be integrated and used to augment other compiler and software-based approaches.

At its heart, HA^{2}lloc employs a modified Memory Management Unit (MMU) in combination with a new memory allocator to detect temporal and spatial memory errors[1]. Our approach utilizes bounds data obtained by the allocator and forwards it to the operating system in order to populate a new set of structures in the MMU. When the MMU handles a memory access that is found in violation with the stored mappings, it triggers a fault which can be handled by the Operating System and the runtime environment.

The main contributions of this paper are:

- The introduction of a new memory protection scheme, HA^{2}lloc, that provides hardware-assisted support to detect memory errors which utilizes metadata obtained from the runtime environment to perform the necessary checks on memory accesses while remaining transparent to the application.
- A study and demonstration of the applicability of the approach as a defense against common attacks, such as virtual function table hijacking, use after free, and counterfeit object oriented programming (COOP).

The rest of this paper is structured as follows. Section 2 provides background information on buffer overflows and their effects. It then introduces previous approaches at protecting systems from these type of vulnerabilities. Section 3 provides a high-level overview of

[1] At this time, we have only emulated the MMU subsystem as to investigate the feasibility of the approach.

HASP '17, June 25, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-5266-6/17/06…$15.00
https://doi.org/http://dx.doi.org/10.1145/3092627.3092635

our proposed approach with section 4 describing our implementation. Section 5 provides in-depth testing and evaluation of our platform, including performance metrics and a discussion of its limitations. We then draw conclusions and present future work in Section 6.

2 BACKGROUND

Memory errors continue to be a trend, as the ten years of data collected from the Common Vulnerabilities and Exposures (CVE) database reflect [9]. Figure 1 reflects this data, showing only memory errors with a rating of high to critical. Software exploitation based on stack buffer overflows has dwindled over the years, with use after free vulnerabilities gaining traction and heap buffer overflow vulnerabilities maintaining steady momentum. We notice that some of the most powerful attacks are heap based, as we see an increasing trend in spatial and temporal heap-based vulnerabilities.

2.1 Example Vulnerability

Consider the sample code shown in Listing 1. Here, we demonstrate both temporal and spatial memory errors. There is a potential use after free vulnerability, as any of the objects stored in the c array may actually get deallocated before their member functions are called, resulting in the temporal memory error. There is also a potential spatial memory error by calling the `load_buffer()` function with a parameter that is larger in size than the buffer contained in the object. This results in a heap buffer overflow.

Listing 1: A small, vulnerable interpreter

```
#include <cstring>

class base {
  public:
    virtual void function() { ; }
    virtual void load_buffer(const char* buffer)
        = 0;
};

class derived : public base {
  char buffer[128];
  public:
    void function() { buffer[0] = '\0'; }
    void load_buffer(const char* buffer) {
      strcpy(this->buffer, buffer);
    }
};

int main(int argc, char* argv[]) {
  base* c[] = {nullptr, nullptr};
  char* p = argv[2];
  char m;

  while(*p) {
    switch(m = *p++) {
      case 'n':
      case 'N':
        if(!c[m == 'N'])
          c[m == 'N'] = new derived;
        break;
      case 'l':
      case 'L':
        c[m == 'L']->load_buffer(argv[1]);
        break;
      case 'f':
      case 'F':
        c[m == 'F']->function();
        break;
      case 'd':
      case 'D':
        delete c[m == 'D'];
        break;
    }
  }
  return 0;
}
```

An attacker can then utilize these vulnerabilities in order to corrupt memory in the heap. If allocation headers are kept near the allocations, then the buffer overflow vulnerability can be leveraged to inject a corrupted header. Furthermore, by careful manipulation of the allocations in the heap, a new `vtable` pointer can be injected to gain arbitrary control flow through a COOP-style attack [10].

Spatial memory errors can also result in the disclosure of sensitive information such as the base address of critical data structures or code pointers, thereby allowing the attacker to bypass randomization schemes that attempt to hide the locations of code and data segments such as ASLR [11]. As seen in the example, spatial memory errors can be exploited to overwrite these critical data structures or code pointers, allowing for information flow attacks or control flow attacks. An attacker is able to utilize temporal memory errors as a way to redirect control flow by injecting control flow data, such as a `vtable` pointer, into the reallocated memory region the stale object used to occupy.

2.2 Previous Work

Baggy Bounds Checking [12] introduces bounds checking for arrays in a granular fashion. Instead of keeping exact bounds for each array, it pads the allocation into bounds that are powers of two. This is done to reduce the overhead of the metadata by storing the exponent of the allocation only. On a 32 bit system, only 5 bits are needed to save the data and at storage time, one full byte is used. C library functions that deal with arrays, such as `strcpy()` and `memcpy()`, are provided with wrappers that check the bounds of the arrays before executing them. However, the mechanism is unable to prevent access errors when the buffer is located within an object such as a struct. Baggy Bounds Checking is a compiler based solution and thus requires binaries to be instrumented at compile time: source code is required. Unfortunately, no tools have been released to the general market. Looseness on the stored metadata also results in some checks being inaccurate. Performance wise, a 60% overhead is reported on a modified SPEC2000 suite and a 15% overhead in some Olden benchmarks.

AddressSanitizer [13] provides a method to instrument bounds check for software written in C and C++. It is implemented as a compiler pass and a runtime library. A portion of memory is dedicated as shadow memory, where metadata about arrays are stored. The memory is mapped into intervals of N bytes, and the mapping into the shadow area computed as $Addr >> Scale + Offset$ where Scale is given by N. If the transformation is applied to the shadow memory area, the resulting address will point to a portion of memory which is not mapped into the process's virtual address space, thus generating an access violation. AddressSanitizer provides a runtime library to aid with dynamic allocations, providing new versions of the `malloc()` family of functions and `free()`. The new

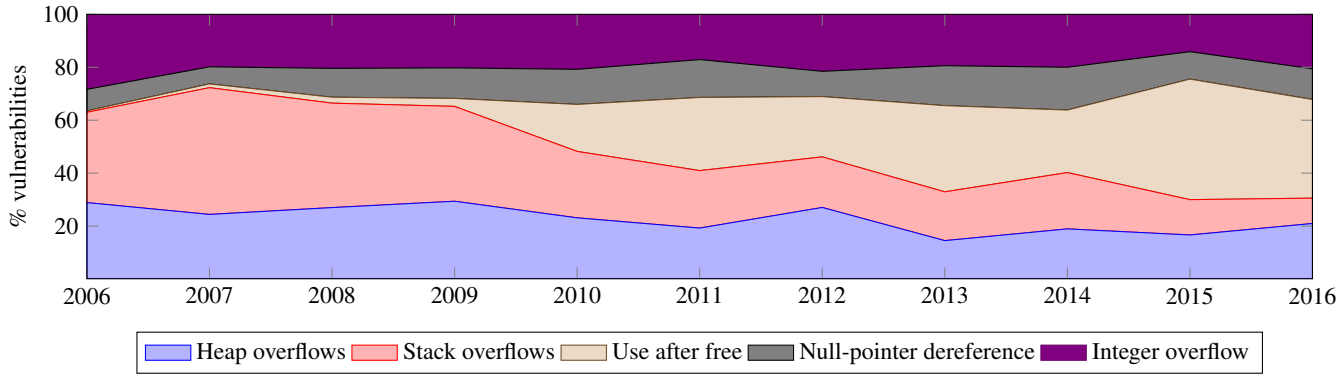

Figure 1: Trends in memory errors collected from the CVE database [9]. We show trends in memory errors in the last ten years that have resulted in a software vulnerability. Observable is how stack exploits have dwindled in favor of heap-based exploits.

allocator functions provide redzones around the returned region. These redzones are flagged as unaddressable and are used to store data from the allocator. The new implementation of `free()` poisons these redzones and puts them into a quarantine mode. Redzones are also added to buffers stored in stack frames. AddressSanitizer, however, presents a few false negatives and false positives, such as unaligned accesses that are partially out of bounds, accesses that fall too far away from the object bounds that may land in a different valid location, and load widening.

Sarbinowski et al propose VTPin in [14] as a way to counter some use after free exploits that result from temporal memory errors. VTPin provides a small library that intercepts calls to the allocator. Specifically, when a deallocation takes place, the VTPin takes control of the allocation and infers whether the deallocation corresponds to a C++ object. If it is, VTPin performs an in-place reallocation, leaving sufficient area to store a new set of virtual function table pointers. These point to an implementation controlled virtual function table. The in-place reallocation ensures that the virtual table pointer area is never reutilized, thus an attacker is unable to overwrite the virtual table pointer area by means of conventional heap spraying attacks such as Heap Feng Shui [6].

Watchdog [15] and WatchdogLite [16] propose a mechanism to store and check bounds data of a pointer or array with some hardware acceleration by using the SIMD extensions of x86 and x86_64 processors. This provides protection against spatial memory errors. Intel MPX [17] provides functionality similar to that of WatchdogLite, with the distinction that a dedicated set of registers, instructions and hardware exceptions were added to the processor. Intel MPX is available on 6th generation and newer processors. Being ISA-based, these approaches require compiler instrumentation for them to be of use.

Woodruff et al introduce Capability Hardware Enhanced RISC Instructions (CHERI) as a method to add *capabilities* to memory accesses in [18]. *Capabilities* are defined as the right to perform an action or set of actions to a given object. Furthermore, capabilities can be transferred between objects. In CHERI's case, the capabilities define the right of an instruction to make a memory access. For the purposes of implementation, CHERI is built as a coprocessor in a MIPS64 compatible core. Much like the previously mentioned approaches, compiler support is necessary to issue the necessary coprocessor instructions in a program. For this purpose, the authors utilize the LLVM compiler infrastructure in order to instrument source code.

2.3 Limitations of Previous Work

Compiler-based approaches such as AddressSanitizer [13], Baggy Bounds Checking [12], Watchdog [15], WatchdogLite [16], and Intel's MPX [17] inherently suffer from the outset as source code is required in order to instrument applications. Furthermore, the instrumentation is only as good as the correctness and completeness of the pointer analysis the compiler can perform. Unfortunately, pointer analysis has proven to be undecidable for the general case [19], and different algorithms suffer from either runtime or spatial considerations [20]. As such, compilers will perform a safe overestimation which can lead to incorrect instrumentation.

Herein lies the main issue with current compiler-based metadata approaches. Because we can not determine whether two symbols alias to the same value, we are unable to properly propagate metadata on this symbol for the general case. As such, there are instances where the information needed to perform the check is not available or inaccurate. Since compilers err on the side of safety, any performed check with incomplete or inaccurate metadata will pass, allowing temporal and spatial memory errors to occur.

Other approaches attempt to address either spatial or temporal memory errors. For example, although VTPin [14] ensures that the portions of an object that correspond to a virtual function table pointer can not be overwritten by subsequent allocations, it is unable to protect these areas against corruption that happens due to conventional heap buffer overflows. We were able to demonstrate this by crafting our own implementation of VTPin and constructing a vulnerable program that allocates two objects in the heap. We then free one of the objects and utilize a heap buffer overflow vulnerability in the other object to write into the reallocation made by VTPin. This results in the virtual function table pointer kept by VTPin being corrupted, resulting in arbitrary code execution from an attacker's

perspective. We should note that this attack is still possible even if the object is not deallocated.

3 PROPOSED APPROACH

Although compiled languages such as C and C++ often lose information on arrays when the final binary is built, such information may be reconstructed at runtime. For example, when a program dynamically allocates memory, the allocator has knowledge of both the allocation size and the address at which the allocation was made. We leverage this runtime information to gather the necessary metadata to enforce our buffer overflow protection and our temporal memory safety scheme.

3.1 Dynamic Memory Allocations

Modern computing systems implement process isolation by providing each process with its own virtual address space. In an `AMD64`-based system, each process is given a potential 48bit address space. However, no application is given a full address space when executing, as systems do not contain enough physical memory to support this. As such, applications are given the ability to dynamically request memory from the system. Enter the `malloc()` family of functions from the C library, and the `new` operator from the C++ language. With this, an application is able to expand its memory footprint by adding memory to the *heap*.

An allocator manages the heap memory for a process. The allocator is provided by the runtime environment, namely the C library in combination with the operating system, and it is completely transparent to the program. When a process deallocates memory using the `free()` function or the `delete` keyword, the allocator flags that portion of memory as unused, and can potentially cache it for future allocations. If there is not enough unused memory in the heap to satisfy a request, then the allocator proceeds to request more memory from the operating system utilizing the *system call interface*.

Internally, an allocator utilizes a series of data structures to keep a record of which allocations made by the application are currently active and which ones are freed. This data structure is called an *allocation header*. The way the allocator manages the allocation headers and the information they contain are specific to the allocator implementation itself. For example, some allocators, such as dlmalloc and derivatives [21], choose to keep allocation headers in front of the allocated space. This has the benefit of the allocator quickly being able to access information about the allocation by offsetting from a pointer to the allocated space. Unfortunately, a heap buffer overflow can easily corrupt adjacent allocation headers. Other allocators, such as OpenBSD's allocator, keep the allocation headers in a separate portion of memory [22]. This portion of memory is randomly mapped to the application and kept in a different memory area from the allocation itself. Although this secures allocation headers from being corrupted, the mechanism requires a search to be performed looking for the allocation header that matches the allocation itself. However, there are still common elements found in allocation headers. The size of of every allocation the application makes, the area of memory occupied by the allocation, and whether the allocated area has been freed or not is kept.

3.2 Design Constraints

With HA^{2}lloc, we wish to provide a drop-in mechanism that is compatible with existing applications without needing to rewrite or recompile them. For this purpose, we constrain our design to meet the following points:

- Transparency: The system must be completely transparent to applications. An application which exhibits legal behavior must not be affected in operation by the buffer overflow protection mechanism, nor should the application be able to infer it is running under the mechanism.
- Portability: Existing applications must work under the system without any type of modification to their source code and/or binaries. Applications are not to be modified at load time either.
- Integration: The mechanism must be easily integrated in an existing operating system and runtime environment with minor modifications. As long as the underlying hardware platform supports the mechanism, it should work without triggering any false-positives.

Given these constraints, compiler modifications are not allowed, as these will reflect a change in the binaries that get deployed on the system, violating the *Portability* requirement. Only modifications to the runtime, the operating system and underlying hardware platform are allowed. As such, we assume that an application will utilize the resources provided by the runtime environment and operating system, and conform to standard architectural and ABI conventions with special function registers.

In order to design HA^{2}lloc we observe the following:

(1) The internal data structures in the allocator have knowledge of the place where the allocated memory resides at and their sizes.
(2) The allocator must communicate with the operating system to request more memory when needed.

We utilize these observations in the next subsection to introduce the concepts behind HA^{2}lloc.

3.3 Introduction to HA^{2}lloc

We show a high level overview of HA^{2}lloc in Figure 2. At its heart, HA^{2}lloc provides a security aware allocator which separates allocation headers from the actual allocations in the heap. In doing so, we obtain two benefits. First, allocation headers can not be corrupted by conventional heap overflows. This aids with the integrity of the allocator. Secondly, it allows us to flag pages that have been specifically added to a process for the purposes of dynamic allocations.

When an application requests memory from the system using the `malloc()` family of functions, HA^{2}lloc's allocator handles the request. Besides performing a request to the operating system to allocate new pages to the application, HA^{2}lloc also forwards allocation metadata to the operating system itself. The allocation metadata consists of the size of the allocation and a possible base address in a page. The operating system records the allocation metadata on the page table entries used by the memory management unit as well as flag the associated pages as heap pages.

Furthermore, we randomize the base address of allocations within the process's virtual address space. In doing so, we introduce an extra

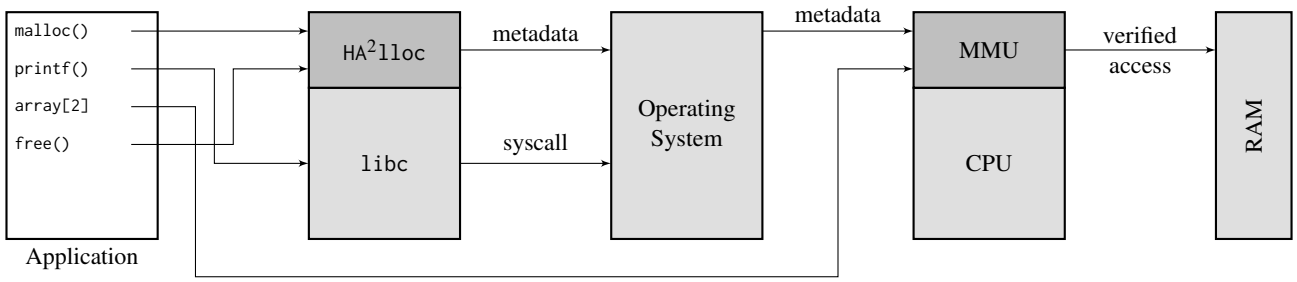

Figure 2: Overview of HA2lloc: HA2lloc provides the facilities required by an application to perform dynamic memory allocations whilst forwarding allocation metadata to the operating system. The operating system itself stores this information in the page table for the application. An extended MMU is then capable of utilizing the information to check memory accesses performed by the application.

layer of unpredictability to the allocator. That is, for two independent runs of the same program, two very different heap address maps are generated. This further allows us to mitigate heap-spraying style attacks, such as heap feng shui [6].

Since HA2lloc only provides the means to perform allocations, application software can go on to utilize other facilities provided by the system libraries. The system libraries can utilize HA2lloc's facilities to perform any dynamic allocations.

Any access the application performs to heap-mapped pages can then be verified by the MMU. The validation step remains transparent to the application, as it is performed directly by the MMU subsystem. Since the page table contains bounds information, the MMU can utilize this information to check accesses to heap mapped pages. If the access occurs within the recorded bounds, it is allowed. Otherwise, a fault is triggered and a signal is sent to the operating system.

We also need to be able to handle temporal memory errors. In order to do so, we must be able to handle any deallocations made by an application. When the process relinquishes an allocation by either calling the `free()` function, the `realloc()` function, or the `delete` keyword in C++, HA2lloc signals the operating system, forwarding information on the ongoing deallocation. The operating system in turn eliminates the allocation entry from the page table. If no more allocations reside in that particular table, the operating system unmaps the page from the process. The unmapped virtual address space is never reused.

When the process attempts a memory access to a deallocated area in the heap, one of two things will happen: either the page is unmapped triggering an illegal memory access, or the MMU is unable to find the bounds of the accessed address in the page table, triggering a similar fault. The operating system then is able to handle the fault accordingly, by either terminating the application or throwing a signal to the application.

4 IMPLEMENTATION DETAILS

4.1 The HA2lloc Allocator

Linux-based systems that utilize the GNU C Library use a modified dlmalloc as the base to manage heap allocations [21]. Allocators based on dlmalloc have the characteristic that they keep allocation metadata in front of the allocation that is returned to callers. The allocation metadata, or allocation header contains information on the size of the allocation, the next allocation bucket, and some other flags. Having the allocation header in front of the allocation allows the allocator functions to quickly obtain data from an allocation.

Although simple in design and fast in execution, a well versed attacker is able to exploit this allocator behavior to spray the heap and fool the allocator into thinking regions are allocated when they are not. Furthermore, heap buffer overflows allow an attacker to corrupt allocation headers, further enhancing their control over the application.

4.1.1 Allocating Memory. For this purpose, HA2lloc's allocator keeps the allocation metadata separate from the allocations themselves. Upon initialization, HA2lloc's allocator maps a page of memory where it keeps all allocation headers. Whenever a program requests memory through the use of `malloc()`, `calloc()`, or `realloc()`, HA2lloc requests memory from the operating system and creates a new allocation header. The allocator header is stored as part of a hash table. In order to handle collisions in the hash table, we utilize a red-black tree [23] on each bucket. This allows us to perform operations on the data structure in $\mathcal{O}\log n$ computational time in contrast to the amortized $\mathcal{O}n$ computational time that would result in handling collisions and resizing the hash table. Furthermore, by performing operations in this fashion in the hash table we can reduce the number of semaphores used in the allocation data structures, allowing for better parallelism in multi-threaded applications. Once the allocation is made and the header is constructed HA2lloc returns a pointer to the allocation to the user.

Of importance to HA2lloc is how pages are mapped to the application. Ideally, we would like to randomize the addresses of the pages mapped to the application whilst still ensuring that large allocations remain continuous in memory. Preliminary testing shows that Linux's `sys_mmap` does not attempt to randomize the addresses of the pages returned. The first mapped page has a relative random address. However, subsequent calls to `mmap()` will return pages at a fixed offset from the first page. This is detrimental to the security of our allocator, as all allocations would be in a predictable memory address. For this purpose, we introduce a new system call in the Linux kernel which performs a function similar to that of `sys_mmap` but it returns pages in a randomized fashion. We forward information about the desired allocation size to the kernel using this mechanism.

This information is used by the HA^{2}lloc's hardware subsystem to transparently perform bounds check in heap accesses (see Section 4.2).

4.1.2 Deallocating Memory. When an application deallocates memory, HA^{2}lloc removes the allocation header from the hash table, modifying the red-black tree if necessary. The removed allocation headers are added to a linked list to be reused by new allocations. The pages corresponding to these allocations are unmapped from the program. We ensure that these pages are never mapped to the program again by keeping a list of pages unmapped by the application within the virtual address map kept by the kernel in the process control block. In doing so, temporal memory errors result in an illegal memory access, triggering a segmentation fault.

4.2 HA^{2}lloc's Hardware Subsystem

HA^{2}lloc's Hardware Subsystem has yet to be implemented and tested. At its base, we extend the MMU to add one extra bit to flag heap allocated pages. Since heap pages are allocated using a new system call, no extra overhead is incurred in this flagging mechanism. We keep bounds information at the page level by associating a 32bit word to a heap page table entry. We illustrate the encoding in Figure 3.

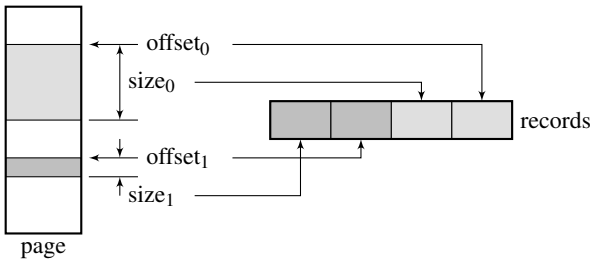

Figure 3: Bounds encoding mechanism used in HA^{2}lloc. A 32bit **word is associated with every heap page table entry. This word contains bounds information used by the MMU to perform checks on heap accesses.**

In order to record information on buffer sizes and offsets into the pages, we first analyze a few architectural constraints an allocator must follow. Type information is generally lost when compiling C code. Furthermore, the `malloc()` family of functions do not receive type information regarding the allocation that is being made. As such, these functions must assume a worse case scenario alignment for the datatype that is being allocated, both in terms of performance and ISA limitations. In C++, the `new` keyword could potentially use type information and specialize the allocation to better suit the datatype, but to the best of our knowledge, no C++ allocator performs this optimization.

For HA^{2}lloc, we assume a worst case alignment of 16B, given that this is the alignment required for common instruction set extensions such as Intel AVX [24]. As such, in a 4096B page, we can start at 256 different offsets. Consequently, we divide the 32bit word into 8bit subsections. We then group the subsections in pairs, with the lower byte denoting in which 16B block the allocation into the page starts, or the *offset* into the page, and the upper byte the number of 16B blocks covered by the allocation, or the *size* of the allocation. This means that we can potentially have up to two allocations per page. For two small allocations, we are then able to leave unused space between them, which can serve as a *red zone* to catch overruns. Multi-page allocations are handled in a similar fashion. Since the size field can cover the entire page, we can let the size field encompass the entire page, indicating that it covers a buffer.

When a memory access occurs to a heap-flagged page, the MMU utilizes the offset and size information recorded on the associated word to the page table entry and checks whether the access is within bounds specified for the allocations in the page. If it is, then virtual to physical address translation occurs as normal and the memory access is allowed. On the other hand, if the check fails, it is deemed to be caused by an illegal access. The MMU triggers a fault at this point, which must be handled by the operating system.

5 PRELIMINARY EVALUATION

A preliminary evaluation of our prototype implementation shows that for large allocations, HA^{2}lloc is faster than the `dlmalloc` implementation used in `glibc`. This is because `glibc` will scan through a circular list of freed allocations before mapping new heap pages to the application. For smaller allocations, `glibc` will expand the heap using the `sbrk` system call and perform the smaller allocations in that area. Since `glibc` can expand the heap multiple pages at a time using the `sbrk` system call, it can cache pages to be used by subsequent allocations and avoid expensive context switches.

Method	Temporal	Spatial
Baggy Bounds Checking [12]	no	yes†
AddressSanitizer [13]	no	yes†
VTPin [14]	yes	no
Watchdog [15]	no	yes†
WatchdogLite [16]	no	yes†
Intel MPX [17]	no	yes†
CHERI [18]	no	yes†
Our approach	yes	yes‡

† Requires instrumentation.

‡ In our current prototyping phase, bounds check is performed in a simulated environment and not implemented in a hardware MMU.

Table 1: Comparison between approaches

Table 1 offers a comparison between our protection mechanism and previous work. When running our sample vulnerable application on Section 2 we found HA^{2}lloc to be capable of detecting and preventing both the temporal and spatial memory errors. We also found that the vulnerabilities in the program were readily exploitable when testing against `glibc`'s `dlmalloc`. We also found that our reimplementation of VTPin was able to prevent the temporal memory error as long as the spatial memory error vulnerability was not triggered.

6 CONCLUSIONS AND FUTURE WORK

In this work, we present HA^{2}lloc, a secure memory allocator that utilizes an extended memory management unit to detect both temporal and spatial memory errors in the heap. We present the concepts behind HA^{2}lloc as well as preliminary testing of its implementation. We also compare HA^{2}lloc to previously proposed mechanisms in terms of coverage and deployability.

Future work for HA^{2}lloc includes the implementation of the memory management unit subsystem in order to test the effectiveness of the spatial memory error detection as well as any incurred overhead from these checks. Furthermore, we wish to be able to test reported vulnerabilities against HA^{2}lloc to further validate its usefulness. Lastly, we plan to extend HA^{2}lloc to also include stack-based buffers, as to provide a complete temporal and spatial memory error detection solution.

7 ACKNOWLEDGEMENTS

This paper is partially supported by the Department of Energy through the Early Career Award (DE-SC0016180). Mr. Orlando Arias is also supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 1144246. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the Department of Energy.

REFERENCES

[1] R. Telang and S. Wattal, "An empirical analysis of the impact of software vulnerability announcements on firm stock price," *IEEE Transactions on Software Engineering*, vol. 33, no. 8, pp. 544–557, 2007.

[2] Mozilla Foundation, "Mozilla security bug bounty program," https://www.mozilla.org/en-US/security/bug-bounty/.

[3] Microsoft Corporation, "Microsoft bounty programs," https://technet.microsoft.com/en-us/library/dn425036.aspx.

[4] Google, Inc., "Google application security," https://www.google.com/about/appsecurity/.

[5] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 48–62.

[6] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007.

[7] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[8] P. Ducklin, "Anatomy of a data leakage bug – the OpenSSL "heartbleed" buffer oveflow," 2014, https://nakedsecurity.sophos.com/2014/04/08/anatomy-of-a-data-leak-bug-openssl-heartbleed/.

[9] The MITRE Corporation, "Common vulnerabilities and exposures," https://cve.mitre.org/.

[10] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 745–762.

[11] P. Team, "Pax address space layout randomization (aslr)," 2003.

[12] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, 2009, pp. 51–66.

[13] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[14] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos, "Vtpin: Practical vtable hijacking protection for binaries," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: ACM, 2016, pp. 448–459. [Online]. Available: http://doi.acm.org/10.1145/2991079.2991121

[15] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 189–200.

[16] ——, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 175:175–175:184. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544147

[17] P. Guide, "Intel® 64 and ia-32 architectures software developerâĂŹs manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.

[18] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665740

[19] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.

[20] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '01. New York, NY, USA: ACM, 2001, pp. 54–61. [Online]. Available: http://doi.acm.org/10.1145/379605.379665

[21] D. Lea and W. Gloger, "glibc `malloc()`," http://malloc.de/en/.

[22] P.-H. Kamp, D. Miller, M. Dempsky, and O. Moerbeek, "Openbsd `malloc()`," http://bxr.su/OpenBSD/lib/libc/stdlib/malloc.c.

[23] R. Sedgewick and L. J. Guibas, "A dichromatic framework for balanced trees," *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, vol. 00, pp. 8–21, 1978.

[24] Intel Corporation, "Intel Architecture Instruction Set Extensions Programming Reference," December 2016, document Number: 319433-028.