Color Reclamation for Heap Memory Coloring Scheme in PIPE Tagged-Memory Architecture

by

Jiahao Li

S.B., C.S. M.I.T., 2018

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2019

Author .....
Department of Electrical Engineering and Computer Science
August 12, 2019

Certified by .....
Howard E. Shrobe, Thesis Supervisor
Principal Research Scientist, MIT CSAIL

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Color Reclamation for Heap Memory Coloring Scheme in PIPE Tagged-Memory Architecture

by

Jiahao Li

Submitted to the Department of Electrical Engineering and Computer Science
on August 12, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Memory safety violations remain a significant hurdle as people try to secure computer software. The PIPE project proposed an approach to securing software by tagging memory and registers with color tags and defining policies in terms of the tags to disallow instructions violating memory safety. However, the PIPE project did not include a scheme for reclaiming and reusing allocated color tags, thus resulting in issues including color wrap-arounds.

This thesis proposes a scheme for reclaiming and reusing color tags within the PIPE tagged-memory architecture. The proposed scheme is then evaluated on a combination of synthetic microbenchmarks and real-world allocation trace replays. Evaluation results show that the reclaimer can effectively reclaim colors in all cases, and in some cases can also improve performance by increasing rule cache hit rate.

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist, MIT CSAIL

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) System Security Integration Through Hardware and Firmware (SSITH) project through a Draper Laboratory subcontract. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the the Defense Advanced Research Projects Agency or Draper Laboratory.

Acknowledgments

First and foremost, I would like to extend my sincerest thanks to Dr. Howard Shrobe and Dr. Hamed Okhravi for their help and support. I have benefited immensely from their mentorship and insights, both within and out of the scope of this thesis project. I am especially indebted to Dr. Shrobe for his guidance and many valuable discussions on the design decisions behind the color reclamation scheme.

I would also like to thank the rest of the Inherently Secure group as well as the HOPE research project group. I am especially thankful for the thoughtful help and answers I received from Chris Casinghino, Arun Thomas, Greg Sullivan, Nathan Burow, Bryan Ward, Samuel Jero, Scott Brookes, Austin Harris, Justin Restivo, and Baltazar Ortiz.

I have received nothing but support from my parents, Pinduan and Zheng, throughout this past year and many before. I would not be where I am today without their love and encouragement.

I would like to extend my special thanks and appreciation to Xinyi Chen for her love and support during the last few years. Thank you for always being there with me.

Finally, I would like to thank all my friends and family that have been such an integral part of my life. Your kindness and companionship have been invaluable to me.

Contents

List of Figures

Chapter 1

Introduction

Securing computer software has been an uphill battle, with new vulnerabilities continuously being exposed as old ones are fixed. In particular, we see frequent recurrences of one class of vulnerabilities – memory safety violations. Such vulnerabilities are frequently leveraged by attacks to achieve various malicious effects, including disclosing sensitive data and hijacking the program's control flow.

There are two major classes of memory safety violations – spatial memory safety violations (e.g. buffer overflows) and temporal memory safety violations (e.g. use-after-free accesses). In buffer overflows, the vulnerable program – usually due to a lack of bounds checking – is tricked into accessing a memory buffer at an index beyond the actual size of the underlying buffer, causing information disclosure vulnerabilities (in cases of read accesses) and memory corruptions (in cases of write accesses). Since 2018, buffer overflows alone account for more than 900 vulnerabilities in the Common Vulnerabilities and Exposure (CVE) database [1], affecting a wide spectrum of applications from web

servers [2, 3] to surveillance camera firmware [4, 5].

With use-after-free accesses, the vulnerable program inadvertently accesses a memory buffer that it has already deallocated. Since the underlying memory region might have already been handed out as another buffer, inadvertently accessing this memory region can cause unexpected information disclosure (in cases of read accesses) or memory corruption (in cases of write accesses). The CVE database includes more than 500 use-after-free vulnerabilities entered since 2018 [6], affecting applications including the Linux kernel [7, 8] and desktop browsers [9, 10].

Such frequent recurrences of memory safety vulnerabilities is partly owed to conventional processor architectures' inability of preventing ill-behaving memory accesses. Instead, programmers are trusted to avoid memory safety violations through meticulous coding and testing, which in practice are usually insufficient.

There have been attempts at more systematic ways to detect and prevent memory safety violations, either through static analysis at compile time [11, 12, 13] or dynamic instrumentation at runtime [14, 15, 16]. However, solutions based on static analysis are usually limited in the types of errors they can detect. Furthermore, they often suffer from a large number of false-positives [17]. On the other hand, solutions based on dynamic instrumentation, while able to detect more errors, require an instrumented version of the binary to be compiled. This instrumented binary usually comes with significant overhead in both its runtime performance and memory usage, therefore limiting its adoption in the production environment.

Another avenue towards eliminating ill-behaving memory accesses

involves modifying the processor architecture to disallow invalid accesses on the hardware level. For example, PIPE [18] proposes an architecture where each word of data (in memory, in cache, and in CPU registers) is associated with a tag. Policies can be defined in terms of tags to deny the execution of certain instructions. In particular, PIPE [18] proposes a heap coloring scheme to disallow memory safety violations. Such a hardware-level protection scheme, as shown in [18], can be implemented with acceptable overhead costs in terms of runtime slowdown and memory usage inflation, and is thus suitable for production environment adoption.

On a high level, PIPE's heap coloring scheme enforces memory safety by checking for color match on memory accesses. When the memory allocator allocates a new piece of memory, the newly allocated memory region is assigned and tagged with a globally unique *cell color*. A pointer pointing to the allocated memory region is tagged with a matching *pointer color*, and returned to the program that requested the memory allocation. Since only the allocated buffer region will have the correct *cell color*, instructions attempting buffer overflows will be denied execution due to color mismatch. Furthermore, when a memory region is deallocated and subsequently reused by the memory allocator to service a different allocation, it will bear a different *cell color*. Attempts to access this memory region via dangling pointers left from the previous allocation (i.e. use-after-free violations) will therefore be denied execution due to color mismatch. See section 2.3 for a more detailed description of the PIPE coloring scheme.

In the PIPE heap coloring scheme, each allocated memory block currently in use needs to be assigned a globally unique color value.

However, it is not specified how color values can be reclaimed and reused. Instead, each newly allocated memory block will be assigned a new color based on a monotonically incrementing color counter. Since a program can potentially make an unbounded number of memory allocations during its lifetime, this simple allocation scheme requires the use of infinite-width color values, making it impractical for real-world applications.

This thesis aims to improve upon PIPE's heap coloring scheme in [18], by devising a scheme to reclaim colors from memory blocks that are no longer in use, so that the reclaimed colors can be reused for future allocations. Since the number of memory blocks in use at any given time is bounded by the size of the address space (and often in practice by a significantly stricter bound), timely reclaiming color values would allow the PIPE heap coloring scheme to be implemented with fixed-width color values with a small number of bits. This would make the heap coloring scheme significantly more practical for adoption in real applications.

In summary, this thesis presents design and implementation of a color reclamation method for the PIPE heap coloring scheme. The implementation is tested in a QEMU-based simulator for the PIPE tagged architecture, over a combination of synthetic benchmarks and real-world memory allocation traces, to show its effectiveness in reducing the size of color values required. High-level design is presented in chapter 3, implementation details is presented in chapter 4, and simulator evaluation is presented in chapter 5.

Chapter 2

Background

Several pieces of background information are necessary in order to better understand the work carried out in this thesis. In this section, we will first talk about the types of memory safety violations that this thesis addresses. Then, we will present a brief description of the PIPE [18] policy framework. Finally, we will proceed to introduce the existing heap coloring scheme used in PIPE [18].

2.1 Memory Safety Violations

2.1.1 Spatial Memory Safety Violations

Spatial memory safety is a property where all memory dereferences are within bounds of their pointers' valid objects [19]. A common case of spatial memory safety violation is in the form of buffer overflow, which happens when a program tries to access a buffer at an index that is beyond the actual allocated size of the buffer. Such out-of-bound accesses can potentially access an unrelated memory region, causing information

```
char content[100];
char secret_key[32];

void read_content(char *output, size_t length) {
    memcpy(output, content, length);
}

char result[200];
size_t length = get_requested_length();
read_content(result, length);
send_reply(result, length);
```

Figure 2-1: Example of an information disclosure vulnerability caused by buffer overflow.

leakage (in the case of read accesses) and memory corruption (in the case of write accesses).

An example information disclosure vulnerability caused by a read-access buffer overflow is given in fig. 2-1. In this case, the attacker can gain access to `secret_key` by making a request with length 132. Due to the lack of bounds check in `read_content`, `memcpy` will read beyond the bounds of `content` into `secret_keys`, causing inadvertent information disclosure.

2.1.2 Temporal Memory Safety Violation

Temporal memory safety is a property where all memory dereferences are valid at the time of dereference [19]. A common form of temporal memory safety violation is use-after-free errors, where a dangling pointer is used to access a memory region that has already been deallocated. Such violations are especially problematic when the previ-

```
char *user_input = (char *)malloc(PASSCODE_SIZE);
// ...
free(user_input);

char *passcode = (char *)malloc(PASSCODE_SIZE);
get_passcode(passcode);
prompt_for_passcode(user_input);
if (memcmp(user_input, passcode) == 0) {
    // User-entered passcode matches the real passcode.
    do_privileged_thing();
}
```

Figure 2-2: Example of an memory corruption vulnerability caused by use-after-free access.

ously deallocated memory region has been re-allocated as an unrelated buffer. Use-after-free accesses in this case will cause information leakage/corruption on that unrelated buffer, similar to the spatial memory safety violation case.

An example memory corruption vulnerability caused by a write-after-free access is given in fig. 2-2. Following `free(user_input)`, the memory region previously occupied by `user_input` is considered handed back to the memory allocator. When the buffer `passcode` is being allocated, it is possible (and in this case especially likely since the size of both allocations are equal) that the same memory region previously occupied by `user_input` will be reused to service the `passcode` allocation. In this case, the later write into `user_input` will inadvertently modify `passcode` as well, causing the passcode check to unconditionally pass.

2.2 PIPE Tagged-Memory Architecture

Traditional computer processors have been unable to prevent certain classes of memory safety violations, including spatial memory safety violations (section 2.1.1) and temporal memory safety violations (section 2.1.2). This is due to traditional processors' lack of buffer bounds information (necessary for enforcing spatial memory safety) and buffer identity information (necessary for enforcing temporal memory safety).

To address this shortcoming, PIPE [18] proposes a tagged-memory architecture that allows the processor to store additional metadata with each word of data, and to enforce custom policies defined in terms of the metadata.

2.2.1 Metadata in PIPE

In the PIPE tagged-memory architecture, each word of data (including those in memory, in cache, as well as in CPU registers) is associated with a *metadata set*. Each *metadata set* consists of a list of *metadata tag*s. Two types of *metadata tags* are allowed:

- *Binary tag*: A *binary tag* is either present or absent in any particular *metadata set*. For example, a *binary tag* named `Writable` can be used to denote whether a word of memory is writable.

- *Value tag*: A *value tag* is either absent or present with a particular value in any particular *metadata set*. For example, having a *value tag* named `Color` with value range $\{0, 1, \ldots, 15\}$ would mean that each word of data would either not have any `Color`, or have a particular integer `Color` in the range $[0, 15]$.

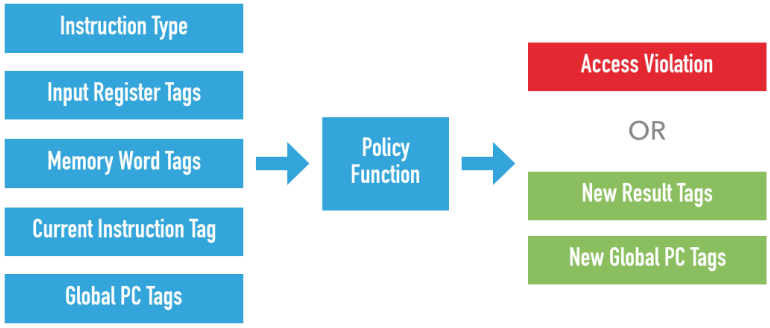


Figure 2-3: Inputs and outputs of the policy function.

Conceptually, a *metadata set* can contain an unlimited number of *metadata tag*s. In practice, *metadata set*s are stored out-of-line in a table in a separate memory area, and an index into that table will be stored within each data word as a layer of indirection.

2.2.2 Policy in PIPE

Once the data words in the system are augmented with metadata, PIPE allows a custom policy to be defined that can allow/deny the execution of certain instructions based on the metadata.

Conceptually a policy is a pure function. Its inputs and outputs are depicted in fig. 2-3. Here are brief descriptions of the 5 inputs to the policy function:

1. Instruction Type: Type of the instruction being executed (e.g. addition instruction, memory store instruction, etc.).

2. Input Register Tags: The *metadata set*s associated with the input registers of the instruction. For example, given an addition instruction, the input registers will be the 2 registers containing the operands being added together.

3. Memory Word Tags: This input is only used for memory load/store instructions. It represents the *metadata set* associated with the word of memory being loaded from/stored into.

4. Current Instruction Tags: The *metadata set* associated with the word of memory containing the current *instruction being executed* (i.e. the word of memory at the current program counter). This input allows the tagging of certain code regions to be treated differently from the rest of the program.

5. Global PC Tags: A special *metadata set* associated with the program counter register. In practice, this *metadata set* is used as a global variable to carry states not associated with any particular data word.

Given these 5 inputs, the policy function can produce two types of output:

1. Access Violation: The policy function determines that the instruction should be denied execution due to an access violation. In this case, the program halts in a fail-stop fashion to prevent further damage.

2. Access Allowed: The policy function determines that the instruction should be allowed to execute. In this case, the policy function can optionally modify:

   (a) Result Tags: The *metadata set* associated with the result of the instruction. For arithmetic instruction, this will be the

metadata set associated with the register containing the calculation result. For memory load/store instructions, this will be the *metadata set* associated with the register/memory word being loaded/stored into.

(b) Global PC Tags: The special *metadata set* associated with the program counter register as described above. Modifying this *metadata set* is used to change the global policy state in order to affect the policy function's processing of future instructions.

2.2.3 Policy Language Through Example: RWX Memory

Both metadata tag definitions and policy functions in the PIPE tagged-memory architecture are expressed in a specially designed policy language. This section will give a brief description of the major parts of the policy language through an example policy: Read/Write/Executable (RWX) Memory.

A simplified excerpt of the policy language code of the RWX Memory policy is shown in fig. 2-4. The excerpt shows only two sections (`metadata` and `policy` sections). Policy language code can contain other sections, but they are not significant for the purpose of this thesis.

The first section – `metadata` – lists all *metadata tag*s that the policy defines. In this case, three *binary tag*s are defined:

(a) `Rd`: Denotes whether the word of memory can be read from.

```
metadata:
  Rd
  Wr
  Ex

policy:
  rwxPol =
      loadGrp(mem == [-Rd] -> fail "read violation")
    ^ storeGrp(mem == [-Wr] -> fail "write violation")
    ^ allGrp(code == [-Ex] -> fail "execute violation")
    ^ loadGrp  (   code == [+Ex], env == _, mem == [+Rd]
                -> res = {}, env = env)
    ^ storeGrp (   code == [+Ex], env == _, mem == [+Wr]
                -> mem = mem, env = env)
    ^ allGrp(code == [+Ex], env == _ -> env = env)
```

Figure 2-4: Simplified excerpt of the policy language code of the RWX Memory policy.

(b) `Wr`: Denotes whether the word of memory can be written to.

(c) `Ex`: Denotes whether the word of memory contains instructions that can be executed.

The following section – `policy` – lists the policy clauses. Each policy clause is of the following format:

```
instTypeGroup(cond1, cond2, ... -> result)
```

`instTypeGroup` defines the types of instructions that the policy clause applies to. `loadGrp`, for example, denotes that the policy clause applies to memory load instructions only.

Following `instTypeGroup`, we have a comma-separated list of conditions that must be satisfied in order for the policy clause to

apply. Each condition matches a policy function input *metadata set* to a set of requirements, each of which requires the *metadata set* to either contain a particular *metadata tag* (in the form of `+TAG`), or *not* contain a particular *metadata tag* (in the form of `-TAG`). For example, the condition `mem == [-Wr]` states that the Memory Word Tags *metadata set* must *not* contain the tag `Wr`.

Finally, following the `->`, we have the policy clause result. The result can either allow the current instruction to execute, or deny its execution with a message. In the case of allowing execution, the policy clause can also specify how it would like to modify the Result Tags and Global PC Tags in the result part after `->`.

As an example, consider the following policy clause (line $11 - 12$ in fig. 2-4):

```
loadGrp(code == [+Ex], env == _, mem == [+Wr]
     -> mem = mem, env = env)
```

This policy clause would match if:

(a) the instruction is a load instruction,

(b) its Current Instruction (`code`) *metadata set* contains the `Ex` tag, and

(c) its Memory Word (`mem`) *metadata set* contains the `Wr` tag.

When this policy clause matches, it will allow the current instruction to execute. Furthermore, it will keep both the Memory Word *metadata set* and the Global PC *metadata set* unmodified (due to the `mem = mem, env = env` part).

Policy clauses connected by the ^ operator are evaluated in declaration order. The first matching policy clause will determine the outcome. If none of the policy clauses matches, the instruction will be implicitly denied.

2.3 PIPE Heap Coloring Policy

2.3.1 Coloring Scheme

PIPE Heap Coloring policy uses two *metadata tag*s: `CellColor` and `PointerColor`. Both `CellColor` and `PointerColor` are *value tag*s with integer color values. Each time the heap memory allocator allocates a new buffer, a `CellColor` tag value is allocated and assigned to words in the buffer region. A `PointerColor` tag with the same integer color value is assigned to the pointer returned by the memory allocator.

Each time the program attempts to access a memory location on the heap via a pointer, the `PointerColor` on the pointer is compared with the `CellColor` on the memory word being accessed. If the colors do not match, or if either color is absent, the policy denies the memory load/store instruction from executing.

Consider an example buffer overflow as shown in fig. 2-5. Assume that `buf` and `secret` have been allocated right next to each other in memory. Since they are allocated separately, `buf` and `secret` will have two different `CellColor`s. When the program tries to access `buf[2]`, the expression `buf[2]` will have the same `PointerColor` as `buf`. Since this `PointerColor` does not match the `CellColor` on the actual memory word being accessed, this load instruction will be denied.

```
1 int *buf = (int *)
2   malloc(2 * sizeof(int));
3 int *secret = (int *)
4   malloc(2 * sizeof(int));
5 int value = buf[2];
```

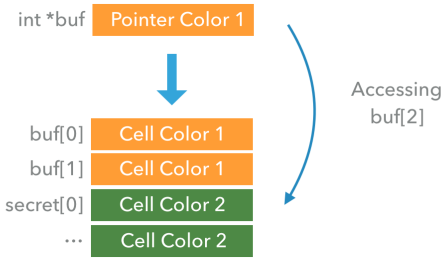
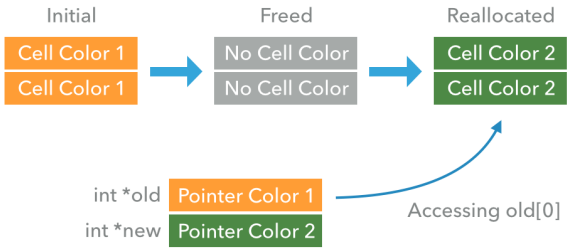


Figure 2-5: An example buffer overflow access prevented by the PIPE heap coloring policy.




```
1 int *old = (int *)(malloc(2 * sizeof(int)));
2 free(old);
3 int *new = (int *)(malloc(2 * sizeof(int)));
4 int value = old[0];
```

Figure 2-6: An example use-after-free access prevented by the PIPE heap coloring policy.

Similarly, consider an example use-after-free access as shown in fig. 2-6. Assume the second allocation (`new`) is handed the same memory region that was just deallocated. Upon deallocating a heap memory region, the memory allocator will zero out its `CellColor`. When the same memory region is handed out again, it will be tagged with a different `CellColor`. Therefore, when the program attempts to access the deallocated buffer via the dangling pointer `old`, the load instruction will be correctly denied execution.

2.3.2 `CellColor` Allocation

Each time a new buffer is allocated, a new `CellColor` needs to be allocated and assigned to it. In order for PIPE's protection against memory safety access to be effective, we must ensure that:

1. the allocated `CellColor` must not currently be used by any other buffer, and that

2. there must be no dangling pointer with a matching `PointerColor` currently present anywhere in memory.

A trivial allocation method for new `CellColor`s that satisfy both constraints is to keep a monotonically increasing color counter. Each time we allocate a new buffer, we use the current counter value as its `CellColor`, and subsequently increase the counter value by 1. This is the current method employed in the PIPE heap coloring scheme.

However, in a practical implementation of the heap coloring policy, `CellColor`s and `PointerColor`s are bounded in range. Therefore, in order to support a potentially unbounded number of memory allocations

throughout the program's lifetime, the color value wraps around to 0 once it hits the maximum color value. Such wrap-arounds, however, break the aforementioned two constraints, thus making the coloring scheme less secure due to the possibility of unexpected color collisions.

To address this deficiency, this thesis aims to provide a new color allocation and reclamation scheme that supports unbounded number of memory allocations while still maintaining the safety properties by adhering to the two constraints.

Chapter 3

Design

The PIPE tagged-memory architecture employs two separate processors: an *application processor* that executes regular application code, and a *policy processor* that evaluates the policy function to determine whether to allow instructions to execute. Conceptually, the *application processor* and *policy processor* access disjoint parts of the memory: the *application processor* only has access to the regular data part of the memory, and the *policy processor* only has access to the metadata part of the memory.

The color reclamation scheme proposed in this thesis works in 2 parts: the *synchronous part* and the *asynchronous part*. In the *synchronous part*, when the memory allocator on the *application processor* is about to deallocate a chunk of memory, it notifies the *policy processor* of its intent of releasing the previously occupied color. In this part, the *policy processor* carries out minimal amount of clean-up (more details are given in section 3.1) and returns control to the *application processor*. In the *asynchronous part*, the *policy processor* periodically scans

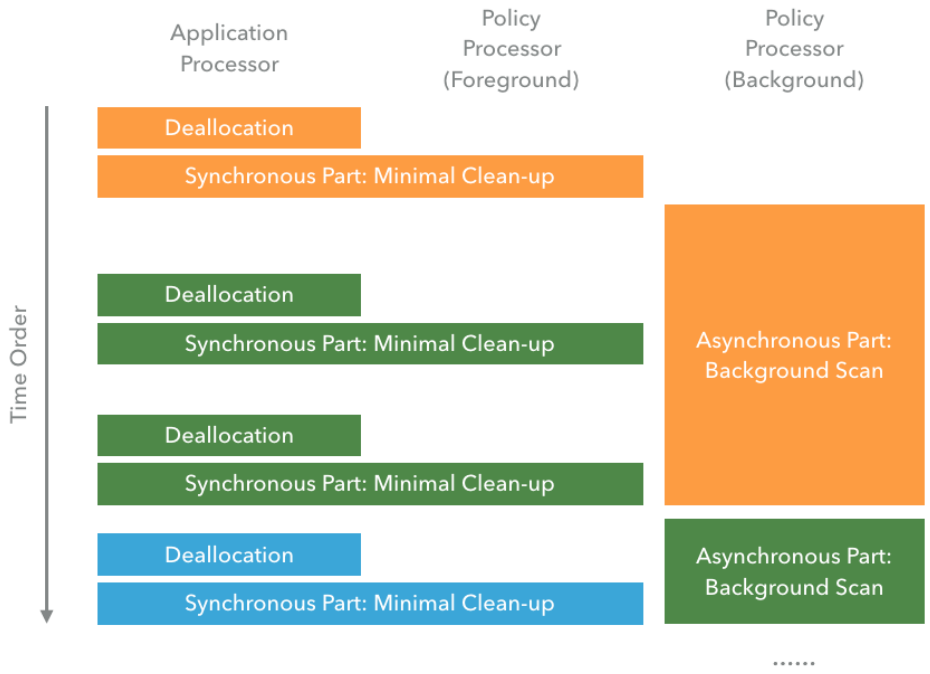


Figure 3-1: High-level timeline of the color reclamation scheme. Deallocation and clean-up blocks are color-coded by their corresponding background scan block. For example, heap colors released in the two green deallocations are purged and reclaimed during the green background scan.

the entire heap memory space in the background, zeroing out memory words with `PointerColor`s that have already been released in the *synchronous part*. After each complete scan, colors that have been purged from the heap memory space are added to a free list, which future allocations can reuse colors from. A high-level timeline illustrating the process and how the two parts interact is shown in fig. 3-1.

The color reclamation scheme maintains the following list of data structures:

1. `staging_list`: A list of colors that have been marked as freed by the memory allocator, but not yet picked up by the background

scanning process.

2. `scanning_list`: A list of colors that the background scanning process is currently purging. The background scanning process will remove any `PointerColor` tag it encounters with a color value present in the `scanning_list`.

3. `free_list`: A list of colors that have already been purged from memory space by the background scanning process, and are deemed safe to reuse for future allocations.

3.1 Synchronous Part: Immediate Cleanup

When the memory allocator deallocates a buffer, it notifies the *policy processor* of its intent to release the corresponding color. In order to enforce temporal memory safety, the *policy processor* needs to ensure that the dangling pointers that still refer to the released `CellColor` must not be used again following the release.

A naive implementation that ensures that dangling pointers are not used again would entail scanning through the entire heap memory space, and removing any `PointerColor` tag matching the released `CellColor`. However, such a scan imposes a long delay, during which the *application processor* cannot proceed with subsequent instructions. Such a long delay is unacceptable in practice, especially in real-time applications that have strict real-time deadlines (e.g. automotive and aerospace controllers).

Instead, to ensure a reasonable delay bound, the color reclamation scheme carries out only minimal immediate cleanup work following a

memory deallocations:

1. All registers are scanned. `PointerColor` tags matching the released color are removed.

2. The released color is added to `staging_list`.

3. A rule is added to the policy to prevent loading the released `PointerColor` in the future. The rule specifies that if a load instructions attempts to load from a memory word containing the released `PointerColor`, the result register of the load should exclude the `PointerColor` in its *metadata set*. These added rules will be referred to as *load-zeroing rules* in the remainder of this thesis.

4. A *background scan* is triggered, if none is currently in progress.

After these steps are finished, the *policy processor* returns control to the *application processor*. Since no full-memory scan is required in this *synchronous part*, application code is guaranteed a low, bounded latency upon memory deallocations.

3.2 Asynchronous Part: Background Scanning

At the start of each *background scan*, the colors currently in `staging_list` are moved into `scanning_list`, and `staging_list` is emptied.

During a *background scan*, the *policy processor* scans through the entire memory space, looking for any `PointerColor` tags matching colors that are present in `scanning_list`.

When a *background scan* finishes, current colors in `scanning_list` will be moved into `free_list`, and will be available for reuse in future allocations.

A pseudocode depiction of a *background scan* is given in fig. 3-2. The following pseudocode functions are used in fig. 3-2:

1. `get_pointer_color(addr)`: Returns the `PointerColor` of the memory word at address `addr`.

2. `get_metadata_set(addr)`: Returns the *metadata set* associated with the memory word at address `addr`.

3. `remove_pointer_color(metadata_set)`: Removes the `PointerColor` tag from `metadata_set` if present. Returns the modified *metadata set*.

4. `set_metadata_set(addr, new_metadata_set)`: Updates the *metadata set* associated with the memory word at `addr` to `new_metadata_set`.

During an ongoing *background scan*, newly released colors might start accumulating in `staging_list`. These recently released colors will not be purged within the current *background scan*, but will instead be picked up in the next *background scan*.

```
def scan():
    scanning_list = staging_list
    staging_list = {}                # empty list

    cursor = MEMORY_START
    while cursor != MEMORY_END:
        pointer_color = get_pointer_color(cursor)

        if (pointer_color is not None) and \
           (pointer_color in scanning_list):
            metadata_set = get_metadata_set(cursor)
            metadata_set = remove_pointer_color(metadata_set)
            set_metadata_set(cursor, metadata_set)

        cursor += 4                  # 4 bytes in a word

    free_list.add_all_from(scanning_list)
    scanning_list = {}
```

Figure 3-2: Pseudocode depicting a *background scan*.

Chapter 4

Implementation

In this chapter, we will first introduce the QEMU-based emulator system used in the development of PIPE tagged-architecture. We will then describe how we implemented a prototype of the aforementioned color reclamation system within this QEMU-based emulator architecture.

4.1 Tagged-architecture Emulator

In order to facilitate the development and testing of policies, the PIPE tagged-architecture project implements a software-based emulator system. The emulator system is based on QEMU [20], an open-source software emulator supporting a large variety of processor architectures. Currently, the PIPE tagged-architecture project is based on the open-source RISC-V instruction set architecture [21].

On top of QEMU's existing support for the RISC-V architecture, the PIPE project adds two additional components that collectively provide tagged-architecture support: a validator framework, and a policy

kernel. The validator framework implements a per-instruction hook into the QEMU emulator to provide an opportunity for policy evaluation, whereas the policy kernel implements the specific policy function that determines whether an instruction should be allowed to execute.

In order to communicate with the validator framework, each policy kernel implements a particular interface. The most important part of the interface consists of the following function:

```
int eval_policy(context_t *ctx, operands_t *ops,
                results_t *res);
```

The function `eval_policy` implements the policy function and determines whether a given instruction is allowed under the policy. It takes two input arguments:

1. `ctx`: This argument contains the execution context for the current instruction. Most importantly, its `epc` field contains the address of the current instruction being executed (i.e. the current program counter).

2. `ops`: The fields in this argument correspond to the policy function inputs described in section 2.2.2 and fig. 2-3.

The function `eval_policy` communicates its result in the following three ways:

1. `ctx`: `fail_msg` field of `ctx` is used to store an error message in case of policy violations. `cached` field of `ctx` is used to denote whether the returned policy result can be safely cached. If cacheable, subsequent instruction executions that have the same policy function inputs will bypass `eval_policy`, and instead use the previously computed result.

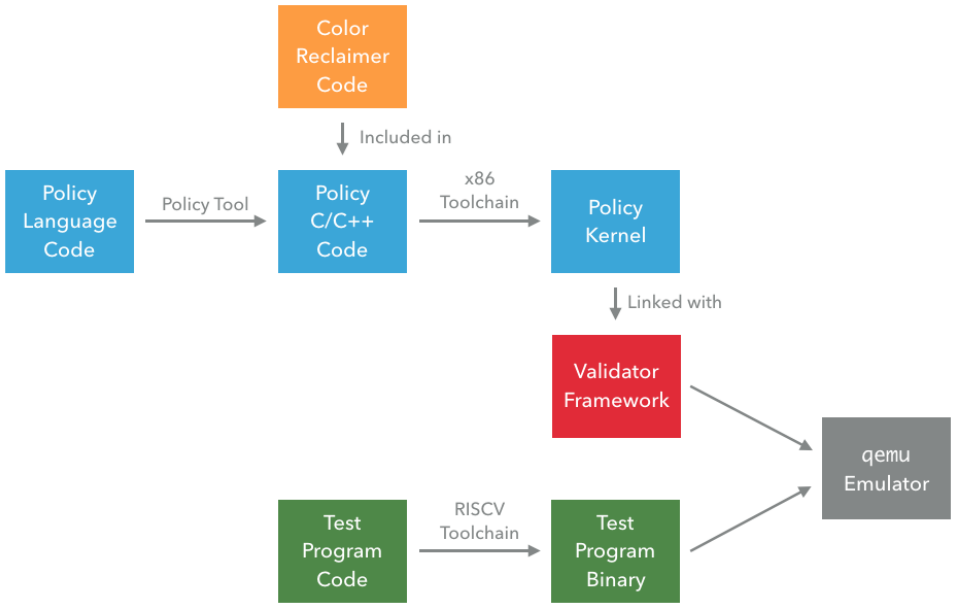


Figure 4-1: Architecture of the QEMU-based emulator.

2. `res`: As mentioned in section 2.2.2, if an instruction is allowed to execute, the policy function can modify relevant *metadata sets*. Such modifications are stored within fields of `res`, and communicated back to the validator framework.

3. Function return value: `eval_policy` returns an integer status code. Three options are available: `policySuccess` which denotes that the instruction is allowed, `policyExpFailure` which denotes that the instruction is explicitly denied by a matching *policy clause*, and `policyImpFailure` which denotes that the instruction is implicitly denied since no *policy clause* matches.

Despite the validator interface being defined and implemented in C/C++, policies are typically written in a special-purpose policy language (see section 2.2.3). Policy language code is translated by the

policy tool into a C/C++ implementation, which in turn gets compiled and linked with the validator framework.

On the other hand, the application processor running inside the emulator is compiled with a modified version of the clang/LLVM [22] toolchain. Notably, in addition to producing the application binary, the modified toolchain also applies a set of initial *metadata tag*s to specific parts of the program, as instructed by the policy code.

An overview diagram of the QEMU-based emulator architecture is shown in fig. 4-1.

4.2 Color Reclaimer Implementation

Normally, policy processor code is written in the special-purpose policy language, and then translated into its corresponding C/C++ implementation by the policy tool. However, many operations our color reclaimer design requires, including register scanning and background processing, are not supported by the policy language. Therefore, we implemented the color reclaimer prototype directly in C/C++. See fig. 4-1 for a diagram depicting how the color reclaimer code is integrated into the compilation process.

The color reclaimer provides the following interface for interacting with the rest of the policy code:

1. `bool reclaimer_process(context_t *ctx, operands_t *ops, results_t *res)`: This function shares the same arguments with the main policy function implementation `policy_eval` as described in section 4.1. `reclaimer_process` is called at the begin-

ning of `policy_eval` every time the policy function is invoked.

In particular, `reclaimer_process` detects and processes special memory stores that the application memory allocator uses to signal the intended releases of colors. Whenever the application memory allocator deallocates a pointer, it writes the released pointer to a special variable tagged with the `ToFreeColor` *metadata tag*. Upon detecting such stores with `mem == [+ToFreeColor]` (see section 2.2.3 for a description of this syntax), `reclaimer_process` handles immediate cleanup (see section 3.1) for the released colors.

`reclaimer_process` returns `true` if and only if such a special store is detected and processed. In this case, `policy_eval` returns early with a `policySuccess` result without having to execute the rest of the policy clauses.

2. `void reclaimer_notify_color_use(uint32_t color)`: This function is called by the heap coloring policy code every time it hands out a new color. It is used to maintain a heap color reference count in order to deal with color wrap-arounds. This part is not central to the color reclaimer design.

3. `void reclaimer_do_scan()`: This function is called once by the policy engine for the execution of each (application processor) instruction. In this function the color reclaimer carries out a small piece of its background scanning (see section 3.2) if one is in progress. Note that this function is unconditionally called for each instruction, whereas `policy_eval` (and consequently `reclaimer_process`)

is only called in case of policy rule cache misses.

4. `void reclaimer_process_post_eval(context_t *ctx, operands_t *ops, results_t *res)`: This function also shares the same arguments with the main policy function implementation `policy_eval`. It is called at the end of each `policy_eval` invocation. This function is used to enforce the *load-zeroing rules* (see section 3.1) by stripping `PointerColor` *metadata tag*s matching any color present in `staging_list` or `scanning_list`.

To summarize how the color reclaimer is integrated into the general policy validator framework (see section 4.1), fig. 4-2 lists the common scenarios and their corresponding code paths through the policy engine and color reclaimer code.

Scenario	Code Path

Buffer Deallocation	`policy_eval()`
	`  reclaimer_process() == true`
	`  // Returns early`
	`reclaimer_do_scan()`

Buffer Allocation	`policy_eval()`
	`  reclaimer_process() == false`
	`  // Process policy clauses`
	`  reclaimer_notify_color_use()`
	`  reclaimer_process_post_eval()`
	`reclaimer_do_scan()`

Regular Instruction - Cache Hit	`// policy_eval() not called here` `reclaimer_do_scan()`

Regular Instruction - Cache Miss	`policy_eval()`
	`  reclaimer_process() == false`
	`  // Process policy clauses`
	`  reclaimer_process_post_eval()`
	`reclaimer_do_scan()`

Figure 4-2: Common execution scenarios and their corresponding code paths through the policy engine and color reclaimer code.

Chapter 5

Evaluation

5.1 Setup and Metrics

Two types of benchmarks are used to evaluate the effectiveness and performance impact of the color reclaimer design: synthetic microbenchmarks (section 5.2), and allocation trace replays (section 5.3). Synthetic microbenchmarks simulate scenarios including both small-working-set and large-working-set workloads, whereas allocation trace replay aims to reproduce the memory allocation behavior of real programs.

The peak color count metric is used to evaluate the effectiveness of color reclamation. Peak color count is defined as the maximum number of unique colors in use at any given time (including colors inside `staging_list` and `scanning_list`). Practically, peak color count represents the minimal number of unique color values the tag system needs to provide in order to run the entire program without incurring color wrap-arounds (see section 2.3.2 for the discussion on the problems with color wrap-arounds). As such, effective color reclamation should corre-

spond to a low observed peak color count.

Since the QEMU-based emulator system cannot provide a cycle-accurate performance analysis, there is no way to directly measure the full performance impact of the color reclaimer. Instead, we focus on one performance factor: rule cache hit rate. As mentioned in the last paragraph, effective color reclamation can potentially reduce the color working set's size. A reduced color working set's size should alleviate the cache pressure on the rule cache, and should consequently lead to higher rule cache hit rate.

Each benchmark is tested under a variety of background scanning speed, including 0 (i.e. equivalent to with color reclamation turned off), 1/16, 1/8, 1/4, 1/2, and 1 words per (application processor) instruction. For cache hit rate tests, caches with size 16, 32, 64, 128, 256, and 512 are used.

5.2 Synthetic Microbenchmarks

Two microbenchmarks exercising the application memory allocator – `malloc_prof_1` and `malloc_prof_2` – are included in the PIPE policy test case suite. For each microbenchmark, we will provide a brief description of its setup, and present its results in both peak color count and rule cache hit rate.

Both microbenchmarks are tested with the default heap size 4KB.

5.2.1 `malloc_prof_1`

`malloc_prof_1` is a microbenchmark that represents a workload where memory buffers are allocated and then deallocated right after in quick succession. It represents a workload where the working set (in terms of number of buffers) is small at any given point of time. In particular, the microbenchmark:

1. For 256 times: allocates a 4-byte buffer, writes to it, reads from it, and then deallocates it.

2. For 128 times: allocates a 8-byte buffer, writes to it, reads from it, and then deallocates it.

3. Repeats the previous iterations with number of repetitions decreasing by half and buffer size doubling each time. Last iteration would have a repetition count of 2 and a buffer size of 512-bytes.

Peak color count results are shown in fig. 5-1. Without color reclamation (i.e. the blue line), the peak color count is 510, since the microbenchmark makes a total of 510 allocations and no color is ever reclaimed. As we turn on color reclamation (i.e. the orange line), peak color count starts dropping. And as we increase the background scanning speed, peak color count drops further. In this particular microbenchmark, since the working set is very small (always just 1 buffer in use at any given point of time), increasing the background scanning speed would reduce the scan duration, and therefore directly reduce the peak color count required to service allocations before the background scan finishes.

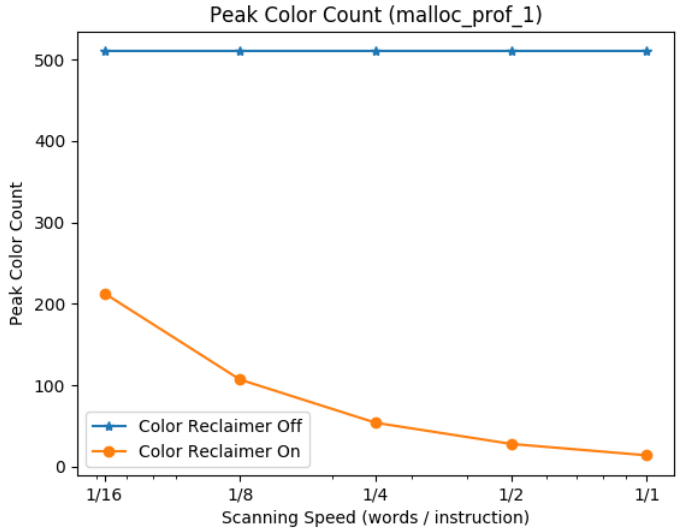


Figure 5-1: Peak color count results for the `malloc_prof_1` microbenchmark.

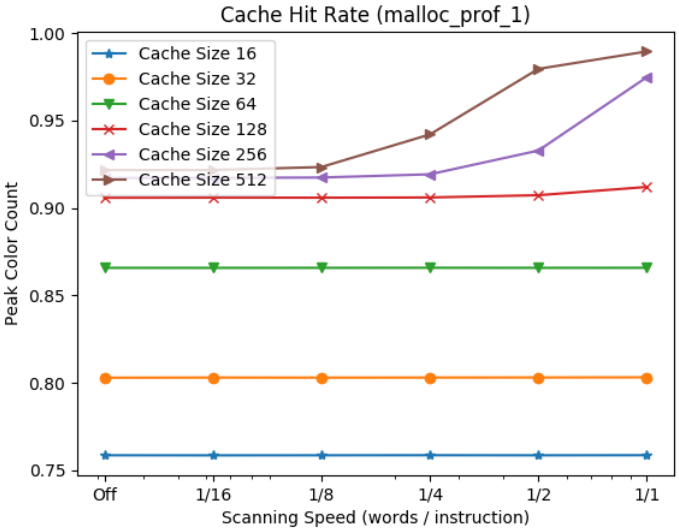


Figure 5-2: Cache hit rate results for the `malloc_prof_1` microbenchmark.

Cache hit rate results for various cache sizes are shown in fig. 5-2. As mentioned before, effective color reclaiming would translate into a reduced color working set size, and would consequently alleviate the cache pressure caused by heap coloring rules.

In fig. 5-2, for low cache sizes (64 or below), we observe no significant improvement as we turn on color reclaiming. In these cases, due to the small cache sizes, neither the no-reclamation working set size (i.e. 510) nor the with-reclamation working set size (i.e. 14 with the fastest tested scanning speed) is able to fit within the rule cache. Therefore, heap-coloring rules loaded into the rule cache are always evicted before they are used again by the next memory accesses with matching colors.

As the cache size increases to 128, we start seeing cache hit rate improvement where scanning speed is 1 word per instruction. In this case, the reduced color working set size enabled by color reclamation allows some heap-coloring rules to stay in the rule cache long enough to be used by subsequent memory accesses of the same colors, thereby contributing to the cache hit rate. This effect is also observed more prominently in larger cache size cases (i.e. 256 and 512), where the cache hit rate improvement starts at lower scanning speeds as well.

5.2.2 `malloc_prof_2`

`malloc_prof_2` is a microbenchmark that mimics workloads where allocations and deallocations happen in large batches, with long periods of memory allocator inactivity in between. In particular, `malloc_prof_2` consists of the following operations:

1. Allocate 150 buffers, each of a random multiple-of-4-byte size between 4 bytes and 32 bytes.

2. For 7500 times, pick a random buffer, and increment its first byte.

3. Deallocate the buffers.

4. Repeat the previous steps for a total of 3 times.

Contrary to `malloc_prof_1`, `malloc_prof_2` simulates workloads where allocations and deallocations happen in large batches, and where the working set size is considerably larger (i.e. 150 buffers vs. 1 buffer).

Peak color count results are shown in fig. 5-3. In this case, using different background scanning speeds yields the same peak color count: 150. On one hand, 150 is the theoretical lower bound for the peak color count since the program uses 150 buffers simultaneously. On the other, since in this case the batch memory allocations are preceded by batch memory deallocations, the background scanning process has an abundant amount of time to finish the scan. Consequently, the color reclamation implementation works equally well in this case regardless of the background scanning speed used.

Cache hit rate results for various cache sizes are shown in fig. 5-4. In all tested scenarios, we have very high cache hit rate ($> 99\%$). According to our measurements, data access instructions (45000 instructions) account for only 0.25% of all instructions (18167871 instructions) in this test case. Since the cache effect of data access instructions is totally over-shadowed by the non-data-access instructions, we did not observe a significant improvement in cache hit rate with color reclaimer turned on in this case.

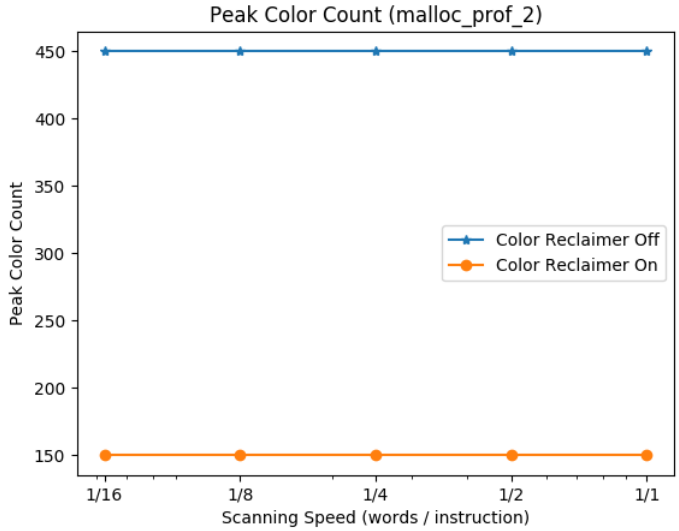


Figure 5-3: Peak color count results for the `malloc_prof_2` microbenchmark.

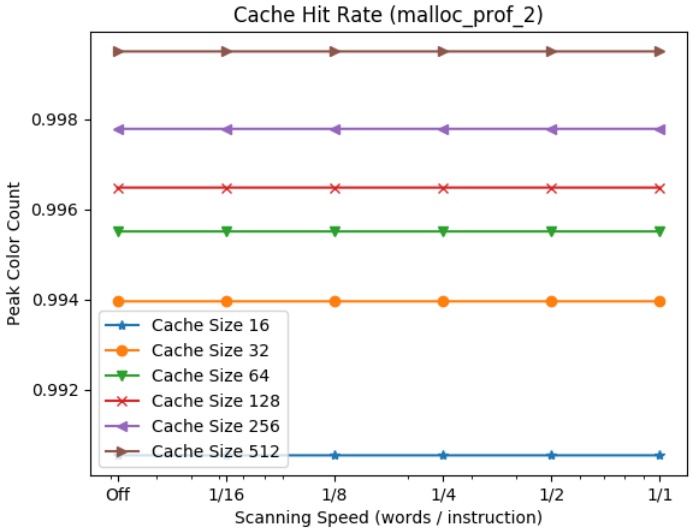


Figure 5-4: Cache hit rate results for the `malloc_prof_2` microbenchmark.

5.3 Allocation Trace Replays

Aside from synthetic microbenchmarks, we also employed allocation trace replays to test the color reclaimer's effectiveness in more realistic workloads. On a high level, with allocation trace replay we:

1. Run arbitrary program on a host platform (e.g. a x86 Linux platform).

2. Instrument and record the program's calls to heap memory management functions: `malloc`, `free`, `posix_memalign`, etc.

3. Write a replay program that repeats the recorded function calls with appropriately scaled timing.

4. Run the replay program in the QEMU-based simulator, and compare the peak color counts with color reclaimer turned off and on.

Using this scheme of allocation trace replay allows us to evaluate the effectiveness of our color reclaimer on a real-life workload, without having to port the actual program onto the QEMU-based emulator environment.

5.3.1 Replay Scheme Implementation

To collect a record of memory management function calls, we implemented a tracer library containing wrapper functions around `glibc`'s heap memory management functions, including `malloc`, `free`, `posix_memalign`, etc. We then used the `LD_PRELOAD` environment variable provided by

the Linux dynamic linker [23] to inject this tracer library into the program being recorded. The injected tracer library outputs the function call trace to a given file in the following format:

```
...
754360809760093 malloc(1024) = 0x5604bd4c9630
754360809773195 posix_memalign(16,4096) = 0,0x5604bd4f0a50
754360809785639 posix_memalign(16,4096) = 0,0x5604bd4f1a60
754360813412085 free(0x5604bd4f0a50)
754360813413051 free(0x5604bd4f1a60)
754360813414212 free(0x5604bd4c9630)
754360813465360 malloc(1024) = 0x5604bd4c9630
754360814143530 free(0x5604bd4c9630)
...
```

where the first number on each line is the timestamp of the function call (taken by reading the x86 processor cycle counter using the `rdtsc` instruction), the next part before = is the name and input arguments of the function call, and the part after = is the output values of the function call.

We then implemented a script that translates the memory management function calls we recorded into two basic operations: *allocation*s and *free*s. For example, the composite action of a `realloc` call will be broken down into an *allocation* followed by a *free*. Inside the script, we also convert the recorded `rdtsc` timestamps into relative timecodes. To do this, we first relativized the timestamps by subtracting the earliest observed timestamp from all timestamps. We then scaled all the timestamps such that 1 timecode unit corresponds to the time it takes to execute 1 `nop` instruction. The converted operations are written as C struct objects in the format below:

```
action_t TRACE_ACTIONS[] = {
  {0, ACTION_ALLOC, 0, 1024},
  {100, ACTION_ALLOC, 1, 1024},
  {200, ACTION_FREE, 1, 0},
  {300, ACTION_FREE, 0, 0}
};
```

In each struct, the first value is the converted timecode value, the second value is the operation type, the third value is the pointer index used to match each *allocation* to the corresponding *free*, and the final value is the buffer size for *allocation* operations and unused for *free* operations.

These struct definitions are directly `#included` into the replay program, which reads through the operations and simulates them under the given timing information.

5.3.2 `nginx` Allocation Traces

Since webserver is a common workload, we used allocation trace replay to simulate the memory footprint of running the `nginx` webserver [24]. Allocation traces are collected with the following shell command:

```
(LD_PRELOAD=/path/to/tracer.so \
 MALLOC_TRACER_OUTPUT_FILE_PREFIX=output \
nginx -p sandbox -c nginx.conf &) && \

for i in $(seq 1 $num_requests); do \
  curl localhost:8097; \
done && \

killall nginx
```

The `nginx.conf` configuration file used is shown below:

```
daemon            off;
worker_processes  1;

events {
    use           epoll;
    worker_connections  128;
}

error_log           logs/error.log info;
pid                 run/nginx.pid;
env                 MALLOC_TRACER_OUTPUT_FILE_PREFIX;
```

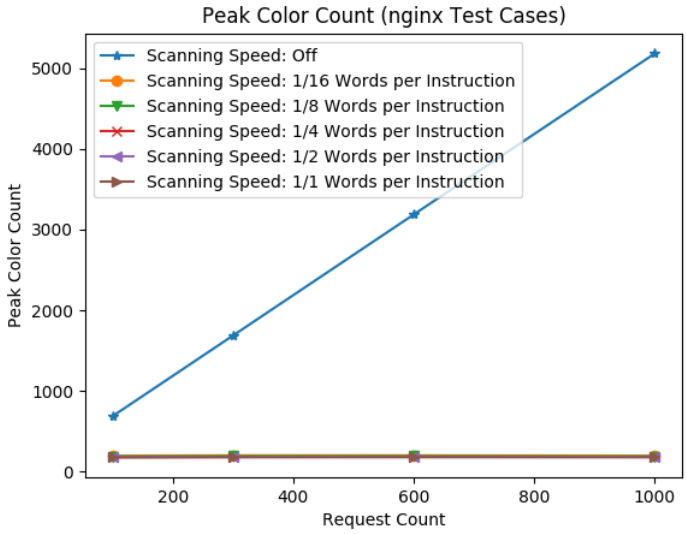
```
http {
    server_tokens off;
    charset       utf-8;

    access_log    logs/access.log  combined;

    server {
        server_name   localhost;
        listen        127.0.0.1:8097;

        error_page    500 502 503 504  /50x.html;

        location      / {
            root      html;
        }

    }
}
```

In order to test the color reclaimer's effectiveness in the case of a continuous stream of requests, we ran 4 batches of tests with 100, 300, 600, and 1000 continuous requests respectively. To accommodate the memory footprint of `nginx`, heap memory size is increased to 400KB.

5.3.3 Results

Peak color count results for the `nginx` allocation replay tests are shown in fig. 5-5. We see that without color reclamation, the peak color count increases linearly as the number of requests increases. With color reclamation turned on, the peak color count stays relatively unchanged as we increase the number of requests. In this case, the background scanning speed does not significantly affect the peak color count: even in the slowest case of scanning at 1/16 words per instruction, the color reclaimer is able to keep the peak color count at around 180.

This test case underlines the point that with color reclamation, peak color count is no longer tied to total number of allocations. In-



Speed	100 Requests	300 Requests	600 Requests	1000 Requests
Off	686	1686	3186	5181
1/16	198	202	202	198
1/8	190	193	193	190
1/4	180	183	185	183
1/2	180	180	180	180
1/1	178	180	180	180

Figure 5-5: Peak color count results for the `nginx` allocation replay tests.

stead, it's tied only to the working set size and the allocation frequency. This property is particularly important in long-running programs like the `nginx` web server, where the program stays alive for an indefinite amount of time, serving a continuous influx of requests.

Since our allocation traces include only allocations/frees but not memory accesses, we did not analyze the cache hit rate number for the `nginx` test cases.

Chapter 6

Related Works

The problem of color reclamation that this thesis tries to address draws a certain parallel with the well-studied problem of garbage collection. In particular, both problems involve a scanning phase through memory to identify live and stale objects. In this section, we will compare and contrast our color reclaimer design with various garbage collector designs on a variety of dimensions.

6.1 Scanning Order

An important feature of a garbage collection scheme is the order in which objects in memory are scanned and marked. Common classes of garbage collectors include *mark-sweep* collectors [25], *mark-compact* collectors [26], and *copying* collectors [27, 28]. In all three classes of garbage collectors, objects are scanned and marked by following references, starting from a particular set of root objects. Since an object and its pointees are not necessarily co-located near each other in mem-

ory, this visitation order can potentially result in poor memory access locality.

In contrast, the color reclamation scheme proposed in this thesis employs a linear scanning order. By going through the heap memory space in increasing order of memory address, the color reclaimer is able to utilize the processor's cache and memory prefetcher more effectively, and enjoy improved memory access locality.

6.2 Object Relocation

Garbage collectors operate in two ways: non-relocating garbage collectors reclaim unused space without moving live objects, whereas relocating garbage collectors can potentially move live objects in heap. Examples of relocating garbage collectors include *mark-compact* collectors [26] and *copying* collectors [27, 28]. By relocating live objects, these garbage collectors can potentially compact and defragment the heap memory space, in order to improve object access locality and future allocation availability.

The color reclaimer scheme proposed in this thesis is non-relocating, since it never moves any live object. In fact, the color reclaimer *cannot* move any live object due to the separation between data memory and metadata memory. In practice, an application-layer relocating garbage collector can be used in combination with the color reclaimer if low heap fragmentation is desired.

6.3 Generational Collection

Weak generational hypothesis states that most objects have a relatively short life-time [29]. Generational garbage collectors [30] use this distinction between objects' lifetime lengths to enable more efficient garbage collecting. In particular, generational garbage collectors pool objects with similar lifetime length together, and set different scanning frequencies for the pools. By only scanning short-living pools frequently, generational garbage collectors reduce the size of the memory space that needs to be scanned frequently, and consequently enables short-living objects to be reclaimed more timely.

The color reclamation scheme currently does not employ generational collection. It is possible to incorporate generational collection into the current color reclamation scheme to further improve its collection efficiency. Such an addition would likely require cooperation from the application processor's memory allocator in maintaining different-aged pools, and can be an avenue for future work.

Chapter 7

Conclusion

In this thesis, we proposed and implemented a color reclaimer design that aims to address the lack of color reuse in the current PIPE heap coloring scheme. By allowing colors to be safely reclaimed and reused, we reduce the number of metadata bits required to maintain color information and alleviate the risk of color wrap-around.

We then evaluated the reclaimer design on a combination of synthetic microbenchmarks and real-program allocation trace replays. Results show that the color reclaimer is effective in reducing the peak color count sustained during program lifetime. Moreover, in certain cases the reduced peak color count consequently brings cache hit rate improvement.

7.1 Future Work

Currently, the application memory allocator needs to traverse the entire memory buffer upon deallocations to zero out the `CellColor` tags.

If we take the idea of color reuse one step further, the memory allocator can presumably keep the memory buffer colored. After dangling `PointerColor`s are purged from registers and memory, the still-colored memory buffer can be directly handed out to service future allocations. This approach saves on application processor cycles used to uncolor and recolor memory buffers.

In the future we can also explore employing generational schemes in color reclamation. By segregating objects with different lifetime lengths into separate pools, the color reclaimer will be able to scan only the young-object pool regularly. This will further increase the reclamation efficiency by reducing scanning time.

Bibliography

[1] "CVE - Search Results." https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow, Jun. 2019. Accessed Jun. 18, 2019.

[2] "CVE-2019-7401." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7401, Feb. 2019. Accessed Jun. 18, 2019.

[3] "CVE-2019-12208." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12208, May 2019. Accessed Jun. 18, 2019.

[4] "CVE-2019-11417." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11417, Apr. 2019. Accessed Jun. 18, 2019.

[5] "CVE-2019-11560." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11560, Apr. 2019. Accessed Jun. 18, 2019.

[6] "CVE - Search Results." https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=use+after+free, Jun. 2019. Accessed Jun. 18, 2019.

[7] "CVE-2019-9020." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9020, Feb. 2019. Accessed Jun. 18, 2019.

[8] "CVE-2019-9003." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9003, Feb. 2019. Accessed Jun. 18, 2019.

[9] "CVE-2019-5096." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5096, Feb. 2018. Accessed Jun. 18, 2019.

[10] "CVE-2019-5098." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5098, Jan. 2018. Accessed Jun. 18, 2019.

[11] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Softw.*, vol. 19, pp. 42–51, Jan. 2002.

[12] W. Le and M. L. Soffa, "Marple: A demand-driven path-sensitive buffer overflow detector," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, (New York, NY, USA), pp. 272–282, ACM, 2008.

[13] Gimpel Software, "PC-Lint." https://www.gimpel.com/html/. Accessed Jun. 19, 2019.

[14] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, pp. 89–100, June 2007.

[15] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 28–28, USENIX Association, 2012.

[16] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, (Berkeley, CA, USA), pp. 51–66, USENIX Association, 2009.

[17] F. Gao, L. Wang, and X. Li, "Bovinspector: Automatic inspection and repair of buffer overflow vulnerabilities," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), pp. 786–791, ACM, 2016.

[18] A. Thomas and C. Casinghino, "PIPE: Hardware acceleration for efficient enforcement of software-defined security policies," in *GOMACTech '19*, March 2019.

[19] M. Payer, "Cs-527 software security – memory safety." https://nebelwelt.net/teaching/17-527-SoftSec/slides/02-memory_safety.pdf, 2017. Accessed Jun. 21, 2019.

[20] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46, 2005.

[21] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.

[22] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[23] "ld.so(8) - Linux manual page." http://man7.org/linux/man-pages/man8/ld.so.8.html. Accessed Aug. 3, 2019.

[24] I. Sysoev *et al.*, "Nginx," *Inc.,"nginx," https://www. nginx. com*, 2004.

[25] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Commun. ACM*, vol. 3, pp. 184–195, Apr. 1960.

[26] R. A. Saunders, "The LISP system for the Q–32 computer," pp. 220–231.

[27] R. R. Fenichel and J. C. Yochelson, "A Lisp garbage collector for virtual memory computer systems," *Communications ACM*, vol. 12, pp. 611–612, Nov. 1969.

[28] C. J. Cheney, "A non-recursive list compacting algorithm," *Communications ACM*, vol. 13, pp. 677–8, Nov. 1970.

[29] J. K. Foderaro and R. J. Fateman, "Characterization of VAX Macsyma," in *1981 ACM Symposium on Symbolic and Algebraic Computation*, (Berkeley, CA), pp. 14–19, 1981.

[30] D. M. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," in *ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 19(5), (Pittsburgh, PA), pp. 157–167, Apr. 1984.