

Introduction

In 1988, Robert T. Morris exploited a so-called buffer-overflow bug in *finger* (a daemon whose job it is to return information on local users) to mount a denial-of-service attack on hundreds of VAX and Sun-3 computers [159]. He created what is nowadays called a worm; that is, a crafted stream of bytes that, when sent to a computer over the network, utilises a buffer-overflow bug in the software of that computer to execute code encoded in the byte stream. In the case of a worm, this code will send the very same byte stream to other computers on the network, thereby creating an avalanche of network traffic that ultimately renders the network and all computers involved in replicating the worm inaccessible. Besides duplicating themselves, worms can alter data on the host that they are running on. The most famous example in recent years was the MSBlaster32 worm, which altered the configuration database on many Microsoft Windows machines, thereby forcing the computers to reboot incessantly. Although this worm was rather benign, it caused huge damage to businesses who were unable to use their IT infrastructure for hours or even days after the appearance of the worm. A more malicious worm is certainly conceivable [187] due to the fact that worms are executed as part of a daemon (also known as “service” on Windows machines) and thereby run at a privileged level, allowing access to any data stored on the remote computer. While the deletion of data presents a looming threat to valuable information, even more serious uses are espionage and theft, in particular because worms do not have to affect the running system and hence may be impossible to detect.

Worms also incur high hidden costs in that software has to be upgraded whenever an exploitable buffer-overflow bug appears. A lot of effort on the part of the programmer is spent in confining intrusions by singling out those software components that need to run at the highest privilege level, with the aim of executing the majority of the (potentially erroneous) code at a lower privilege level. While this tactic reduces the potential damage of an attack, it does not prevent it. A laudable goal is therefore to rid programs of buffer-overflow bugs, which is the aim of numerous tools specifically created for this task. So far, no tool has been able to ensure the absence of exploitable buffer

overflows without incurring either manual labour (program annotations) or performance losses (run-time checks). As a result, most security vulnerabilities today are still accredited to buffer-overflow errors in software [64, 126]. Interestingly, the US National Security Agency predicted a decade ago that buffer-overflow attacks would remain a problem for another ten years [173]. While many new projects part from C as the implementation language, most server software is legacy C code such that buffer overflows remain problematic. This book presents an analysis that has the potential to automatically detect all possible buffer overflows and thereby prove the absence of vulnerabilities if no overflow is found. This analysis is purely static; that is, it operates solely on the source code and neither modifies nor examines the program’s behaviour at runtime. Furthermore, it works in a “push-button” style in that no annotations in the program are required in order to use the tool. The challenge in the pursuit of this fully automated, purely static analysis is threefold:

soundness: It must not miss any potential buffer overflows.

efficiency: It has to deliver the result in a reasonable amount of time.

completeness: It should not warn about overflows if the program is correct.

The question of whether a buffer overflow is possible is at least as difficult as the Halting Problem and therefore undecidable in general. Due to the nature of this problem, an effective analysis must necessarily compromise with respect to completeness. The key idea of a static analysis is to abstract a potentially infinite number of runs of a program (which stem from a potentially infinite number of inputs) into a finite representation that is able to express the property to be proved. The technical explanation of worms in the next section introduces the “property to be proved”, namely that a program has no buffer overflows. The finite representation that we have chosen to express this property are sets of linear inequalities or, in their geometric interpretation, polyhedra. To motivate the choice of linear inequalities (rather than, say, finite automata as used in model checking [49]), we examine a small example program in Sect. 1.2. We then briefly comment on the three challenges of soundness, efficiency, and completeness of our analysis, a preview of the three parts that comprise this book. This chapter concludes with a comparison of related tools and a summary of our contributions.

1.1 Technical Background

In its simplest form, a program exploiting a buffer overflow manages to write beyond a fixed-sized memory region allocated on the stack. Consider, for example, a function that declares a local 2000-byte array `buffer` into which it copies parts of a byte stream that it receives from the network. The call

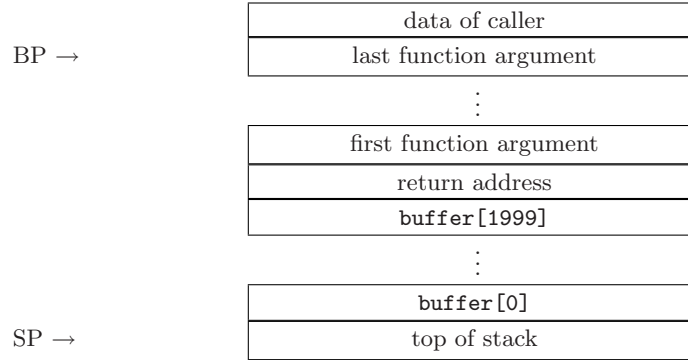


Fig. 1.1. A view of the stack after entering a function that declares a 2000-byte buffer. The pointers BP (base or frame pointer) and SP (stack pointer) manage the stack, which grows downwards (towards smaller addresses).

stack after invoking this function takes on a form that resembles the schematic representation in Fig. 1.1.

If a byte stream can be crafted such that more than 2000 bytes are copied to `buffer`, the memory beyond the end of the buffer will be overwritten, thereby altering the return address. A worm sets the return address to lie within `buffer` itself, with the effect that the byte stream from the network is run as a program when the function returns. It is the program encoded in the byte stream that determines the further action of the worm. A detailed description of how to craft one such input stream was given by a hacker known by the pseudonym of Aleph One, who presented a skeleton of a worm [141] that forms the basis of many known worms [159]. While the technical details are certainly interesting, the focus of this book lies in preventing such intrusions. Specifically, this work aims to prove the absence of buffer overflows, which is equivalent to showing that every memory access in a given program lies within a declared variable or dynamically allocated memory region. Detecting possible out-of-bounds accesses to variables is useful for any programming language with arrays (or plain memory buffers); however, only languages that do not check access bounds at run-time can create programs where buffer overflows create security vulnerabilities. The most prominent language in this category is C, a programming language that is widely used to implement networking software. Programmers chose C mostly for its ubiquity but also for the speed and flexibility that its low-level nature provides. However, it is exactly this low-level nature of C that makes program analysis challenging. Before Sect. 1.4 reviews the techniques to overcome the complexity of these low-level aspects, we detail what kinds of properties our analysis needs to extract from a program.

1.2 Value-Range Analysis

In order to prove the absence of run-time errors such as out-of-bounds array accesses, it is necessary to argue about the values that a variable may take on at a given program point. In the following, we shall call a static analysis that infers this information a value-range analysis. While this term was coined in the context of an interval analysis [95] we use a more liberal interpretation in that the inferred information may be more complex than a single interval. In this section we show how linear inequalities can be used to infer possible values of variables and that this approach can prove that all memory accesses lie within bounds. We illustrate this for the example C program in Fig. 1.2. The purpose of the program is to count the occurrences of each character in its first command-line argument. The idea is to define a table `dist`, where the i th entry stores the number of characters with the ASCII value i that have been observed in the input so far. Among the declared variables is the `dist` table containing 256 integers and a pointer to the input string `str`. In line 10, `str` is set to the beginning of the first command-line argument, namely `argv[1]`. This input string consists of a sequence of bytes that is terminated by a NUL character (a byte with the value zero). Note that the use of a NUL character to denote the length of the string is not enforced in C, even for arrays of bytes: The next line calls the function `memset`, which sets the bytes of a memory region to a given byte value, in this case zero. Here, the length of the buffer is passed explicitly as `sizeof(dist)` rather than being stored implicitly. The use of several conventions to store size information for memory regions is one of the idiosyncrasies of C that fosters incorrect memory management.

The `while` loop in lines 13–16 is the heart of the program. The loop iterates as long as the character currently pointed to by `str` is non-zero. Due to the `str++` statement in line 15, the loop will be executed for each character in the `argv[1]` buffer until the terminating zero character is encountered. The body of the loop increments the i th element of the `dist` array by one, assuming that the current character pointed to by `str` has the ASCII value i . Note that the character read by `*str` is converted to an integer, which ensures that the compiler does not emit a warning about automatic conversion from characters to an array index, which, according to the C standard [51], is of type `int`. The purpose of the last lines of the program is to print a fragment of the calculated character distribution to the screen.

Now consider the task of proving that all memory accesses are within bounds. While this task is trivial for variables such as `i` and `str`, expressing the correctness of the accesses to the memory regions `dist` and `*str` is complicated by the fact that the input string can be arbitrarily long.

In order to simplify the exposition, we assume that the program is run with exactly one command-line argument such that `argc` is equal to 2 and the return statement in line 9 is never executed. Under this assumption, the

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char* argv[]) {
5      int i;
6      char* str;
7      int dist[256];          /* Table of character counts.*/
8
9      if (argc!=2) return 1;    /* Expect one argument.*/
10     str = argv[1];           /* Let str point to input.*/
11     memset(dist, 0, sizeof(dist)); /* Clear table.*/
12
13     while (*str) {
14         dist[(int) *str]++;
15         str++;
16     };
17
18     for(i=32; i<128; i++) /* Show dist for printable */
19         printf("'%'c':%i\n", i, dist[i]); /* characters.*/
20
21     return 0;
22 }

```

Fig. 1.2. Example C program that calculates the distribution of characters.

correctness of all memory accesses can be deduced with a few linear equalities and inequalities:

- The content of `argv[1]` is a pointer to a memory region of variable size x_s . Since we cannot explicitly represent an arbitrary number of array elements, we merely track the first known zero element of this memory region as x_n (the so-called NUL position), which indicates the end of the string. A conservative assumption is that the buffer is no bigger than what is needed to store the first command-line argument and the NUL position. Hence, the relationship between the buffer size and the NUL position can be expressed as $x_n = x_s - 1$.
- Line 10 assigns the pointer to this memory region to `str`. C allows so-called pointer arithmetic in that the address stored in `str` can be modified as if it were an integer variable. In our example, line 15 increments `str` by one and hence introduces an offset x_o relative to the beginning of the buffer; that is, x_o denotes the difference between the pointers `str` and `argv[1]`.
- From the offset x_o and the null position x_n , we can check if the loop invariant holds. As long as $x_o < x_n$, the value of `*str` is non-zero and the loop is executed. As soon as $x_o = x_n$, the loop body is not entered again and the execution of the loop stops. If we can further infer that $x_o = x_n$

holds every time the loop stops, we have shown that the buffer pointed to by `argv[1]` is never accessed beyond its bound because all offsets $0, \dots, x_o$ during the execution of `*str` are no larger than x_s since $x_o \leq x_n = x_s - 1$.

- The values of characters read by `*str` are not known, except that they are non-zero with the exception of the last element. However, the value must be within the range of the C `char` type; that is, the index into the `dist` array, x_d , is restricted by $\text{CHAR_MIN} \leq x_d \leq \text{CHAR_MAX}$. The access to `dist` is within bounds if $0 \leq x_d \leq 255$ holds; that is, if $\text{CHAR_MIN} = 0$ and $\text{CHAR_MAX} = 255$.
- Finally, the correctness of the access `dist[i]` in line 19 can be ensured if the loop invariant $0 \leq x_i \leq 255$ can be guaranteed, where x_i represents the value of i within the loop body.

Note that the given chain of reasoning mainly relies only on linear inequalities that can be rewritten to $a_1x_1 + \dots + a_nx_n \leq c$, where $a_1, \dots, a_n, c \in \mathbb{Z}$, and x_1, \dots, x_n represent variables or properties of variables in the program. In particular, the state of a program can be described by a conjunction of inequalities; that is, a set of inequalities all of which hold at the given program point. Note that in this representation an equality such as $x = y + z$ can be represented as two inequalities, $x - y - z \leq 0 \wedge -x + y + z \leq 0$. Simple toy languages consisting of assignments of linear expressions can easily be abstracted into operations on inequalities [62]. The next section introduces some of the subtleties that arise in the analysis of real-world languages.

1.3 Analysing C

Implementing a static analysis that is faithful to the semantics of a real-world programming language requires that the semantics of the language be well (or even formally) defined. Giving a formal semantics to an evolving language that already has undergone several standardisations is a laborious task [143] and not very practical if C programs do not adhere to any (single) standard. Worse, even the latest C standard [51] leaves certain implementation aspects up to the compiler, such that the answer to the question of whether the program in Fig. 1.2 is correct with respect to memory accesses can only be “maybe”: On many platforms, including Linux on IA32 architectures and Mac OS X on PowerPC, the `char` type is signed, and hence $-128 \leq x_d \leq 127$, thereby violating the requirement that the index into `dist` lie within the interval $[0, 255]$. On platforms where `char` is unsigned, such as Linux on PowerPC, the program is correct. Next to implementation-specific semantics, C itself can be quite intricate. The seemingly plausible change of the statement `dist[(int) *str]++`; to `dist[(unsigned int) *str]++`; does not solve the problem: The so-called promotion rules of integers in C will first convert the value of `*str` to `int` (i.e., to a 32-bit value in $[-128, 127]$) and then to an unsigned integer (i.e., to $[2^{32} - 128, 2^{32} - 1] \cup [0, 127]$), leaving the program essentially unchanged.

Designing an analysis that interprets C programs in the same way as a particular mainstream compiler is a major undertaking in itself; see, e.g., [137]. Hence, rather than implementing a C front end for the analysis, we use the open source GNU C compiler as the front end and extract its intermediate representation. We convert this intermediate representation into Core C, a language amenable to our static analysis; Core C, defined in Chap. 2, contains mainly statements (rather than declarations) and attaches type information to operations (rather than to variables), thereby making many implementation-specific details explicit. Its formal semantics forms the basis of a sound abstraction to operations on inequalities, whose principles are explained in the next section.

1.4 Soundness

Given that a program may operate on a plethora of different inputs, it follows that an analysis that automatically proves every possible execution of the program correct must abstract from the actual program states, for instance, by summarising the possible valuations of variables at a given program point. Section 1.2 argued that the property of correct memory management can be expressed with a set of linear inequalities. Indeed, the idea of the analysis is to infer a set of inequalities that describes possible valuations of variables at a certain program point. Furthermore, since we are interested in verification, any such inequality set must be not only sound (correct) but precise enough to infer invariants that show that the program never exhibits a buffer overflow. Hence, the abstraction of sets of inequalities was chosen for its expressiveness. For the sake of this section, however, we will focus on soundness and leave the discussion of the achievable precision to Sect. 1.6.

1.4.1 An Abstraction of C

Simple program statements like `i=2*j+3` are readily translated into linear inequalities: With x_i and x_j representing the values of `i` and `j`, respectively, the assignment can be expressed as $x_i - 2x_j = 3$. However, analysing the full programming language C requires the translation of features such as arrays, pointer arithmetic, unions, etc., into a concise and, in particular, finite representation. To this end, several abstractions are needed. The following list summarises all abstractions applied within this work:

value abstraction: Summarising the possible values of a variable of each run into a finite representation such as an interval is the classic application of abstract interpretation [95]. With respect to the example, we observe that the value of the loop index `i` can be summarised to the interval [32, 127]. Several numeric domains, such as intervals, affine equations [109], and convex polyhedra [62], have been proposed to abstract concrete program

values. The analysis presented in this book uses the domain of convex polyhedra in addition to a simple domain of congruences [85]; that is, information on the multiplicity of variable values.

content abstraction: In C, the size of some memory regions is determined by the value of a variable at run-time. At any given program point, all runs of a program (and hence all variable-sized memory regions) must be described by a single abstract state. Since the abstract state is a polyhedron over a fixed, finite number of variables, it is not possible to map each concrete element of a memory region to one variable in the polyhedron. This may seem like a severe limitation, but the example program shows that the content of the `dist` array is irrelevant when proving correct memory management.

l-value abstraction: Each memory region in C has an address that can be inquired and passed around like any other value. These so-called pointers play a crucial role in C and motivated research into so-called points-to analyses [3, 46, 74, 99, 144, 176]. A points-to analysis treats addresses of variables purely symbolically since the actual addresses of variables can, in principle, differ between two program runs. The invariants inferred by a points-to analysis state which (symbolic) addresses may be found in a pointer variable at run-time.

region summary: Due to dynamic memory allocation, C programs can allocate an arbitrary number of distinct memory regions. These must be summarised into a finite set of memory regions to obtain a terminating and efficient analysis.

None of these abstractions are particularly new, although their combination has not been thoroughly explored. We briefly discuss the problems and our improvements of these abstractions, and their combination.

1.4.2 Combining Value and Content Abstraction

A static analysis usually summarises the possible values of variables, while other memory regions are ignored. Compilers, for instance, perform constant propagation and points-to analysis on simple variables – that is, variables that are not arrays or structures. In contrast to simple variables, worst-case values are usually assumed when accessing structures and arrays for variables whose address is taken or that are accessed with incompatible types. Venet and Brat showed how an interval analysis can be defined over so-called fields that are “added” to variables and C `structs` as part of the analysis [182]. The idea is that fields are only added if the access position is unequivocal; that is, if the array index or the pointer offset is constant. Consider the access `dist[i]` in line 19 of our example program. The index variable `i` is always accessed in its entirety and hence at the same offset 0. The initialisation in line 18 therefore adds a field containing the polyhedral variable x_i . In contrast, the variable `dist` is accessed at a variable offset that is calculated from the index `i`.

In this case, the write position is an interval $x_i \in [32, 127]$ (rather than a constant) and therefore no field is added. The approach of adding a new field only if the access offset is constant produces a finite number of fields and hence a finite number of variables in the polyhedron. In Chap. 5, we extend this approach to allow the same part of a memory region to be accessed with different types. These accesses are surprisingly common in C programs. For example, in Fig. 1.2, the call to `memset` accesses `dist` as a memory region of `char`, whereas line 14 accessed `dist` with its declared type `int`. Hence, treating differently typed accesses to the same memory region precisely is important and one novelty in this book.

This approach to finiteness simply ignores the content of memory regions that are accessed at different offsets, thereby resulting in an analysis that is too imprecise for many verification tasks. This problem can be tackled by inferring information about certain properties of a memory region rather than inferring the memory region’s actual content. We consider two possibilities:

element summary: Memory regions such as the `dist` array can be summarised by representing all array elements with a single abstract variable. In the case of the example, x_e might represent the values of all elements of `dist`. An analysis might infer that $x_e \in [0, 0]$ after zeroing the array at line 11. During each loop iteration, one element of the array is incremented while the remaining elements stay the same. This operation can be reflected on the abstract variable x_e by incrementing it weakly; that is, by setting x_e to an approximation of the previous value and the previous value plus one [80]. For the example program, $x_e \in [0, x_s]$ could be inferred; that is, each array element has a value between 0 and the size of the string.

meta information: Rather than inferring the values of (elements of) memory regions, it is possible to infer information relating to a certain property of a memory region. For instance, we explicitly state where the NUL character in the `argv[1]` buffer resides. The position of the NUL has been recognised to be the crucial information when analysing C string buffers [189].

In this work, we do not pursue the idea of summarising elements, mainly due to unresolved issues on constructing summary elements, if and how they can be split when overwriting them and hence how to limit the number of summary elements. In contrast, inferring information on the first zero position in a buffer requires a single polyhedral variable for each memory region and hence has no finiteness problems. Tracking NUL positions as part of a polyhedra-based analysis was presented in [71, 167], and the approach is further developed in Chap. 11.

1.4.3 Combining Pointer and Value-Range Analysis

In order to evaluate a read or a write access through a pointer variable, it is necessary to know what memory regions that pointer points to. Several different approaches can be taken to infer this information. During the last decade,

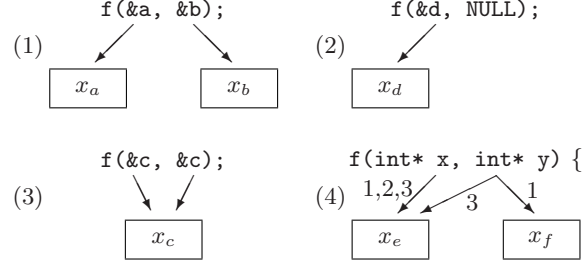


Fig. 1.3. Points-to information from different call sites.

tremendous advances have been made in the field of flow-insensitive points-to analysis [99,176] in which a set of all l-values (addresses of memory regions) is calculated that a given pointer variable may possibly contain during any execution of the program. The precision of points-to analysis can be substantially improved by performing a field-sensitive and/or a context-sensitive analysis.

A field-based analysis treats fields of a C **struct** as independent variables. A sound field-sensitive analysis must cater to pointer arithmetic commonly found in C programs; that is, a pointer might have a non-zero offset added to it before it is dereferenced, thereby accessing a different field from what its original l-value suggests. Chapter 5 shows how pointer arithmetic can be analysed by using the value-range analysis to calculate offsets relative to a base address, thereby giving precise offset information when pointers are dereferenced. In contrast, the most precise field-based points-to analyses distinguish between constant and non-constant offsets [175]. Tracking a points-to set using a points-to domain separately from the numeric offset that is tracking using a numeric domain is not always straightforward, and a formal description of how to combine both analyses is one contribution of this work.

For the sake of scalability, most points-to analyses are context-insensitive; that is, they combine points-to sets from different call sites when analysing a given function. While a context-insensitive approach scales well, it is not suitable for polyhedral analysis. Consider the example in Fig. 1.3, which shows the resulting points-to sets in drawing (4) for a function `f(int* x, int* y)` that was called as (1) `f(&a,&b)`, (2) `f(&d, NULL)`, and (3) `f(&c, &c)`. The pointed-to memory regions are shown as squares that each contain a single field represented by a polyhedral variable x_i that stores the value of the underlying integer. The first invocation seems to imply that the polyhedron at the callee should contain one variable for each parameter, here x_e and x_f in (4), to which the values of x_a and x_b from the caller are assigned. For the second invocation, however, the memory region containing x_f does not exist and x_e should be the only variable in the polyhedron. Hence the variable x_f represents no concrete memory region, which raises some difficult questions as

to what a linear relationship between, say, x_f and x_e means. Another problem occurs at the call site (3), where we chose to represent x_c by x_e but x_f would have been equally justifiable.

The problem of different calling contexts also arises in the context of performing a context-sensitive analysis that aims to reuse a previously analysed function. While polyhedra are, in principle, able to express linear relationships between input and output variables of a function that can be substituted at every call site, the C language itself seems to be a major obstacle to a context-sensitive analysis. For instance, Nystrom et al. [139] proposed a two-stage points-to analysis that is fully context-sensitive; that is, their analysis is as precise as inlining each function at all call sites. In a bottom-up pass, their analysis calculates summaries for each function, which are then inserted at each call site before a top-down pass calculates the points-to sets. Each summary describes all side effects that a function has on its local heap. However, for functions that are called with incompatible points-to sets, all statements that are relevant to l-value flow have to be copied to each call site, thereby defying the goal of context-sensitive analysis without inlining function bodies. This observation suggests that a fully context-sensitive analysis of C is likely to be impossible. In this work, we simply expand each function at each call site, which, in principle, incurs an exponential growth in the code size, but has been successfully applied in verification [31]. This choice also prohibits the analysis of recursive functions.

Finally, analysing dynamically allocated memory requires further techniques to ensure finiteness. Allocation sites that are only executed once should simply create a new memory region that can be read and written like declared variables in the program. In contrast, memory regions allocated within a loop must be summarised. We follow the classic approach in that memory regions that are allocated by a `malloc` statement at the same program point are summarised. By transforming the input program such that every function is expanded at its call site, this tactic is automatically refined such that memory regions allocated by a `malloc` statement in a given function are not summarised for different call sites of the function. In the upcoming analysis, functions are only inlined semantically, that is, they are re-analysed for every new call site such that care has to be taken to achieve the same semantics for dynamically allocated memory regions.

This concludes the overview of what we choose to extract from a C program. The details of these abstractions form Part I of this book. We now embark on the question of how to automatically approximate the state space of a C program.

1.5 Efficiency

Any useful program-analysis tool has to be efficient in order to be of practical help to the programmer. Interestingly, an efficient analysis can be implemented

on top of semi-decision procedures such as theorem proving by using timeouts [152]. Theorem proving is an attractive approach due to its ability to describe properties over a potentially infinite state space such as the value of a variable or the shape of a heap. However, the ability to create arbitrarily sized descriptions can affect termination of automated proving strategies, hence the use of timeouts. In contrast, classic model checking operates on finite automata (that is, a finite state space) and therefore always terminates [49]. In practice, however, it is difficult to soundly map the state of a program to a finite automaton of acceptable size. Thus, model checking is often impractical in that the size of the finite automaton grows too rapidly with respect to the input program to permit the analysis of larger systems [48]. Rather than using a finite state space, our analysis uses a convex polyhedron to describe a potentially infinite state space, which necessarily implies that some descriptions are approximations to the actual state space. On the positive side, our analysis can be terminating, as the inferred polyhedra are always finite. In this book, we use the framework of abstract interpretation by Cousot and Cousot [56] to describe this approximating analysis. We briefly illustrate the idea of a static analysis based on abstract interpretation before discussing the challenge of implementing such an analysis efficiently.

Consider the **for** loop in lines 18–19 of the running example in Fig. 1.2, whose control-flow graph is depicted in the upper half of Fig. 1.4. The edges of the control-flow graph are decorated with the polyhedra P, Q, R, S , and T , which denote the state at that given program point and which we write as sets of inequalities. In order to illustrate how these polyhedra are incrementally inferred, we write P_j to indicate the j th update of the state P . As before, let x_i denote the value of the program variable i . After executing the initialisation statement `i=32`, the initial state of P is given by $P_0 = \{x_i = 32\}$. This state is propagated to $Q_0 = P_0$, where the test `i<128` partitions this state into $S_0 = \{x_i = 32, x_i \leq 127\}$ and $R_0 = \{x_i = 32, x_i \geq 128\}$. Note here that $x_i < 128$ is tightened to $x_i \leq 127$ since all program variables are integral. With respect to the sets of points described by these states, S_0 is equivalent to Q_0 and R_0 is unsatisfiable; that is, the set of points described by R_0 is empty. An unsatisfiable polyhedron implies that the corresponding point in the program is unreachable; here, the state of R_0 implies that the loop will not terminate without iterating at least once. The analysis continues by propagating the satisfiable state S_0 . Since the value of x_i in S_0 is 32 and therefore between 0 and 255, the array access `dist[i]` is within bounds. Incrementing the loop counter yields a new state $T_0 = \{x_i = 33\}$, which is propagated back to the beginning of the loop to where the control-flow paths merge. It is at this merge point that the two state spaces P_0 and T_0 are joined to form $Q_1 = P_0 \sqcup T_0 = \{32 \leq x_i \leq 33\}$, where the join operator \sqcup calculates a polyhedron that includes its two arguments. Since the maximum value of x_i is still below 128, another iteration of the loop is calculated, yielding $T_1 = \{33 \leq x_i \leq 34\}$ after the instruction `i++`. This state in turn can be joined to form $Q_2 = P_0 \sqcup T_1 = \{32 \leq x_i \leq 34\}$. Depending on the loop bounds, the

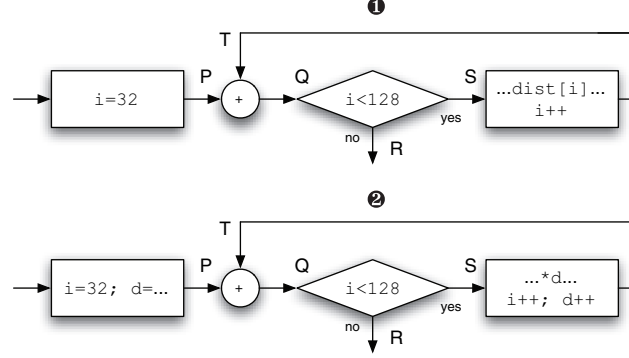


Fig. 1.4. The control-flow graph of the original **for** loop and the modified variant.

analysis may perform an excessive number of iterations, which is unacceptable for an efficient analysis. In order to avoid this, widening can be applied, which accelerates the fixpoint calculation [59]. The principal idea of widening is to compare two state spaces that result from two consecutive iterations and remove those inequalities that are not stable. In the example, we calculate the widened polyhedron $Q'_2 = Q_1 \nabla Q_2$; that is, we remove all inequalities from Q_1 that do not exist in Q_2 . The result is $Q'_2 = \{32 \leq x_i\}$. Enforcing the condition `i<128` yields $S_2 = \{32 \leq x_i \leq 127\}$ for the loop body and $R_2 = \{32 \leq x_i \leq 128\}$, which is equivalent to $\{x_i \geq 128\}$ as state space when the loop exits. Analysing the loop body with S_2 will infer that $x_i \in [32, 127]$ and hence that the index `i` lies within the bounds of the array. Furthermore, after the evaluation of `i++`, the new state space $T_2 = \{33 \leq x_i \leq 128\}$ arises and hence $Q_3 = P_0 \sqcup T_2 = \{32 \leq x_i \leq 128\}$. Intersecting this state with the loop invariant $x_i \leq 127$ yields $S_3 = \{32 \leq x_i \leq 127\}$, which is equivalent to S_2 , and hence a fixpoint has been reached. It can be shown that the inferred state includes all possible values that the variable i can take on in the program.

While the calculation above of the loop invariant $x_i \in [32, 127]$ demonstrates the basic technique of inferring a fixpoint of a loop, the real strength of polyhedra lies in the ability to infer relationships between different variables. In order to illustrate this ability, consider the following modified **for** loop that is functionally equivalent to the one in Fig. 1.2:

```
int* d=&dist; d+=32;
for (i=32; i<128; i++, d++)
    printf("%c'␣:␣%i\n", i, *d);
```

Instead of recalculating the array index, the access position is calculated incrementally by advancing the pointer `d` by one element in each loop iteration. The corresponding control flow is shown in the second graph of Fig. 1.4.

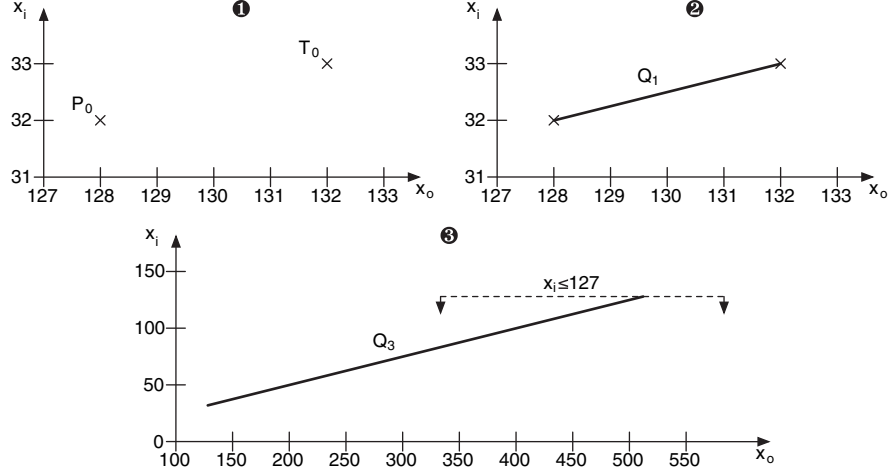


Fig. 1.5. Inferring the state space within the loop using polyhedral analysis.

Let x_o denote the byte offset of pointer **d** relative to the beginning of **dist**. Assuming that each **int** element of the array requires four bytes of storage, the statement **d+=32** increments x_o from 0 to 128 and the abstract state with which the loop is entered is given by $P_0 = \{x_i = 32, x_o = 128\}$. After evaluating the test **i<128**, x_i is incremented by one, while x_o is incremented by the size of one element of **dist**, namely 4 bytes. Thus, the state after executing the loop body once is $T_0 = \{x_i = 33, x_o = 132\}$.

While the join $P_0 \sqcup T_0$ can be described by $\{32 \leq x_i \leq 33, 128 \leq x_o \leq 132\}$, a more precise set of inequalities exists that includes P_0 and T_0 . Consider the geometric interpretation of the state space in the first graph of Fig. 1.5. A more precise (but still concise) characterisation of the state space is given by the convex hull of the two points; that is, the smallest closed, convex space that includes P_0 and T_0 . In our example, a set of inequalities that includes the convex hull of P_0 and T_0 is $Q_1 = \{32 \leq x_i \leq 33, x_o = 4x_i\}$, as depicted in the second graph of the figure.

As with the example before, we evaluate another loop iteration in which $S_1 = Q_1$ and where $T_1 = \{33 \leq x_i \leq 34, x_o = 4x_i\}$. The join $P_0 \sqcup T_2$ is again the convex hull of the two polyhedra, yielding $Q_2 = \{32 \leq x_i \leq 34, x_o = 4x_i\}$, which differs from Q_1 only in the upper bound on x_i . Applying widening yields the infinite state $Q'_2 = Q_1 \nabla Q_2 = \{32 \leq x_i, x_o = 4x_i\}$. The loop invariant ensures that the next iteration yields $Q_3 = Q'_2 \cup \{x_i \leq 127\}$, as shown in the third graph of Fig. 1.5.

The polyhedron $Q_3 = \{32 \leq x_i \leq 127, x_o = 4x_i\}$ describes an invariant that is sufficient to show that the pointer **d** has an offset between 32 and 512 and hence lies within the **dist** array, which contains 1024 bytes. Note that

this invariant required reasoning about the relationship between x_i and x_o : Without this relational information, widening would have resulted in $\{32 \leq x_i, 128 \leq x_o\}$, which, by adding the loop invariant $i < 128$, would give the less precise loop invariant $\{32 \leq x_i \leq 127, 128 \leq x_o\}$, which leaves x_o unrestricted.

We chose to infer relational information, as previous work based solely on intervals yielded results that were too imprecise for the verification of string buffer operations [184]. One drawback of using convex polyhedra as the basis for a static analysis is their inherent complexity. Specifically, calculating the convex hull of two polyhedra is an exponential operation [24]. This book therefore presents a novel sub-class of general polyhedra that is based on the idea of decomposing polyhedra into sets of planar polyhedra. To this end, Chap. 7 introduces efficient algorithms for planar polyhedra; in particular, we present a novel convex hull algorithm for planar polyhedra. By building on these planar algorithms, Chap. 8 presents the Two-Variables-Per-Inequality (TVPI) domain, which provides an efficient way of manipulating polyhedra in which each inequality has at most two variables. The following chapter presents techniques to refine polyhedra around the contained set of integral points, a process that is required to ensure that coefficients of inequalities do not grow indefinitely. Such a guarantee cannot currently be given for general polyhedra. As such, the TVPI domain presents, to our knowledge, the most precise polyhedral domain with a performance guarantee.

Given an abstraction from C and an efficient domain to calculate an over-approximation of its state space, we proceed to detail improvements in the precision of the analysis.

1.6 Completeness

For the sake of staying focussed on relevant aspects of finding buffer overflows, we chose to test and refine our analysis against a program called `qmail-smtp`, which is part of a mail transfer agent (MTA) whose task it is to forward email traffic. As this program parses incoming emails from the network, it is susceptible to buffer-overflow attacks and therefore a prime candidate for inspection. It is also simple enough in that it is single-threaded, does not make use of recursive functions, and uses few library functions.

The verification of real-world programs opens up many challenges, some of which are not clear until the analysis is run the first time on the chosen input program. While the aspects of soundness and efficiency need to be addressed before an analysis is implemented, the precision of an analysis (or the lack thereof) often manifests itself when the analysis is run. When precision is unduly lost, the analyser emits warnings that do not correspond to actual mistakes in the program. These so-called false positives then motivate a refinement of the analysis. Note that the way the C program is abstracted and the choice of the polyhedral domain both significantly affect the ability to infer precise results. However, this section presents three aspects of our analysis

that are solely dedicated to improving the precision. These aspects are the ability to argue about NUL positions in string buffers, an improved widening strategy, and a refinement of the points-to analysis using Boolean flags in the polyhedral domain. We discuss each aspect in turn.

1.6.1 Analysing String Buffers

A basic idiom in the context of string buffer operations is to iterate over the contents of a string until the NUL position, which terminates the string, is reached. An example of this operation is given in lines 13–16 in Fig. 1.2. Here, the buffer `argv[1]` denotes the first command-line argument, which, if it exists, contains a NUL-terminated string of arbitrary size. Due to the unknown size, it is not possible to model individual elements of this buffer with polyhedral variables. Instead, the buffer `argv[1]` can be treated as a dynamically allocated memory region whose size is given by the polyhedral variable x_s . Furthermore, it is known that a NUL character exists that terminates the input argument. Without loss of generality, we can assume that this NUL character resides in the last element of the buffer. Suppose that the polyhedral variable x_n denotes this NUL position; then $x_n = x_s - 1$. As a third parameter, let x_o denote the offset of the pointer `str` relative to the address `argv[1]`. To the programmer, it is rather obvious that the pointer `str` is incremented in line 15 until the NUL position has been reached. To the analysis, this information is only indirectly available since the test `*str` in line 13 does not query the NUL position x_s but merely accesses the buffer. Let x_c denote the character returned by `*str`. The idea of a string buffer analysis is to encode the NUL position by refining x_c to $[1, 255]$ if the access position x_o is in front of the NUL position x_n and to refine x_c to 0 if $x_o = x_n$. By using the ability of polyhedra to express linear relationships between variables, we relate x_c , x_o , and x_n such that testing the loop condition `*str` results in refining x_n and x_o . In particular, testing if `*str` is true corresponds to adding $x_c > 0$ to the state, which recovers the information that $x_o < x_n$. Similarly, testing if `*str` is false corresponds to adding $x_c = 0$ to the state, which recovers the information that $x_o = x_n$. In fact, this test partitions the state space (since x_c is positive) and thereby infers that $x_o = x_n$ on loop exit and that $x_o < x_s$ within the loop, which proves that `argv[1]` is never accessed out-of-bounds. The details of this analysis are given in Chap. 11, which presents the string buffer analysis as a refinement of the basic analysis of C that is described in the first part of the book.

1.6.2 Widening with Landmarks

The key to an efficient polyhedral analysis is to accelerate the fixpoint calculation to overcome slowly growing coefficients in inequalities. This process is known as widening [59] and was already applied in Sect. 1.5. The principal idea of widening is to remove inequalities that have changed between

two consecutive iterations. The full removal of inequalities, however, incurs a substantial precision loss, as witnessed by the R_i states from the last section that describe the state space at the end of the **for**-loop in lines 18–19. While the actual value of the loop index i on exit of the loop is 128, applying widening can only infer that $x_i \geq 128$. While an operation called narrowing [59] can be applied to refine this state again, we pursue a different strategy in which widening is modified in that changing inequalities are not removed but merely relaxed. The amount by which inequalities are relaxed is inferred by observing conditionals (so-called landmarks) in the analysis. For instance, the test $x_i \leq 127$, which stems from the loop condition $i < 128$ in line 18, conveys the information that the upper bound of x_i in $Q_1 = \{32 \leq x_i \leq 33\}$ and $Q_2 = \{32 \leq x_i \leq 34\}$ should be increased by another 94 units to yield $Q'_2 = \{32 \leq x_i \leq 128\}$. Using this state instead of the fully widened state $Q'_2 = \{32 \leq x_i\}$ enables the analysis to infer that $R_2 = \{x_i = 128\}$ rather than $R_2 = \{x_i \geq 128\}$. Widening with landmarks is presented in Chap. 12, where it is shown to be crucial for analysing string buffers in a precise way.

1.6.3 Refining Points-to Analysis

Using a standard points-to analysis is often too imprecise when it comes to the verification of programs. Next to field sensitivity and context sensitivity, a points-to analysis can be categorised with respect to its flow sensitivity. A flow-insensitive analysis infers a single points-to set for each program variable that is valid for the whole program. In contrast, a flow-sensitive points-to analysis infers a points-to set at each program location. Only the latter analysis can therefore determine that a statement such as **if** ($p \neq \text{NULL}$) $*p = 42$; does not dereference a **NULL** pointer. While the analysis as presented in the first part performs a flow-sensitive analysis, Chap. 13 details a refinement of this flow-sensitive analysis where the content of points-to sets can be related with the numeric values of program variables, thereby substantially improving the precision of standard flow-sensitive points-to analysis.

1.6.4 Further Refinements

The refinements presented allow our analysis to verify non-trivial examples. Unfortunately, even with the techniques described so far, the program in Fig. 1.2 still evades verification. One problem is the argument **argv** to **main**, which constitutes an array of pointers. In order to express that this array contains an arbitrary number of pointers to **NUL**-terminated strings, it requires the ability to state that all pointers in the **argv** array have an offset of zero, which is beyond the abstraction techniques of our analysis. However, it is possible to analyse the memory management of the example precisely if a string constant is assigned to **str** in line 10. Interestingly, the verification of the example evades the current implementation of our analysis when **argv** is fixed to contain a single pointer to a **NUL**-terminated buffer of *arbitrary* size

as described in Sect. 1.6.1. Chapter 14 details this and other shortcomings and suggests efficiency and precision improvements in order to make the analysis applicable to real-world C programs.

We conclude the introduction with an overview of related tools and a summary of our contributions to the field of soundly analysing C programs.

1.7 Related Tools

In this section, we present other tools to analyse C or C++ programs, which can generally be partitioned into sound analyses and unsound analyses. While both approaches create false positives, the unsound analyses may also miss errors and thus cannot prove program correctness. We focus mostly on sound analyses, as their techniques are more relevant to the analysis presented in this book.

1.7.1 The Astrée Analyser

The Astrée analyser [30,31,60] is a value-range analysis in the sense of Sect. 1.2 with the aim of proving the absence of run-time errors; that is, range overflows, division by zero, out-of-bounds array accesses, and other memory management errors. The analysis is sound and precise even for floating-point calculations. However, it is restricted to embedded systems in that dynamic memory allocation is only allowed at start-up (no heap-allocated data structures) and recursion is only allowed if it is bounded, as all functions are semantically inlined. The target of the analysis is flight-control software for Airbus A340 and A380 aeroplanes, which features, for instance, second-order digital filters that are repeatedly evaluated. The analysis is able to prove the absence of run-time errors on programs as large as 400,000 lines of code in less than 12 hours using 2.2 GB of memory. In order to estimate the valuations of floating-point variables that arise in the digital filter code, special domains such as the Ellipsoid domain were defined [31]. Linear relationships are inferred using the Octagon domain [130], which can express relationships of the form $\pm x \pm y \leq c$, where c can either be an integer or a floating-point number. Since the largest programs analysed contain about 80,000 global variables, the variables on which relational information is required are divided into packs – that is, sets that potentially overlap. Relational information is only inferred within each pack. This is important for scalability since the Octagon domain, like our TVPI domain, stores information for each pair of variables and thus has a memory footprint that is necessarily quadratic in the number of variables. In order to improve upon the precision of the relatively weak Octagon domain, symbolic propagation of variable values is used [131]. The ability to infer that no floating-point overflow occurs is based on the assumption that the number of iterations of the outermost control loop is bounded by a constant, namely $6 \cdot 10^6$. In order to achieve the required precision and scalability, the analyser

is able to partition the set of traces at a particular program point; that is, it is possible to track separate states for different values of a variable and to keep states separate where control-flow edges join [123]. The latter can be used to unroll loops, which is crucial for the kind of embedded code the analysis targets, where loops often initialise variables in the first loop iteration. Arrays that are of static size are either fully unfolded (each element is represented in the analyser) or smashed (all elements are represented by one abstract element). The effect of integer calculations that exceed the range of the target variable is either reported as erroneous or the wrapping that occurs in the concrete program is made explicit, depending on the specification of the user. Finding fixpoints of loops is guided by widening thresholds, which are values that indicate likely bounds on variables. All parameters regarding trace partitioning, array handling, and widening thresholds can be communicated to the analyser by using annotations in the source code. From the experience of finding these parameters, heuristics have been devised that work well for programs that have similar structures, thereby reducing the burden on the user. The implementation of the Astrée analyser uses a configurable hierarchy of modules that implement trace partitioning, track memory layout, etc., and the various numeric domains [61]. This design facilitates the addition of new domains and thereby the adaptation to new classes of programs.

1.7.2 SLAM and ESPX

The unsound PREFIX tool [38] was one of the first tools deployed by Microsoft in order to uncover faults in device drivers. The tool was eventually replaced by SLAM [18], which was created at Microsoft Research and is now integrated into the development process of Microsoft Windows. It is commercially available as part of Microsoft Developer Studio, where it serves to reveal programming errors related to locking, handling of files, memory allocation, and other temporal properties [20]. Using a simple specification language called SLIC to describe the effects of certain function calls, a tool called C2BP translates the input C program into a Boolean program. This Boolean program is then checked using the BEBOP model checker [19]. Unless the model checker can verify the properties that were described using the SLIC specification, another tool, called NEWTON, is run in order to refine the Boolean program using the counterexample from BEBOP. The idea is that repeated refinement of the abstraction will eventually create a Boolean program that is precise enough to prove the program correct. The abstraction done in C2BP is meant to be sound but incorrectly handles memory accesses through pointers when they denote overlapping memory regions. Furthermore, it is incorrectly assumed that integer variables cannot wrap, although this issue is being addressed, as pointed out in the talk of [52].

With respect to buffer overflows, Microsoft uses an approach similar to SLAM in that a new specification language, SAL, was defined, with which the programmer is able to annotate C programs. In order to aid the

programmer in annotating source code, an unsound tool called SALinfer can provide likely invariants, which the programmer can adapt. An inter-procedural checker called ESPX checks the annotations and is sound if all buffer operations are correctly annotated. To date, Microsoft developers have inserted 400,000 annotations into the next version of Windows, of which 150,000 were inferred automatically [90]. This labour-intensive approach led to the detection of 3,000 potential buffer overflows, which means that approximately 100 annotations are needed to detect a single buffer overflow.

1.7.3 CCured

CCured is a pragmatic approach that combines static analysis with run-time checks. The input C program is parsed using a front end written in O’Caml that exhibits either the semantics of the GNU C compiler or that of Microsoft’s Visual C compiler and translates to the so-called C Intermediate Language (CIL) [137]. The intermediate representation is then analysed using dependent types with the intent of proving most memory operations correct [136]. Any pointer whose properties cannot be statically guaranteed is converted into a fat pointer that includes the beginning and end of the buffer that the pointer points to. The code is then transformed by adding run-time checks to all memory accesses that the static analysis cannot guarantee to be safe. In order to avoid dangling pointers caused by freeing dynamically allocated memory regions too early, all calls to `free` are removed and a garbage collector [32] is used. The resulting program is then translated back to C and compiled using a normal C compiler. Recent work has addressed the task of verifying the binary output with the invariants generated during the static analysis [94].

1.7.4 Other Approaches

A vast number of tools have been proposed that use heuristics to highlight locations in the C source code that are likely to be erroneous. By using heuristics, these tools are simpler than sound analysers but may miss faults. An important aspect of all unsound approaches is that their precisions are difficult to compare. For sound approaches, it is sufficient to compare the number of false positives. Unsound approaches, however, can to a certain degree trade off the number of false positives with the number of missed bugs. Since the number of missed bugs is not known, the comparison of the number of false positives is mostly meaningless.

LClint [75] and its variants [115, 116] use lightweight annotations that are added to the C programs to find buffer overflows and other faults. In contrast, Wagner proposed a fully automatic buffer-overflow analysis based on intervals [184], which, however, is not very precise. Dor et al. were the first to analyse pointer accesses to string buffers using polyhedra [71]. However, their work turned out to be unsound [167], which triggered their work on soundly analysing C string functions aided by user annotations [72]. Ghosh et al.

use fault injection to find buffer overflows; that is, a tool repeatedly creates strings with the aim of overflowing a specified buffer in the stack. The process is guided by inspecting the dynamic run-time behaviour of the program [77]. Haugh and Bishop claim that automatically verifying the absence of buffer overflows is impossible. Their STOBO tool instruments a program to observe program behaviour when run on normal input data. The inferred data are then used to characterise possible overflow conditions [98]. Archer is a static analysis for detecting memory access errors using a custom constraint solver that can express linear relations between two variables [189]. The tool uses heuristics on function names to infer relationships between function arguments and return values. The authors observe that a major deficit of their analysis is the inability to track NUL positions. Eau Claire is another checker for buffer overflows that uses theorem provers [45]. Elgaard et al. show how to find all NULL pointer dereferences using a model checker [73]. Unfortunately, their technique is only sound and complete on straight-line code. User annotations are necessary to handle loops, in which case completeness is lost. Jones and Kelly [108] augment programs with run-time information on buffer bounds. Further afield is the analysis of format string vulnerabilities [162], where an input string is passed to `printf`. The observation is that any percent character in the first argument to `printf` determines how many arguments are read from the stack. Thus, a program may be vulnerable if the user can pass an arbitrary string as the first argument to `printf`. In practice, these vulnerabilities can easily be found and removed syntactically by ensuring that the first argument to `printf` is a constant.

There exists great interest in removing memory management faults, both in academia and industry. Hence, this overview of related tools only includes the most predominant tools that we became aware of during our research.

Value-Range Analysis of C Programs
Towards Proving the Absence of Buffer Overflow
Vulnerabilities

Simon, A.

2008, XXII, 302 p. 119 illus., Hardcover

ISBN: 978-1-84800-016-2