Opleiding Informatica

Efficiently combining compiler-based

zero-day defenses

Marcel Huijben

Supervisors:

dr. Erik van der Kouwe

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)

www.liacs.leidenuniv.nl

14/08/2019

Abstract

Software bugs can cause exploitable vulnerabilities in programs. Multiple defenses have been proposed to counter the exploitation of these vulnerabilities, however these defenses cannot by themselves guarantee the full safety of a program. However naively combining existing defenses can result in large runtime penalties. In this paper, we propose a combination of existing defenses which more efficiently combines existing defenses into a combined defense mechanism. We combined the existing defenses Baggy Bounds Checking en Compiler Enforced Temporal Safety into one larger defense which has the security guarantees of both defenses. We then evaluated both the performance and security of the combined defense, and proved that our combined defense mechanism can prevent a number of programs being exploited at a significant runtime overhead.

Contents

1 Introduction **5**
1.1 Contributions 6

2 Background **7**
2.1 Spatial Safety 7
2.2 Temporal Safety 9

3 Overview **10**
3.1 System Structure 11

4 Design **12**
4.1 Per-object data 12
4.2 Per-pointer data 13
4.3 Pointer arithmetic 13
4.4 Pointer dereference 14
4.5 Allocation & Deallocation 14
4.6 Off-By-One Pointers 16
4.7 Alternative Designs 16

5 Implementation **18**
5.1 Intermediate Code Transformation 18
5.1.1 Per-opcode Transformation 18
5.1.2 Stack Allocation Transformation 18
5.1.3 Global Data Transformation 19
5.1.4 Heap Allocation Transformation 19
5.1.5 External Module Compatibility 19
5.2 Runtime Routines 19
5.3 Optimizations 20

6 Evaluation **22**
6.1 Performance 22
6.1.1 Limitations 22

6.1.2 Results 25
6.2 Security 27
6.3 Optimizations 28

7 Limitations **29**
7.1 Setjmp 29
7.2 Small Key Space 29
7.3 Concurrency 29
7.4 Sub-Object Safety 30
7.5 Library Compatibility 30

8 Related Work **32**

9 Conclusions **34**
9.1 Future Work 34

Chapter 1

Introduction

When programming an application in a low-level programming language, programming errors can lead to vulnerabilities in the final application. The main cause of this is the compiler failing to catch all memory errors and semantic constructs in these languages allowing for the reading from and writing to arbitrary memory locations. In these low-level languages, the memory management is the entirely the responsibility of the programmer, with the compiler providing little to no support for ensuring the safety of the code written. Invalid use of these unsafe memory constructs can cause bugs. A subset of these bugs is exploitable and can compromise the security of the application.

Exploitable bugs created by breaking memory safety have in the past been discovered in numerous real-world applications. Famous examples include the HeartBleed bug found in OpenSSL implementations, which allowed attackers to obtain privileged information vulnerable implementations by abusing a buffer over-read and affected a large number of web servers [4]. Another example can be found in the SQL Slammer worm, which targeted a stack overflow vulnerability in Microsoft's SQL Server to spread itself and infected over 200,000 individual systems [10].

Over the past years numerous defense mechanisms have been proposed to mitigate the security risks posed by these programming errors, examples of which include Softbound [7], Baggy Bounds Checking [2] or Compiler Enforced Temporal Safety [8]. Each of these defense mechanisms is able to prevent a subset of all exploitable bugs, and each defense mechanism comes with a set of restrictions on the application incorporating them.

However no single defense mechanism can currently guard an application against all types of bugs that might appear in the program, combining multiple defense mechanisms can be desirable to improve the security of the application. However since the different defense mechanisms institute their own set of restrictions, it becomes harder to efficiently combine multiple defenses.

When combining defenses, the restrictions imposed by the separate defenses may conflict, making combining such defenses significantly harder. Furthermore, combining defenses naively will impose a large runtime overhead on the application. Since defenses often keep track of related metadata, such as object size or address usage, it might be possible to instead combine the metadata and checks performed to reduce the runtime performance penalty on the resulting application.

1.1 Contributions

In this paper we will describe methods to combine certain existing defenses into efficient combined defenses which provide the security of their components. We will also describe a combination of two defenses. By describing these combined defenses, this paper will make the following contributions:

- We will describe a novel design for a combined defense mechanism.
- The described combined defense mechanism will be implemented.
- The combined defense mechanism will be evaluated to determine the runtime overhead imposed by the combined defense mechanism.

Chapter 2

Background

In the past, mechanisms to prevent exploitation of bugs by malicious users have been developed. Most of these defenses keep track of information about the state of the program, for example by tracking allocations, and later use this metadata to insert runtime checks into the program to ensure the safety of the program. Each of these defenses has a different set of bugs they can guard against. We split the goal of defenses into roughly two major goals, ensuring spatial safety and ensuring temporal safety.

2.1 Spatial Safety

A program is considered to uphold spatial safety when no spatial errors can occur. A spatial error is an error in which a pointer is used to access a memory location not contained in the object the programmer intended the pointer to access. A common example of a spatial error is a buffer overflow, where addresses beyond the bound of an allocated buffer can be accessed.

To guarantee spatial safety, a number of defenses have been proposed, such as Softbound [7] and Baggy Bounds checking [2]. Softbound guarantees spatial safety by adding pointers to the base and bound of an object to every pointer, and verifying that this pointer is within the object bounds on pointer dereference. Baggy Bounds Checking provides spatial safety by padding object sizes to next power of two higher than or equal to the size of the object, and keeping a metadata table containing the size of the object, using the pointer address as the index for this metadata table.

When using Softbound to guard an application, two pointers of metadata are propagated with every pointer: a pointer to the base of the object and a pointer to the end of the object. This metadata is propagated on pointer arithmetic. When a pointer is dereferenced, the pointer dereferenced is checked against the base and the bound to ensure the pointer is between the base and bound of the object. The check will prevent any invalid pointers from being dereferenced. Figure 2.1 shows one pointer pointing halfway into an object, with the base and bound pointers.

Baggy Bounds Checking on the other hand checks the validity of pointer arithmetic instead of pointer

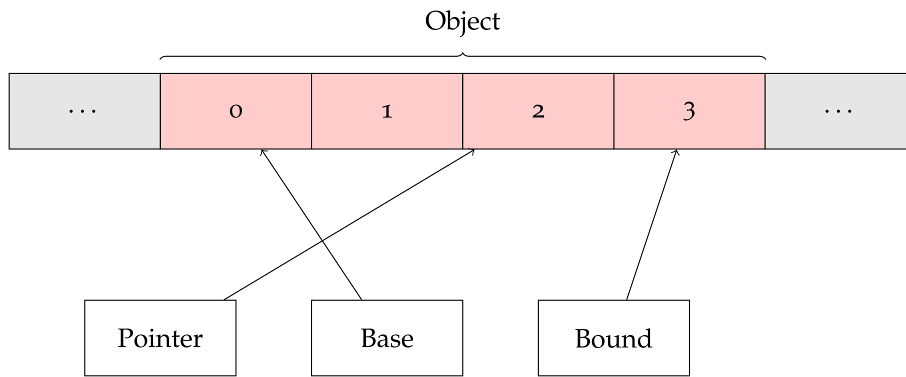

Figure 2.1: Softbound

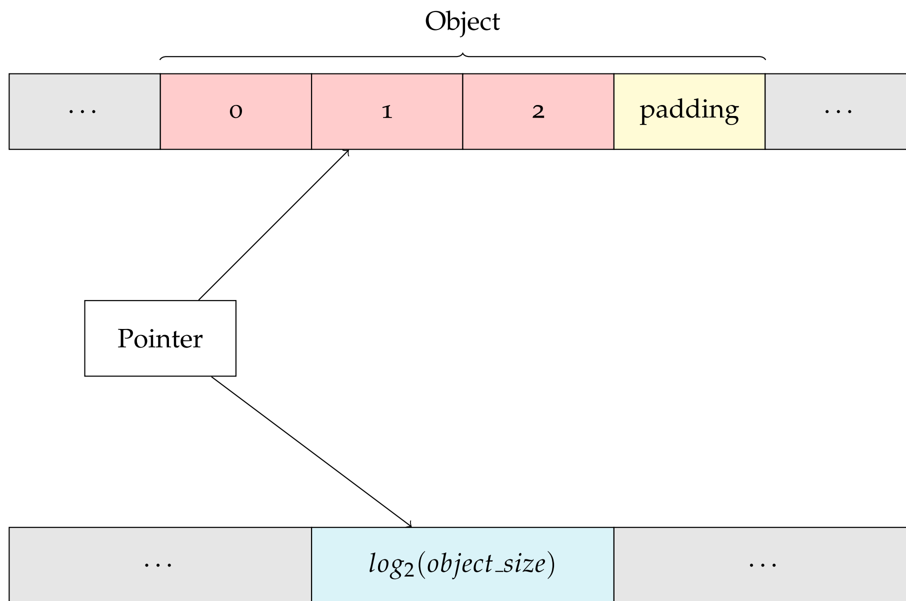

Figure 2.2: Baggy Bounds Checking

dereferences, thus preventing the creation of invalid pointers. To achieve this, Baggy Bounds Checking first pads all object to the nearest power of two. This allows the defense to store the binary logarithm of the object size in a separate table with metadata. To access the metadata describing the size of the object pointed to by a given pointer, Baggy Bounds Checking uses the address stored in the pointer itself. To accomplish this, Baggy Bounds Checking divides the entire address space into equally sized slots, and stores metadata per chunk. The location of the metadata belonging to a pointer is then computed using the formula $\text{index} = \text{pointer} >> log_2(\text{slot_size})$. This index is then used to index the metadata table, which can simply be stored as a continuous array.

By aligning all objects to the nearest power of two, and knowing the size of all objects, the object bounds can easily be computed from any pointer pointing within the object bounds. On arithmetic, the resulting pointer after the arithmetic operation is asserted to be within these object bounds. Figure 2.2 shows a pointer pointing halfway into an object, furthermore it also shows that the pointer can be used to index the object size in the metadata table.

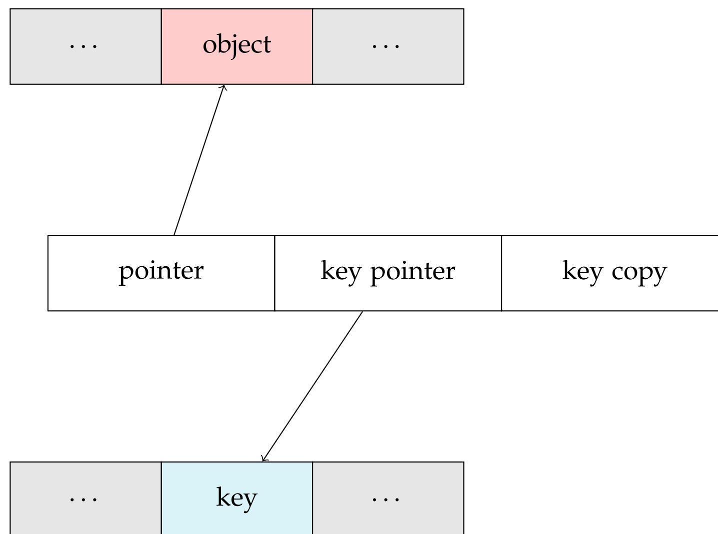

Figure 2.3: CETS

2.2 Temporal Safety

Temporal safety refers to the prevention of any temporal errors. A temporal error is an error where a program accesses an object after the object is no longer allocated. An example of a temporal error is a use-after-free bug, where an object is references after the memory allocated for that object is already freed.

Compiler Enforced Temporal Safety, or CETS [8] for short, can guarantee temporal safety when spatial safety is guaranteed. CETS gives every allocated object a unique key, which is generated on allocation. Every pointer to this object is then augmented with a copy of this unique key, and another pointer which points to the original key in the metadata table. When an object is deallocated, the unique key stored in the metadata table is invalidated. When a pointer is dereferenced, the key which is paired with the pointer is compared to the stored unique key, and the operation is only valid when the keys match. If the keys did not match, this implies that the original object is deallocated, and thus the pointer is used after the object is no longer valid. Using this mechanism CETS can prevent the usage of any object after it has been deallocated.

However, there is one more type of temporal error, namely double-free errors. To resolve this, CETS adds any pointer returned by a heap-based allocator to a list of freeable pointers. When a pointer is being freed, CETS checks if the pointer appears in the list of freeable pointers. If it does appear in this list, the pointer is removed from the list. This ensures that no pointer can be freed twice.

Figure 2.3 shows a pointer pointing to an object, the pointer is augmented with a copy of the key and the key pointer pointing to the key in the metadata table.

Chapter 3

Overview

In this paper, we will be looking to combine the defensive features of Baggy Bounds Checking and Compiler Enforced Temporal Safety. The reason for choosing to combine Baggy Bounds Checking and CETS is that the two defenses together can guard against a wide number of attacks. For example to provide a guarantee of temporal safety, CETS requires a guarantee of spatial safety. Baggy Bounds Checking can guarantee a subset of spatial safety. By combining these two defenses we can thus catch a subset of both temporal and spatial errors. Currently, CETS is often combined with Softbound to provide the guarantee of spatial safety CETS requires, however Baggy Bounds Checking on average has a lower runtime overhead than Softbound. Baggy Bounds Checking has a runtime performance penalty of 60% on average whereas Softbound has a 67% penalty. [2] [7] Thus we chose to research if combining these two defenses is possible. To determine how to combine these two defenses, we first have to identify the overlap between the two defenses. We found the following overlap between the two defenses:

- Both Baggy Bounds CETS keep per-object metadata separate from the object itself, for Baggy Bounds Checking the size of the object is stored, whereas for CETS a key is stored.
- Metadata is created once on object allocation, and destroyed on object deallocation, with no modifications made to the per-object metadata after the object is allocated.
- Pointer-bound information is used to access the metadata, either propagating some information with the pointer as CETS does, or automatically by directly using the pointer itself as an index to access the metadata, like Baggy Bounds Checking does.
- Aside from per-pointer metadata and per-object metadata, no other metadata is being stored.

By identifying these similarities between the two defenses, we can start combining the two defenses into one defense. From the described similarities we can conclude that the following optimizations are possible:

- We can merge the per-object metadata of both defenses into one per-object structure, instead of keeping two separate structures.

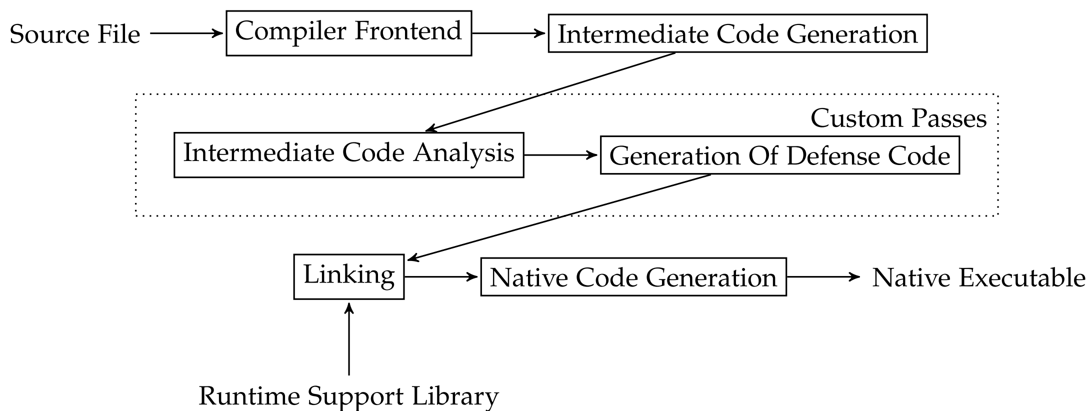

Figure 3.1: LLVM Architecture With Custom Passes

- Since Baggy Bounds Checking can use the pointer itself to find the per-object metadata, there is no need to keep a separate pointer to the per-object metadata as CETS does.
- Allocation and deallocation routines of both defenses can be fused into a single routine.

Using these guidelines, we can design a fused defense, which is described in chapter 4.

3.1 System Structure

To develop our fused defense, we need to both generate defense code when compiling programs and provide a set of runtime routines used in the generated code. Since designing an entire compiler from scratch would be infeasible, we made use of the LLVM compiler framework [6]. The system used is shown in figure 3.1.

When building an executable using our defense, first the source files are sent through the LLVM frontend, which parses the source files given to it and transforms them into an internal representation, which is then passed to the intermediate code generator. The intermediate code generator transforms this internal representation into LLVM IR code. This LLVM IR code is then passed to our plugin, which starts by analyzing the code given to it. The results of this analysis are then used in the defense code generation phase, which transforms the original LLVM IR code using the information obtained in the analysis phase to insert defense code when required. This transformed IR code is then passed on to the linker, which uses both the transformed IR code and the runtime support libraries, and links them together. The result of the linking phase is then passed to final stage, which transforms the intermediate code into a native executable.

The custom passes are contained in a shared library which can be loaded into LLVM as a linker plugin. The runtime support library was compiled into a static library which can be compiled into the executable during the linking stage. An overview of the modified LLVM architecture can be found in figure 3.1.

Chapter 4

Design

To actually combine the two defenses listed earlier, we will have to combine the datastructures and runtime routines used by Baggy Bounds Checking and CETS. As discussed in chapter 3, we need to merge both the per-object data as well as the per-pointer data. Baggy Bounds Checking requires us to perform a check on pointer arithmetic, and CETS requires a check on pointer dereferences, so we will have to insert runtime routines to perform these checks. Furthermore, we need to insert code at allocation and deallocation to update the metadata tables. All of these routines is described below.

4.1 Per-object data

To merge the per-object data structures, we will need to combine all required fields of both Baggy Bounds Checking and CETS in such a way that we can efficiently access all data in this structure without incurring too much of a runtime overhead. Another important factor when designing the data structure is the final size. Since Baggy Bounds Checking requires metadata to be available for the full address space, keeping the size of the structure low is especially important.

On a 64-bit platform, the theoretical maximum size of an object is 2^{64} bytes. However, since this would fill the entire address space, leaving no room for metadata or code, we can assume the theoretical maximum size of an object to be lower than this. Since Baggy Bounds Checking pads all objects to a power of two, we only have to store the binary logarithm of the actual size. Furthermore, since we cannot create an object of 2^{64} bytes, this means the maximum object size could in theory be 2^{63} bytes. This means we only have to store values in the range $[0, 63]$, which can be stored in 6 bits.

However, to remain compatible with raw pointers obtained from external libraries, Baggy Bounds Checking initializes table entries for unmanaged data to a size of 2^{64} to allow any arithmetic operation on this pointer to succeed and reach the entire address space. To achieve this, we will add a seventh bit to the size field.

To reduce the total size of the metadata table, we want to reduce the size of each entry. To achieve this, we will use only 2 bytes to represent the per-object metadata. Taking into account the 7 bits used to store the object

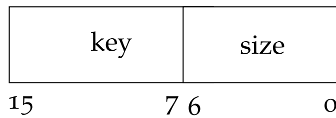

Figure 4.1: Layout of the combined metadata table

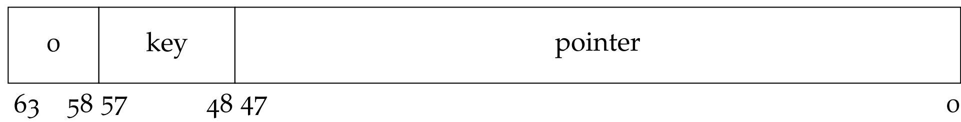

Figure 4.2: Layout of pointers

size, we have only 9 bits left to represent the key, giving us a total of only 512 unique keys. The final structure of a metadata table entry is shown in figure 4.1.

4.2 Per-pointer data

When using CETS, two fields are propagated with each pointer: a copy of the stored per-object key, and a pointer to the location of this key. However, since we stored the per-object key in a structure with the Baggy Bounds Checking size metadata, we no longer need the pointer to the location of this metadata, meaning we only have to propagate the copy of the per-object key with our pointer.

To efficiently store this key, we make use of the fact that our target architecture only uses 48 out of the 64 bits in a pointer to actually store the location, meaning there are 16 bits available for storage within the pointer itself. Since the only data we have to store is the copy of the per-object key, which is only 9 bits, we can easily fit this data within the original pointer. Thus we can store the copy of the per-object keys in bits 48-57 of the pointer. Figure 4.2 shows the bit layout of pointers combined with metadata.

4.3 Pointer arithmetic

CETS requires its metadata to be propagated on any pointer arithmetic operations. Since we discarded the pointer to the location of the per-object key, the only field we have to propagate is the copy of the per-object key. We stored this key in the 16 most significant bits of our pointer, which are normally unused, we can assume that any operation normally valid on a pointer does not modify these bits, meaning they are automatically propagated on any pointer arithmetic operation. An attacker could however attempt to inject an invalid offset into the program which would cause the arithmetic to overflow into the key in an attempt to modify this key. However Baggy Bounds Checking can catch overflow and prevent it.

Baggy Bounds Checking requires a runtime check on pointer arithmetic to assert that the pointer is still within object bounds. This assertion is verified by performing a runtime check after the arithmetic operation. In this check, the result of the arithmetic operation is checked against the original pointer and the size field stored in the metadata table. Since the optimized bounds check from Baggy Bounds checking verifies that all bits not within the bounds of the object are the same between the the original pointer and the new pointer, this also

```
new_ptr = ptr + offset;
ptr_bitmask = (1 << 48) - 1;
in_bounds = (new_ptr ^ ptr) >>
                metadata_table[(ptr & ptr_bitmask) >> slot_size].size == 0;
if(!in_bounds)
    abort();
```

Listing 4.1: Pointer Arithmetic

```
ptr_bitmask = (1 << 48) - 1;
key_match = (ptr >> 48) == metadata_table[(ptr & ptr_bitmask) >> slot_size].key;
if(!key_match)
    abort();
```

Listing 4.2: Pointer Dereference

verifies that the key field stored within the two pointers matches without having to perform a separate check. Thus we can use this single check to prevent attackers both from modifying key data using an overflow as from obtaining an out-of-bounds pointer. The code for our pointer arithmetic routine is given in listing 4.1.

4.4 Pointer dereference

CETS requires a check to see whether the copy of the per-object key stored with the pointer matches the per-object key stored in the metadata section. Since we stored the per-object key in the Baggy Bounds Checking metadata table, we can use a simple array index to access the original key. We can thus compare the keys using a few bitwise operations. The operations performed on pointer dereferences is shown in listing 4.2.

4.5 Allocation & Deallocation

On allocation, a metadata table entry has to be filled in with information describing the currently allocated object. For Baggy Bounds Checking, we only have to fill in the size field, whereas for CETS, we have to generate a new key. We do this by storing the value of the last allocated key as a global value, and simply incrementing this value when we need a new unique key. We make sure to skip the key value of zero, which is used as a constant to describe invalid keys. Our allocation routine is described in listing 4.3.

In this code, `next_key` represents a global value representing the next CETS key to generate. We fill all slots belonging to the newly allocated object with the correct size and key information. Finally we store the copy of the CETS key in the new pointer, which can then be used to refer to the new object.

On deallocation, we first need to determine whether the object was allocated at all. CETS uses a separate freeable pointers map to keep track of this information. However, since we have metadata for every address in the program due to Baggy Bounds Checking, we can reuse this data to check whether an object was allocated

```
num_slots = size / slot_size;
new_key = next_key;

next_key = (next_key + 1) % 512;
if(next_key == 0)
    next_key = next_key + 1;

base_index = ptr >> slot_size;
for(i = 0; i < num_slots; ++i) {
    metadata_table[base_index + i].size = log2(size);
    metadata_table[base_index + i].key = new_key;
}

ptr = ptr | (new_key << 48);
```

Listing 4.3: Allocation

```
ptr_bitmask = (1 << 48) - 1;
base_index = (ptr & ptr_bitmask) >> slot_size;

num_slots = 1 << metadata_table[base_index].size;

base_ptr = ptr >> metadata_table[base_index].size;
if((ptr >> 48) != metadata_table[base_index].key || ptr != base_ptr)
    abort();

for(i = 0; i < num_slots; ++i) {
    metadata_table[base_index + i].size = 64;
    metadata_table[base_index + i].key = 0;
}
```

Listing 4.4: Deallocation

at all. An object not allocated has a key value describing an invalid key, which is set to zero. By checking if the key present in the pointer is equal to the key currently stored in the metadata table associated with the given address, we can determine both if the object has already been freed or if the address is being reused by a new object, and a deallocation call was made through an old pointer. Next we still have to check whether the pointer given points to the base of the object, which is a simple bitwise operation using the size field in the per-object metadata section. If the object can be freed, we zero the of the per-object metadata entries belonging to the memory being deallocated, which also sets the key value to describe an invalid key value, causing future CETS checks referencing this address to fail. The deallocation routine is given in listing 4.4.

```
new_ptr = ptr + offset;
ptr_bitmask = (1 << 48) - 1;
object_size = metadata_table[(ptr & ptr_bitmask) >> slot_size].size;
in_bounds = (new_ptr ^ ptr) >>object_size == 0;
if(!in_bounds)
{
    in_bounds = new_ptr == ((ptr >> object_size) + 1) << object_size;
    if(!in_bounds)
        abort();
}
```

Listing 4.5: Pointer Arithmetic with Off-By-One support

4.6 Off-By-One Pointers

Since the C standard allows obtaining pointers to one element past the last element of an array, our defense should support this to prevent breaking any existing code. To do this, we modify the original pointer arithmetic routine into one that does support obtaining pointers to one element past the end of the buffer. Our modified routine is shown in listing 4.5.

We calculate the address directly at the end of the current object, and compare the result pointer to this value. If these two values match, we know that the calculation yielded a pointer to one element past the current object, and the operation should be valid according to the C standard. Since off-by-one pointers tend to appear less frequently than pointers that are in bounds, we decided to perform the off-by-one check in the slower path taken when the initial calculation is not in bounds, reducing the performance impact.

However, allowing off-by-one pointers could pose a security risk, as dereferencing these pointers is still an illegal operation. The existing dereference check prevents exploitation in most cases. Since the off-by-one pointer points outside the object, the dereference check would compare the key in the out-of-bounds pointer to the key for object directly behind our original object. Since distinct objects should have different keys, this would cause the CETS check to fail, thus preventing the program from dereferencing the out-of-bounds pointer.

4.7 Alternative Designs

Our current implementation requires both checks on dereferences and pointer arithmetic. An alternative design could however perform checks solely on dereferences. Using the same mechanism as described in section 4.6, we could use the CETS check to determine whether a pointer dereference is trying to access another object by comparing the key stored with the pointer to the key stored in the metadata table, which should be distinct for when the objects referred to differ. This would allow us to remove all checks on pointer arithmetic, as the validity of the pointer is then checked on dereference. This does however have a drawback, since the key space is rather small, the chance that two distinct objects have the same key is realistic, which would allow out-of-bounds pointers pointing to another object with the same key value to access this object without any

checks failing.

This problem could be solved by increasing the size of the key field in the metadata table to increase the number of availlable key values, thus decreasing the chance of collisions. However this would significantly increase the size of the metadata table. Another alternative would be to remove the size field completely, saving space in the metadata table which could then be used to increase the size of the key field. However this would require reintroducing the freeable-pointers list from CETS to determine whether free is called with a pointer to the base of the freed object, instead of with a pointer pointing halfway into the object.

Another alternative design would be to to keep the CETS per-pointer copy of the key separate from the pointer, instead of storing it in the upper bits of the pointer itself. This would save a number of bitwise operations per check, however this would increase the amount of data that needs to be passed around on function calls or when copies of pointers are made.

Chapter 5

Implementation

To test the performance and security of the design given in the previous chapter, an implementation of the proposed design was written. In the sections below, a description of the reference implementation is given.

5.1 Intermediate Code Transformation

To implement the transformation of the initial code into a program containing the checks given in the previous chapter, the LLVM compiler framework was used to rewrite the compiled program [6]. The LLVM compiler framework initially transforms the given program into an intermediate representation. We can then transform this intermediate code. Finally this new intermediate code is given to the LLVM backend, which can transform the intermediate code into a native executable.

5.1.1 Per-opcode Transformation

The first step in implementing the given design, is updating instructions required to perform pointer arithmetic. In LLVM pointer arithmetic is performed using the `getelementptr` instruction. After every `getelementptr` instruction, we insert a call to routine described in section 4.3. This way we insert checks on every pointer arithmetic operation.

The next instructions to update are the `load` and `store` instructions, which perform a pointer deference. Before every `load` and `store` instruction, we insert a call to the routine described in section 4.4. Using this method we can perform a temporal check before every pointer dereference.

5.1.2 Stack Allocation Transformation

We will also need to update the LLVM instruction `alloca`. This instruction is responsible for allocation of stack memory. We need to transform this instruction to align the amount of memory allocation to a power of two, and change the size of the allocation object to the same power of two. We accomplish this by transforming the instruction into an `alloca` instruction allocating the aligned number of bytes, and inserting a cast to the

original type allocated. We then replace all uses of the old allocation with the transformed allocation. Next we have to insert a call to the allocation routine described in section 4.5 after the new `alloca` instruction. This way we have successfully setup metadata for all stack-based data.

However we still need to insert a call to the deallocation routine when the stack data is deallocated. We do this by keeping a list of all allocation calls inserted, and inserting a call to the deallocation routine described in section 4.5 whenever we encounter the exit of a function. In LLVM, functions are terminated using the `ret` instruction. So to clear all metadata whenever a stack object is deallocated, we insert deallocation calls before every `ret` instruction.

5.1.3 Global Data Transformation

To defend any global variables, we have to insert calls to the allocation routines described in section 4.5 at the start of every program. We also align all global objects to the nearest power of two of their object size. By inserting these calls and aligning the objects, we can ensure that all global data has metadata initialized.

5.1.4 Heap Allocation Transformation

Our compiler will also have to detect calls to heap allocation routines and update them to generate metadata for the allocated objects. We do this by detecting any calls to the C memory management functions malloc, realloc and free, and replacing them with calls to our own custom versions of these functions, described in section 5.2. These versions fill the metadata table with data describing the allocated objects, thus allowing heap data to be used in our checks.

5.1.5 External Module Compatibility

To remain compatible with external modules, we have to clear the key field of any pointer passed to an external library, such that the external library can still use the pointer as if it were a raw pointer. We do this by detecting any `call` instructions going to external modules, and inserting a bitwise and which clears the top bits of the pointer for any pointer argument passed to such functions. This ensures that external libraries can still use pointers to data in our programs.

5.2 Runtime Routines

We also implemented a runtime library which contains the functions required to update the runtime metadata table of the program. This library consists of all functions described in chapter 4. Furthermore it also contains updated versions of the C language heap allocation functions: malloc, calloc, realloc and free. For the memory allocating functions malloc, calloc and realloc, the replacement functions force the alignment of allocated objects to be on the nearest power of two of their object size, thus forcing them to be compatible with Baggy Bounds Checking. Furthermore they also contain a call to the metadata allocation routine described in section 4.5, so that metadata on newly allocated objects is inserted. The functions used to replace free on the other

hand contains a call to the metadata deallocation routine described in section 4.5. Using these new functions we can force C programs using the standard library to allocate data on the heap to update the global metadata table when required.

Finally, to mitigate the problem of library compatibility for some of the routines of the standard C library, described in more detail in section 7.5, the runtime support library contains overloads for some problematic functions.

The functions below return pointer to elements of possibly protected buffers. Simply removing the key from the pointer in order to use this functions would result in the functions returning a raw pointer without a key to a possibly protected buffer, which would cause any CETS check to fail due to a mismatch in keys. These functions have been modified to correctly copy the key into their results:

- strtok
- strchr
- memchr
- strcat
- strncat
- strstr
- strcpy
- strncpy

5.3 Optimizations

Our implementation also includes some further optimizations to reduce the number of runtime checks when we can prove these checks unnecessary. In our implementation, we keep track of all stack-allocated objects in our current function at compile time, and determine their size. If we later encounter an arithmetic operation on a pointer to this allocated object, we can verify at compile time whether this check will fail or succeed. If we can prove a check to always succeed, we will not have to insert the check. Baggy Bounds Checking implements a similar optimization, which attempts to eliminate unnecessary checks to increase overall performance.

Checks performed on dereference can also be redundant. Our implementation implements a version of unnecessary temporal check elimition as implemented for CETS. Our implementation attempts to figure out at compile time whether a given argument to a load or store instruction points to a stack-allocated object in the current function. If so, this object will always be available during the execution of the current function, as it is only deallocated at the end of the function. Thus if we encounter such a load or store instruction, we can remove the temporal check. Furthermore, since any global objects are always allocated, temporal checks on global objects can never fail. Thus we can remove any temporal checks on global objects.

Finally, we can also remove some metadata allocation and deallocation routines. When all checks regarding a given object can be eliminated, and no pointer to this object can be used by any code outside of this function, for example by passing a pointer to this object as an argument or storing a pointer to this object in a global

variable. We can ensure that the metadata of this object is never accessed. When the metadata of an object is never accessed, we do not need to initialize the metadata for this object, thus we can remove the allocation and deallocation routine.

Chapter 6

Evaluation

The evaluation of a defense mechanism can focus on multiple aspects. For our evaluation of our proposed defense mechanism, we evaluated the runtime performance of programs compiled with our proposed defense mechanism built into them. Furthermore we evaluated how effective our defense mechanism is at prevent exploitation. Finally we also measured how effective our proposed optimizations were by analyzing how many checks they were able to eliminate.

6.1 Performance

To evaluate the effectiveness of the combination, we benchmarked the performance of the new combination using the SPEC CPU2006 benchmark [9]. The SPEC CPU2006 benchmark contains a number of modified real-world applications. Using this benchmark we can determine the overhead incurred when a real-world application would use our designed defense.

To determine the overhead, we ran the benchmarks on the Leiden University cluster of the DAS-5 cluster computer [3]. This cluster contains Intel Xeon E5-2630 cores, running at 2.40GHz. During the tests we used 16 of the 24 nodes, each node containing a dual 8-core processor. Each node has 64GB of memory available. While running the tests, the cluster was running CentOS version 7.4.

6.1.1 Limitations

We were unable to successfully run all benchmarks included in the SPEC CPU2006 benchmark suite. The 403.gcc benchmark used non-standard GNU extensions to the C language, which our defense mechanism is unable to handle correctly. During execution of a number of the other benchmarks, we encountered a problem caused by the LLVM optimizer transforming bytecode in such a way that breaks guarantees required by the C standard. An example of this can be found in the 433.milc benchmark, which contains a function named `load_longlinks`, which contains a piece of code that could be simplified to code given in listing 6.1.

In this code, `data[i].field` is always between 0 and 7, making this code valid as no illegal accesses should

```
int buffer[4] = {0};

if(data[i].field < 4)
    buffer[data[i].field]++;
else
    buffer[7 - data[i].field]--;
```

Listing 6.1: Simplified version branching code appearing in the milc benchmark

```
int buffer[4] = {0};

bool condition = data[i].field < 4;
size_t index = data[i].field;
int* ptr1 = &buffer[index];
int* pt2 = &buffer[7 - index];

int* result = select(condition, ptr1, ptr2);
int value = *result;
*result = select(condition, value + 1, value - 1);
```

Listing 6.2: Transformed version of branching code after optimizations

be performed. During optimization however, LLVM transforms the code into a construction similar to the code given in listing 6.2.

In this code, `select` represents the LLVM instruction select, which is used to generate conditional code without introducing any branches into the intermediate code. This instruction can then in later stages be optimized into conditional moves if the target supports them. However, due to the modification of the original code, instead of performing pointer arithmetic within a branch, the pointer arithmetic is now performed unconditionally, by calculating both `&buffer[index]` as wel as `&buffer[7 - index]`. When our defense inserts checks into this code, it tries to check whether either of these two goes out of the bounds of the original buffer, which has only a size of 4. Since this happens unconditionally, this will always cause our check to fail, thus preventing the program from running correctly.

A similar problem also occurs during the 458.sjeng benchmark in the function `order_moves`, in which the pointer to an array element is calculated outside of a conditional branch, this pointer could however point past the end of the buffer depending on a number of conditions which are all checked later. Another example of this type of optimization reordering instructions to precompute pointers which can go out of bounds can be found in the function `IntraChromaPrediction` in the 464.h264ref benchmark. This instance is similar to the optimization found in the 458.sjeng benchmark. Another pointer to an array element is precomputed at the start of a loop body, this calculation can however go out of bounds. In this instance, the loop body contains a switch statement which verifies that all conditions preventing the out-of-bounds access are met before performing the actual array access, however during optimization the address calculation is assumed not to have any side effects, and thus optimized into a single calculation at the start of the loop body.

A possible solution to this problem could be to change the Baggy Bounds Checking routines to return an invalid pointer constant when an arithmetic operation goes out of bounds. During a dereference we could then

```
void
_E__pr_info( char const *fmt, ... )
{
    va_list pvar;

    va_start(pvar, fmt);
    (void) vfprintf(stdout, fmt, pvar);
    va_end(pvar);

//SPEC    (void) fflush(stderr);
}
```

Listing 6.3: A problematic vararg print function found in the spinx3 benchmark

check whether the pointer equals this constant, and abort the program then if required. This would however introduce a bit more overhead to the target program. First of all, every Baggy Bounds Checking routine would have to propagate this invalid value, introducing a new check at the start of every Baggy Bounds Checking routine. Second of all, every CETS routine has to introduce a new conditional branch to check whether the pointer passed to it matches the constant, and abort if it does match. This second check could be combined with the existing CETS routines by making sure the metadata table contains an entry for the invalid pointer value, in which the key field does not match the key in the invalid pointer constant used to signal that a Baggy Bounds Checking routine failed. This would cause the CETS check to fail, thus still correctly aborting the program.

Next, we encountered a set of problems caused by the LLVM backend which transforms intermediate code into native machine code. This backend makes a number of assumptions about the intermediate code, which do not hold when our defense is used. The main assumption made by the backend is based on the same fact we used to design our implementation, namely that pointers on our target platform only use 48 of the 64 bits available to them. The native code generator assumes that the 16 most significant bits in any pointer value are a sign extension of bit 47, as would be the case when our defense is not used. However, since we store the CETS key in the 16 most significant bits, this assumption does not hold during arithmetic. When the native code generator starts optimizing pointer arithmetic, it does not take into account the fact that this assumption does not hold, thus generating instructions that either change the value in the higher 16 bits, or completely modify the value of the pointer, causing later checks using this pointer to fail. An example of this can be found in the 400.perlbench test, in which the backend optimized the pointer arithmetic in the function `Perl_sv_add_arena` to use operations which cleared all the top bits of any pointers used, thus causing our checks to fail and abort the program. A similar construct is generated for the function `refresh_neighbour_lists` in the 429.mcf benchmark.

The 482.sphinx3 benchmarks failed due to library incompatibility issues. This benchmark makes heavy use of vararg functions when printing information, which are then forwarded to the external standard C library routine `vfprintf`. An example of which can be found in listing 6.3, which contains the `_E__pr_info` function from this benchmark.

This function is then called using a number of `const char*` arguments, which are then dereferenced in the

Benchmark	Status	Reason	Location
400.perlbench	Failed	Native code optimizer	Perl_sv_add_arena
401.bzip2	Success		
403.gcc	Failed	Use of GNU extensions	Throughout the entire program
429.mcf	Failed	Native code optimizer	refresh_neighbour_lists
433.milc	Failed	Optimizer pre-calculation of out-of-bounds pointer	load_longlinks
445.gobmk	Failed	Out-of-bounds pointer in benchmark	gg_sort
456.hmmer	Success		
458.sjeng	Failed	Optimizer pre-calculation of out-of-bounds pointer	order_moves
462.libquantum	Success		
464.h264ref	Failed	Optimizer pre-calculation of out-of-bounds pointer	IntraChromaPrediction
470.lbm	Success		
482.sphinx3	Failed	Library incompatibility issues	_E__pr_info

Table 6.1: Status of the SPEC CPU2006 benchmarks

external library function `vfprintf`. However, since these pointers still contain their metadata key in their top bits, dereferencing these pointers would result in a segmentation fault. Usually our implementation would clear the top bits of these pointers to ensure that external libraries can still dereference these pointers, however due to the layer of indirection through a vararg function, our implementation cannot detect that any pointers passed to the `_E__pr_info` function are directly forwarded to an external library, thus it does not insert code to clear the top bits of these pointers, which results in the benchmark crashing due to a segmentation fault in an external library.

Finally the 445.gobmk benchmark failed to run due to an arithmetic operation that went out of bounds, but is never dereferenced. The function `gg_sort` function, which has the signature
`void gg_sort(void* base, size_t nel, size_t width, int (*cmp)(const void*, const void*))`, is supposed to sort a buffer given to it in its base parameter. This buffer is supposed to contain nel elements of size width. The function starts by calculating a pointer to the last element in the buffer by using the following operation: `char* end = (char*)base + width * (nel - 1)`, which would hold when the buffer has at least one element. However, when this function is called from the function `aa_sort_moves`, the nel parameter can be zero, which would cause this equation to result in a pointer to width bytes before the start of the buffer, which our Baggy Bounds Checking checks successfully catch. Due to the nature of the function, this pointer is never dereferenced when the buffer has no elements to sort, which is why the function would succeed when pointer arithmetic is not checked. The proposed solution to the problem encountered using the LLVM select instruction could also resolve this issue without the need to modify the original code.

An overview of every SPEC CPU2006 C benchmark describing if the benchmark was successfully executed or not, and if not what the cause and location of the failure were can be found in table 6.1.

6.1.2 Results

We compiled the remaining benchmarks using both the implementation described in section 5, and using the clang 4.0.0 C compiler. Each benchmark instance was run 32 times. The results of the runs are given in the table 6.2.

	Default				Baggy Bounds Checking + CETS				
Benchmark	**average**	**min**	**max**	σ	**average**	**min**	**max**	σ	**overhead (%)**
401.bzip2	435.9	434	438	1.045	3193.7	3138	3205	6.213	632.7
456.hmmer	383.3	382	385	0.644	4384.2	4373	4400	6.816	1043.8
462.libquantum	414.2	406	421	4.146	1923.4	1916	1931	4.457	364.4
470.lbm	357.0	353	364	3.610	1087.3	1082	1096	3.547	204.6

Table 6.2: Benchmark Results for SPEC CPU2006

	Only Metadata		Metadata + Baggy Bounds Checking		Metadata + CETS	
Benchmark	**average**	σ	**average**	σ	**average**	σ
401.bzip2	567.4	1.132	2045.1	3.541	1779.2	2.533
456.hmmer	524.3	0.759	2648.2	4.580	2043.4	3.957
462.libquantum	462.3	4.183	947.3	2.624	1276.8	2.562
470.lbm	349.9	1.711	795.2	2.776	791.1	4.003

Table 6.3: Benchmark Results for SPEC CPU2006 with our defense partially enabled

From this table, we can conclude that our implementation has quite a significant runtime overhead. Baggy Bounds Checking has an average runtime overhead of 60% [2], and CETS has a runtime overhead of around 48% [8], meaning the combination adds a significant overhead on both defenses. Even compared to the overhead of Softbounds combined with CETS, which has an overhead of 116% on average [8], our implementation is significantly slower.

To determine the cause of the performance overhead, we measured the overhead of each separate component of our combined defense. We compiled the benchmarks we evaluated earlier with only the metadata allocation routines built in, with only the baggy bounds checking and metadata allocation routines built in, and with only the CETS checking and metadata allocation routines built in. Using this method, we can determine more accurately which component of our defense impacts the overall performance the most. Each benchmark instance was run 32 times. The results of these runs are shown in table 6.3.

From these results, we can conclude that our metadata allocation routines add a small runtime overhead to the target program. For the bzip2 and hmmer benchmarks, we found that our Baggy Bounds Checking routines added a significant runtime overhead to our target program, while a smaller portion of the runtime overhead came from the CETS checking routines. For the libquantum benchmark instance, the CETS checking routines resulted in the most significant runtime overhead, whereas the Baggy Bounds Checking routines added a smaller runtime overhead. Finally for the lbm benchmark, both Baggy Bounds Checking and CETS have around an equal performance impact on the runtime of the program.

By analyzing the benchmark programs, we found that a significant part of the runtime overhead came from pointer operations our program could not prove to be safe, such as passing a pointer to buffer along with its size to a function, then using the buffer in the new function. Since our implementation naively inserts bounds checks on any operation involving this buffer, this led to quite a significant performance overhead on applications that performed large amount of operations on buffers, like the bzip2 and hmmer benchmarks, as also shown by the Baggy Bounds Checking overhead for these programs found in table 6.3. Programs that instead used fewer of such routines, like the lbm benchmark had a significantly smaller runtime overhead.

```
void fill_buffer(int* x, size_t size) {
    for(size_t i = 0; i < size; ++i) {
        x[i] = i;
    }
}

int main() {
    int buffer[10];
    fill_buffer(buffer, 10);
    ...
}
```

Listing 6.4: Example of hard to optimize code

```
unsigned int x, y;
int buffer[10];
...
x = x % 5; /* x <= 4 */
y = y % 5; /* y <= 4 */

int result = buffer[x + y]; /* x + y <= 8 */
```

Listing 6.5: Example of arithmetic guarantees on pointer operations

The runtime overhead of the current implementation could likely be reduced by implementing a more sophisticated optimization mechanism to eliminate runtime checks. By performing a program-wide pointer analysis, functions accepting pointers to a buffer and a correct buffer size as argument could be identified more easily, allowing many of the currently redundant checks to be eliminated. An example of a function which currently cannot be optimized is given in listing 6.4.

Our current implementation would naively insert both a bounds check and a dereference check for the array index `x[i]` in the loop, while we can guarantee that both checks will succeed in this code.

Another optimization could be to keep track of the set of values any variable used for pointer arithmetic could have. This would allow us to remove unnecessary bounds checks when we can guarantee that all bounds checks will succeed. An example is giving in listing 6.5.

In this example, the values of x and y are bounded. From this code, we can conclude that the array access `buffer[x + y]` would always be valid, thus any bounds check should succeed. However our current implementation cannot detect this, and would thus insert a bounds check for this array access. Analyzing the program to remove these checks could significantly improve performance.

6.2 Security

To determine the effectiveness of our defense mechanism, we measured the security using a CVE framework [11]. We compiled the programs provided by the CVE framework using our modified LLVM toolchain to insert defense code into the target programs.

	Loads		Stores		GetElementPtr		AllocA	
Test Name	**Base**	**Optimized**	**Base**	**Optimized**	**Base**	**Optimized**	**Base**	**Optimized**
bzip2	2854	2755	1900	1787	2499	2461	25	24
hmmer	5075	4825	2552	2404	4282	4250	87	78
libquantum	277	166	207	59	229	171	3	3
lbm	221	204	101	98	279	279	7	7
CVE-2017-8364	435	351	147	90	299	245	56	36

Table 6.4: Optimization Results

Since our current implementation however lacks supports for shared libraries, a number of the provided CVE tests failed to run successfully. We tested the remaining test cases, containing a stack overread bug, a heap overread bug and CVE 2017-8364. We managed to defend both the programs containing the stack overread and heap overread.

Due to the nature of the bug in CVE 2017-8364, which contains a buffer overflow which can be triggered by setting the size header in the example input file to a size larger than the size of the internal buffer used. However, due to Baggy Bounds Checking, the size of the internal buffer is aligned to the nearest power of two greater than the original buffer size. Since the CVE framework set the size to a value within the range of this padding, this causes the program to continue running without being exploited. We fixed this by modifying the exploit to use a larger size in the header of the input file, which once again triggered a buffer overflow, which our defense managed to catch.

6.3 Optimizations

To determine the quality of the optimizations, we measured the number of checks each optimization removed from a given benchmark. We analyzed the number of temporal checks inserted on store and load instructions, the number of bounds checks inserted on getelementptr instructions, and the number of metadata initialization routines inserted on the stack memory allocation instruction alloca.

We analyzed this for each of the benchmarks listed in table 6.2, and the rzip program of CVE 2017-8364 analyzed in section 6.2. The results are given in the table below.

From this, we can conclude that the optimizations can eliminate a number of checks. Our temporal check elimination seems to be effective in a significant number of cases, eliminating around 15% of the checks on loads and 25% of the checks on stores on average. Our spatial optimization seems to be a little less effective, being able to eliminate only 9% of all checks on average. Finally our metadata initialization routine optimization seems to be less effective, only being able to optimize a small number of cases.

Chapter 7

Limitations

The current implementation described does however pose some limitations.

7.1 Setjmp

Metadata on local variables has to be cleaned up on function exit to ensure that further usage of pointers to the addresses used for local variables are no longer valid. Normally this data is cleaned up on function exit, however a routine that bypasses normal function control flow like a setjmp does not run any of the cleanup routines which should be run on function exit, which leaves the data in the metadata table invalid.

When the cleanup routines are bypassed using a setjmp, the metadata table will still contain valid keys for objects that are no longer valid. This would allow an attacker to use pointers to the now invalid objects without the defense being able to catch the invalid use of these pointers, which could allow the program to be exploited.

7.2 Small Key Space

Since our current implementation only allows for 512 distinct key values, of which one is used to mark invalid keys, the chances for the reuse of keys for distinct objects are significant. When two objects are assigned the same key value, any CETS checks will not be able to differentiate between the two objects. When an object with a given key is deallocated, and later another object is allocated at the same memory location with the same key value, using invalidated pointers to the old object to access the new object will now succeed due to the keys being the same. This would allow attackers make use of these temporal errors to exploit the code.

7.3 Concurrency

Our current implementation is not safe for use in concurrent processes. Since a global table is used to keep track of all metadata relating to all objects in the program, concurrent programs would have to concurrently

modify this global table. Allocation and deallocation across different threads could lead to metadata table entries containing invalid data, which could cause both spatial and temporal checks to fail for valid accesses, or to succeed for invalid accesses. This could either cause valid programs to abort, or allowing invalid programs to be exploited.

7.4 Sub-Object Safety

Like Baggy Bounds Checking, our design does not enforce sub-object safety. A solution for programs that only read from single fields at a time could be to regard every field as a single object. However this could break compatibility with existing code, an example of which is given below:

```
struct X {
    uint8_t r;
    uint8_t g;
    uint8_t b;
    uint8_t a;
};

X x;
uint32_t v = ...;
memcpy(&x, &v, sizeof(x));
```

The code above is an example of a type punned structure often found in current C code. In this example, a structure consisting of four 8-bit integers is assumed to fit in one 32-bit integer. However, to guarantee sub-object safety, our design would have to pad all fields of this structure to a minimum of one slot size. If one slot is larger than 8 bits, the assumption that four 8-bit integers fits in one 32-bit integer would no longer hold, as the 8-bit integers are no longer tightly packed.

7.5 Library Compatibility

By storing the copy of the per-object key within the pointer, directly passing such a pointer to an external library may cause issues if the library attempts to dereference this pointer. To mitigate this problem, a mask can be used to erase the per-object key from the pointer, thus transforming it into a raw address. This however does cause a security risk, as the external library can still perform unchecked reads and writes on this pointer.

Another problem arises when a library passes an unsafe pointer into the safe code. When using such pointers, no metadata on the pointer exists, and thus all checks would fail, breaking existing code. This problem can be solved by checking whether the top 16 bits of the passed pointer are zero, as they should be when obtaining a pointer from an external library. Since the key value of zero represents an invalid pointer value, which can only occur within the per-object metadata table as it represents an invalid key, and thus can never be associated

with a valid safe pointer, we can assume a value of zero to mean that the pointer was from an external library, and thus allow all operations on the pointer to succeed. This however does however pose a security risk.

Chapter 8

Related Work

In the past, numerous defenses which can protect applications against the exploitation of bugs have been proposed. Systems which can improve memory safety include Softbound [7], which has been formally proven to guarantee full memory safety, unlike our implementation, which cannot guarantee subobject safety. Another example is Baggy Bounds Checking [2], which as discussed earlier can prevent a large number of memory errors. Softbound and Baggy Bounds Checking can not gurantee temporal safety however.

Defenses that can guarantee temporal safety have been proposed in the past. Compiler Enforced Temporal Safety [8] ensures that no temporal errors occur, it however requires a guarantee of spatial safety to ensure temporal safety. Our proposed implementation can guarantee both temporal safety and a subset of spatial safety, using Baggy Bounds Checking to provide the spatial safety guarantee CETS needs.

Other combinations of defenses that can guarantee both temporal and spatial safety have been proposed. Softbound and Compiler Enforced Temporal Safety have historically been combined [8], which helps provide the guarantee of spatial safety CETS requires to enforce temporal safety. Softbounds combined with Compiler Enforced Temporal Safety has been found to have an average overhead of 116%, making it more efficient than our current implementation.

Other defenses focus on preventing a subset of errors by preventing attackers from hijacking control flow. Examples include Shadow Stacks and Stack Cookies, which are designed to prevent control flow hijacking by overwriting function return addresses on the call stack. Another example is Abadi Control Flow Integrity, which identifies valid targets for function calls and returns, and can thus prevent attackers from reaching invalid targets [1]. However, both of these examples can only enforce a subset of spatial safety, focussing on the common attack pattern of overwriting return addresses. They however provide no protection against more specialized attacks, such as leaking data from the target program by using buffer overreads. Our current implementation provides a stronger safety guarantee, by detecting any out-of-bounds pointers we can make a stronger guarantee of spatial safety, allow us to prevent buffer overreads to prevent data leaking.

Another set of defenses focus on preventing commonly exploited data types and structures. Code Pointer

Integrity for example identifies the set of addresses a pointer can point to, and prevent attackers from reaching objects that would normally be unreachable from a given instruction [5], this does however still allow attackers to reach any object in the points-to set, even if the programmer did not intend the user to access them. Our defense prevent any invalid pointers from being obtained from a valid pointer, thus preventing any attacker from accessing any location outside of the intended object.

Chapter 9

Conclusions

Numerous defenses exist that help solve the current security risks of applications written in unsafe low-level languages. Since these applications can contain a wide variety of bugs, defenses have been proposed to prevent malicious users from exploiting the bugs in these applications, two of which are Baggy Bounds Checking en Compiler Enforced Temporal Safety. In this paper, we proposed a defense that combines Baggy Bounds Checking and Compiler Enforced Temporal Safety into one new combined defense.

We managed to combine the data structures of both defenses into one data structure that can be accessed using a simple array index. We also proposed optimizations to reduce the runtime overhead of the combined defense.

After evaluating our combined defense, we found that our combined defense managed to protect programs from a number of exploits, however we also found that our defense had a significant runtime overhead, which would make it unsuitable for practical applications.

9.1 Future Work

In the future, research can be done on reducing redundant checks present in our current implementation, which could significantly increase the performance of the current defense. Optimizations could include implementing a program-wide pointer analysis to determine whether certain bounds checks will always succeed, even across function calls. Other optimizations could include keeping track of the possible contents of any variable used to index to figure out whether a bounds check has a chance of going out-of-bounds, and eliminating the check if it does not.

Furthermore, research into more complex forms of control-flow analysis can be done. To mitigate the problems due to LLVM optimizations described in section 6.1, complex analysis of the dependencies of select statements is required to transform the generated intermediate code in such a manner that checks are only performed on pointer arithmetic results selected by such conditional statements. This would eliminate checks that would always incorrectly fail, thus ensuring that our defense can be used on more programs.

Further research can also be done into combining other defenses. By identifying overlap in current defenses and combing their data structures and runtime checks, it could be possible to create more efficient combined defenses.

Finally, research could be done on removing some of the restrictions current defenses impose on their target programs. By lessening the restrictions of a defense, it might become possible to combine the defense with other defense mechanisms, which could lead to more efficient combined defenses.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[2] 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.

[3] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[4] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM, 2014.

[5] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, 2014.

[6] The llvm compiler infrastructure. `https://llvm.org/`. Accessed: 01-06-2019.

[7] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[8] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[9] SPEC CPU2006 Benchmarks. `https://www.spec.org/cpu2006/`. Accessed: 01-06-2019.

[10] SQLExp SQL Server Worm Analysis. `http://securityresponse.symantec.com/avcenter/Analysis-SQLExp.pdf`, 2003.

[11] Elgar R. van der Zande. Examining out of bounds defense systems' performance against independent cves, 2019.