# Static Detection of Buffer Overflows in Executables

Diplomarbeit

Roland Kindermann
University of Bonn

December 22, 2008

**Abstract**

Buffer overflow vulnerabilities have been causing severe damage for the last two decades. This thesis presents a technique that detects stack buffer overflows in executables statically, i.e. without executing the executables.

A fundamental technique used by the buffer overflow detection is the value set analysis [Bal07, BRS04]. The value set analysis computes sets of potential values of all registers and all locations in the memory at all positions in the analysed executable. The approach uses these value sets and applies several heuristics in order to determine whether a given memory access shows suspicious properties. Such a memory access is then considered a buffer overflow.

In addition to the development of the heuristics, two modifications were made to the value set analysis in order to adapt it to the specific requirements of the buffer overflow detection.

The performance of the presented approach was evaluated both using synthetic test cases by Kratkiewicz [Kra05] and the media streaming server Icecast [Fou].

## Acknowledgements

I would like to thank my supervisor Prof. Dr. Peter Martini for giving me the opportunity to do my diploma thesis in his department and for his helpful remarks. I would like to thank Prof. Dr. Rainer Manthey for his willingness to serve as secondary referee. Many thanks to my advisor Felix Leder for many insightful conversations and his helpful comments on my work and the text of this thesis.

Many thanks to Inken Körber for her support and her patience, especially during the stressful final days of my thesis. Last but not least, I would like to thank my parents, my family and my friends for their support during the writing of this thesis and my entire studies.

# Contents

# Chapter 1

# Introduction

Buffer overflows have been a well known class of vulnerabilities since the Morris worm [Spo88] used buffer overflows to infect many computers connected to the internet in 1988. They are not only easy to overlook during development but also in many cases easy and conveniently to exploit. As detecting buffer overflows manually is difficult and time-consuming, it is desirable to detect buffer overflows automatically. This thesis describes a technique that automatically detects stack buffer overflows in executables.

There are two main classes of vulnerability detection approaches. Dynamic approaches try to identify vulnerabilities by executing the analysed program or parts of the analysed program. Static techniques in contrast try to identify buffer overflows without executing the analysed program. A major advantage of static approaches is that they are also able to identify buffer overflows that occur only under very special conditions. The approach presented in this thesis performs static analysis.

The main advantage of analysing executables rather than the high level language source code is that the approach can also be used when the source code of the analysed program is not available. Furthermore, programs written in different high level languages can be analysed while approaches that analyse high level code usually focus on just a single programming language. The major downside of analysing executables is the lack of information about variable and especially buffer boundaries.

A fundamental technique used by approach described in this thesis is the value set analysis [Bal07, BRS04]. The value set analysis computes sets of potential values for all registers and all locations in the memory at all positions in the analysed executable. These sets allow to compute for every memory access which locations in the memory are potentially accessed. Based on this information, four different heuristics are used to identify memory accesses that show suspicious properties. These memory accesses are then considered buffer overflows.

In addition to the development of the heuristics, two modifications to the value set analysis are made in order to adapt it to the specific requirements of the buffer overflow detection.

A set of synthetic test cases [Kra05] is used for the evaluation of the approach. Using these test cases, a number of properties of buffer overflows that have a effect on whether the buffer overflow is detected are identified. In addition, it

is investigated how memory layout, calling conventions used and the value set analysis modifications affect the performance of the buffer overflow detection. The described technique is compared with five tools that detect buffer overflows in C code using results by Kratkiewicz [Kra05].

In addition to the evaluation using the synthetic test cases, the presented approach is used to analyse the media streaming server Icecast [Fou].

# Chapter 2

# Fundamentals

This chapter describes background information about buffer overflow detection. It splits into five parts. First, some related approaches are described. The second section covers fundamentals concerning executables and assembler code. The third section explains buffer overflows and how they can be exploited and describes challenges in detecting them. The last two sections explain two analysis techniques for assembler code. Reaching definitions analysis tries to determine for every position in the program, where each register's value possibly was set. The value set analysis provides information about the potential values of variables and registers. This information is fundamental for the presented buffer overflow detection approach.

## 2.1 Related work

To my knowledge, no technique for static detection of buffer overflows in executables has been published so far.

Also, few other techniques for the static detection of vulnerabilities other than buffer overflows in executables exist. One approach that actually tries to statically identify vulnerabilies in executables is [CFBV06] which tries to identify tainted data vulnerabilities using symbolic execution.

A multitude of tools that statically detect buffer overflows in high level language code exist. Some of them are:

ITS4 [VBKM00] is a tool that performs lexical analysis, i.e. it searches for the use of suspicious functions like `strcpy` in the code. While there are some heuristics that estimate how likely it is, that a given use of a suspicious function really is a buffer overflow, ITS4 can however not distinguish between a secure use and an insecure use of a suspicious function. Other tools that perform lexical analysis are Flawfinder [Whe] and RATS [Sec].

BOON [WFBA00] aims to identify buffer overflows in string variables caused by C string manipulation functions. BOON generates a set of constraints on the variables in the analysed program. For string buffers, both constraints on the size of the buffer and constraints on the length of the string stored in the buffer are generated. The constraints are then used to determine the possible range of the buffer sizes and string lengths. These ranges allow to determine whether it is possible that the string stored in a buffer is longer than the size of the

buffer. The analysis of BOON is control flow insensitive, i.e. no assumptions about the order in which the string manipulation functions are called are made. This improves the performance of the approach but may cause problems with functions that are not idempotent.

Splint [LE01, EL02] aims to find buffer overflows as well as several other types of bugs like e.g. possible dereferences of nil-pointers or memory leaks. Like BOON, Splint generates a set of constraints for variable values and buffer sizes. Unlike BOON, Splint performs a control flow sensitive analysis. Consequently, the constraints are imposed on pairs of variables and positions in the program. These constraints are then used to determine for every memory access, whether or not it is a possible buffer overflow. The basic analysis of Splint is *intra*procedural. Splint relies on annotations made to the source code by the user for *interprocedural* analysis. These annotations specify preconditions, i.e. conditions that have to be fulfilled when the function is called, and postconditions, i.e. ones that are guaranteed to be fulfilled when the function returns.

ARCHER [XCE03] traces the possible control flow paths in the analysed program and maintains a set of constraints on scalar, pointer and array variables. These constraints can then be used in order to determine which buffer accesses and pointer dereferences are buffer overflows. For *inter*procedural analysis, ARCHER generates "triggers" for each functions, i.e. conditions that lead to buffer overflows if they are fulfilled at the function call. Also, ARCHER uses a statistical approach to identify two types of library functions related to buffers: allocation functions that receive a size argument and allocate a buffer of that size and functions that use buffers and receive the buffer's size as argument. This way, ARCHER can both update the constraints on buffer sizes according to library functions and identify buffer overflows in library functions without analysing their source code.

UNO [Hol02] uses a model checking approach to identify different types of bugs including the use of initialised variables, dereferencing of nil-pointers and user defined classes of bugs. In this process, UNO also computes ranges of potential values for variables used as indices and for buffer sizes and it checks for potential buffer overflows.

CSSV [DRS03] translates the analysed program into an indeterministic integer program containing assertions in a way that an assertion in the integer program potentially fails if the analysed program contains a buffer overflow. Then, an integer analysis algorithm that identifies linear inequalities among the variables is used. These inequalities are then used to determine, whether or not the asserted expressions are always fulfilled. Similar to Splint, CSSV makes use of user-specified pre- and postconditions for each function. CSSV is sound, meaning that every runtime error will be detected. On the downside, CSSV's time and memory requirements are very high.

PolySpace [Tec] is a commercial verifier that among other bugs also identifies buffer overflows. The exact methods used by PolySpace have not been published. PolySpace is however still worth noting due to the fact that it has been shown to perform very well both in synthetic [Kra05] and in realistic [Zit03] scenarios.

Furthermore, there is a wide variety of dynamic buffer overflow detection approaches. One example is StackGuard [CPM+98]. StackGuard tries to detect stack smashing attacks [One96] at runtime and to terminate the process before an attacker can gain control. A pseudorandom value, the so called ca-

nary, is placed immediately before each return address on the stack. Thus, most buffer overflows that overwrite the return address will overwrite the canary first. Whenever a function returns, it is checked whether the canary before the function's return address has been modified and the current process is terminated if this is the case. An improved variant of the canary mechanism has found its way into GCC [Eto03]. A major disadvantage of the canary-based approaches is that they do not prevent denial of service attacks.

A multitude of different techniques for the analysis of binary code exists. Two of these techniques are used in this thesis: the value set analysis [Bal07, BRS04] and reaching definitions analysis [ALSU07]. The value set analysis computes a set of potential values for every register and every location in the memory at very position in the program (cf. section 2.5). The reaching definitions analysis is a data flow analysis technique tries to determine for every variable at every position in the program the set of positions where the variable value was possibly modified last (cf. section 2.4).

## 2.2 Analysing executables

This section gives a short introduction to the analysis of executables. First, the conversion of executables to assembler code is explained. Then, several aspects of assembler code including instructions and operands, control flow modifying instructions and function calls are covered. The last subsection introduces the control flow graph.

### 2.2.1 Disassembling executables

Before an executable is analysed, the binary file is first converted into assembler code using a disassembler. The disassembler determines, which parts of the executable contain code and which contain other data and, as most instructions are represented by multiple bytes in the executable, which bytes belong to the same instruction. Then, the binary representation of each instruction is converted back to the instruction mnemonic and a more convenient representation of the operands.

The disassembler used in course of the buffer overflow detection is IDA Pro [Ida]. Among other things, IDA Pro also provides information about the control flow, function boundaries and cross references, e.g. which function is called where.

### 2.2.2 Operand types

Each assembler instruction uses zero to three operands. E.g. the instruction `mov` takes two operands and stores the value of the second one in the first one. The instruction `jmp` takes one operand and lets the control flow jump to the address specified by the operand.

There are three types of operands: *immediate* operands, *register* operands and *memory* operands [Cor99a]. Immediate operands simply represent a constant value. E.g. the instruction `add ..., 42` adds the value 42 to the first operand. In some cases, immediate values are noted in hexadecimal form as

well which is denoted by a trailing h. E.g. `add ..., 42` may as well be written as `add ..., 2Ah`.

x86 processors have several registers or accumulators. They are used among other purposes for storing intermediate results. Registers are accessed using register operands denoted by the name of the used register. Examples of registers are `eax`, `ebx`, `ecx` and `edx`.

Memory operands are used to read from or write to the memory. Memory operands are denoted by an expression in square brackets that specifies the address that is accessed. E.g. the instruction `mov [1234], 1` stores the value 1 at the address 1234 in the memory. The expression between the brackets may be an arbitrary combination of a base register, an offset and a pair of index register and scale. An example of an operand using all possible elements is `[eax+ebx*4+13]`. The operand corresponds to the value at the address resulting from adding the value of the base register `eax`, the offsets 13 and the product of the value of the index register `ebx` and the scale `4`. Also, memory operands may use a segment selector that specifies which memory segment is accessed. If no segment selector is specified as in all examples of this thesis, the segment is automatically chosen by the processor.

### 2.2.3 Instructions affecting the control flow

Usually the instructions in a program are executed in ascending order of their addresses. Some instructions however modify this control flow. The simplest example is the unconditional jump instruction `jmp`. It uses one operand which specifies the address of the instruction at which program execution is continued. Often, jump instructions use immediate operands. In this case, the jump will always point to the same location in the program. Some jumps however use register or memory operands. In this case, the jump may point to different locations depending on the operand's value. Such a jump with a register or memory operand is called an indirect jump.

Similar to the `jmp` instruction, conditional jumps make the control flow jump to the location specified by the sole operand. Conditional jumps however only affect the control flow if certain conditions are fulfilled. The "jump if equal" instruction `je` for instance only performs the jump if the compared values are equal. Which values are compared is not specified in the jump instruction itself but in a previous instruction. This may be done using the compare instruction `cmp`. E.g. the instruction sequence

```
cmp eax, 10
je  ...
```

performs a jump only if the value of the register `eax` is ten. The compare instruction stores the results of the comparison of its operands in a special flags register. The zero flag e.g. is set to one if the compared values are equal and to zero otherwise. The `je` instruction then evaluates the zero flag and jumps only if its value is one.

The compare instruction is not necessarily located immediately before the conditional jump. There might be an arbitrary number of instructions that do not affect the zero flag between the `cmp` and the `je` instruction in the example. Also, the zero flag can be set by other instructions. The `add` instruction e.g.

sets the zero flag if the sum of the two operands is zero. Thus, the sequence of instructions

```
add eax, 1
je  ...
```

increases the value of `eax` by one and performs a jump if the result is zero.

Another instruction that alters the control flow is the `call` instruction which is used to call functions. Like the `jmp` instruction the `call` instructions lets the control flow continue at the address specified by the operand. Unlike after a jump, the control flow can be continued after a call at the instruction immediately following the call instruction, the so-called return address, using the return instruction `ret`.

### 2.2.4 The call stack

Functions are an essential concept in imperative programming. Every function instance has its own private data. This data includes local variables, function arguments and the return address, i.e. the address at which the control flow continues after the current function has returned. All this is stored in the function's stack frame, sometimes called its activation record. The stack frames of all function instances, which are currently active, are stored on the call stack. The call stack is one of the three regions commonly found in the memory of a program. The other two are the heap on which memory can be dynamically allocated and an area for the global variables.

The x86 architecture has two special registers for accessing the call stack: The stack pointer `esp` points to the last value and thus indicates the current top of the stack. The base register `ebp` usually indicates, where the area of the local variables of the current function begins. One important property of the stack is that it grows from the large addresses to the smaller ones, i.e. `esp` is decreased when a value is pushed onto the stack. Also, the stack usually grows downwards in illustrations.

Whenever a function is called, the stack has to be updated accordingly. This process is illustrated in figure 2.1. The initial state of the stack is shown in subfigure (a). First, the caller pushes the function arguments onto the stack which leads to the situation shown in subfigure (b). Then, the call instruction makes the control flow leave the calling function and enter the called one. Also, in order to enable a return to the caller later, the call instruction pushes the return address onto the stack. This is shown in subfigure (c). The first thing the called function usually does is updating the base pointer. In a first step, the old value of the base pointer is pushed onto the stack. Thus, the old value can be restored before the called function returns. Then, the base pointer is set to the value of the stack pointer. Now, the base pointer points to the beginning of the still empty local variables space of the called function. This situation is illustrated in subfigure (d). Finally, the called function can allocate memory for local variables by further decreasing the value of the stack pointer. This is shown in subfigure (e). Of course, how much memory is allocated for local variables can vary during the execution of the function.

When the called function returns, the stack is reset to its original state. First, the stack pointer is set to the current value of the base pointer, i.e. to the situation in subfigure (d). Then, the old base pointer value is popped from

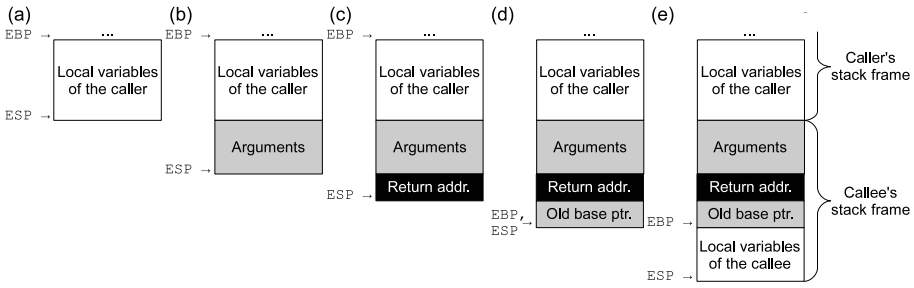



Figure 2.1: A section of the call stack at different stages of a function call.

the stack into the base pointer leading to situation (c). Finally, the return instruction pops the return address from the stack and lets the control flow jump back into the calling function leading to situation (b).

Some variations to the mechanisms explained above exist. E.g. short functions might omit updating the base pointer. Also some functions store the old base pointer and then use the base pointer register for other purposes than to indicate the beginning of the current function's local variables area. In addition, the calling conventions, i.e. the rules by which arguments and return values are passed between functions, may vary. E.g. the called function instead of the caller might remove the arguments from the stack. In this case, the stack would be reset to state (a) when the function returns. In other calling conventions, the arguments are passed in the registers instead of on the stack.

### 2.2.5 The control flow graph

The control flow graph (*CFG*) of a program contains information about the possible control flow paths of the program. It consists of one node for each instruction and one directed edge from each instruction to each possible successor, i.e. to each instruction that is potentially executed directly after the instruction at the tail of the edge. Therefore, instructions like `mov` or `add` only have a single outgoing edge. Jump instructions that have an immediate value as operand and thus always jump to the same location have only one outgoing edge as well. Indirect jumps, i.e. ones that use a register or memory operand in contrast can have an arbitrary number of outgoing edges. Analogously, conditional jumps with immediate operands have two outgoing edges while indirect conditional jumps may have an arbitrary number of successors.

Which successors a call instruction has depends on whether the CFG is an *intra*procedural or *inter*procedural CFG. In the *intra*procedural CFG, the successor of each call instruction is its successor inside the same function, i.e. the instruction to which the called function returns. In the *inter*procedural CFG in contrast, a call's successor is the first instruction of the called function. Thus, a call has only one successor in the intraprocedural CFG but may have many successors in the interprocedural CFG if its operand is not an immediate one, i.e. if the call is an indirect one. Similarly, a return instruction has no successors in the intraprocedural CFG. In the interprocedural CFG in contrast all instructions to which the function may return are the return instruction's

successors.

Figure 2.2 shows a simple program and its intraprocedural (a) and interprocedural (b) CFG. The program contains two functions, one consisting of instructions 2 to 8 and one consisting of instructions 10 and 11. As instruction 5 is a conditional jump, it has two successors in both CFGs. Instruction 6 is a call instruction and thus has different successors in the intraprocedural and the interprocedural CFG. In the intraprocedural CFG, the successor of instruction 6 is instruction 7 which is the next instruction in the same function. In the intraprocedural CFG, the successor is instruction 10 which is the first instruction of the called function. Instruction 11 is a return instruction and thus has no successors in the intraprocedural CFG. In the interprocedural CFG however it has an outgoing edge to instruction 7 which is intraprocedural successor of the call at instruction 6.

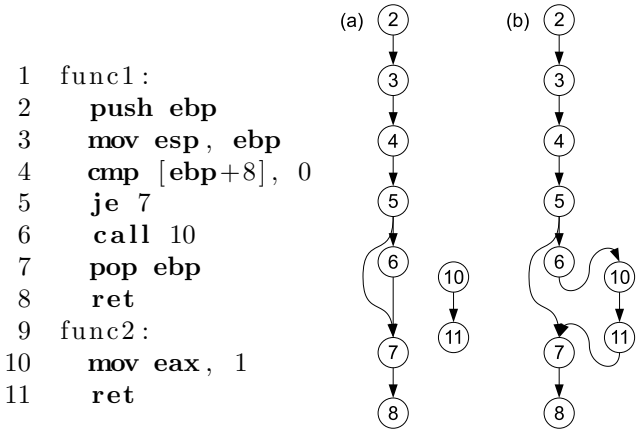



Figure 2.2: A sample program and its *intra*procedural (a) and *inter*procedural (b) CFG.

## 2.3 Buffer overflows and exploits

In this section buffer overflows are explained. The first subsection explains, what exactly a buffer overflow is. Then, ways to exploit buffer overflows are discussed. Finally, a basic overview of the challenges in detecting buffer overflows is given. As the buffer overflow detection approach described in this thesis tries to identify buffer overflows that occur on the stack exclusively, this section and especially subsection 2.3.2 focus on such overflows as well.

### 2.3.1 Buffer overflows

A common concept in high level programming languages are array variables. These variables consist of several elements that can be accessed individually. Thus, when a value from an array is read or written, an index value that specifies which element is accessed is required. As every array only has a limited number of elements there is also only a limited range of valid index values. In this thesis,

the range of valid index values is always considered to be zero to the buffer size minus one.

In programming languages like C or C++, the programmer is responsible for checking that the index values always are inside the valid range. If in such a language an invalid index is used, the address in the memory to which the index value would correspond if it was valid is overwritten. This is then called a buffer overflow. There are however also programming languages like Java or Python that perform bounds checking and in which thus buffer overflows can not occur.

### 2.3.2 Exploits

Stack smashing exploits are a simple yet effective way to gain control over a program that contains a buffer overflow [One96]. The basic idea behind stack smashing exploits is to use a buffer overflow to overwrite a return address and to redirect the control flow this way.

Figure 2.3 shows a simple buffer overflow. In line 1 a buffer of size 10 is created. The loop in lines 3 to 6 overwrites the elements of the buffer. The loop however does not stop when the index variable `i` has the value 9 and thus the end of the array is reached. Instead, further loop iterations overwrite the memory after the buffer.

```
1  char b[10];
2  int i = 0;
3  while(i < 25){
4     buffer[i] = 'A';
5     i = i + 1;
6  }
```

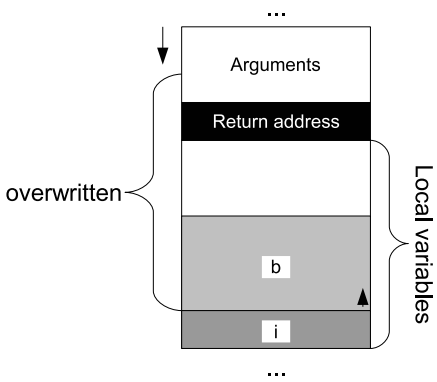



Figure 2.3: A simple buffer overflow and the stack layout of the surrounding function.

The right side of figure 2.3 shows the stack frame (cf. section 2.2.4) of the function surrounding the loop shown on the left side of the figure. As `b` and `i` are local variables, they are located in the lower part of the stack frame. As described in section 2.2.4, the stack grows downwards, i.e. from the larger to the smaller addresses. The indices of the buffer elements grow in opposite direction, i.e. the elements with the higher index values also have the higher address. Thus, the loop first overwrites the lowest buffer element on the stack and then continues upwards. In the eleventh iteration, the value of the index variable `i` is 10 which exceeds the valid range of indices $[0, 9]$. Consequently, a location in the local variables area above the buffer is overwritten instead of a buffer element.

As described in section 2.2.4, the return address determines where the program execution continues after the current function returns. Thus, the buffer overflow shown above alters the control flow of the program by overwriting the return address. If the return address is not overwritten with As but instead with a value chosen by an attacker, that attacker can let the control flow continue after the current function at a location of his choice.

While stack smashing attacks are quite effective as they give the attacker full control over the control flow of the attacked program, there are other exploits as well. If the loop in figure 2.3 would overwrite fewer bytes and thus not reach up to the return address, it might still be possible that an attacker could alter the program execution in a critical way by modifying the values of the local variables between the buffer and the return address. Besides, not only writing but also reading buffer overflows are potentially exploitable. If e.g. the local variables above the buffer would contain a password or a cryptographic key, an attacker might gain access to the key or the password through a reading buffer overflow.

### 2.3.3 Detection

Whether or not a buffer access like the one in line 4 of figure 2.3 is a buffer overflow entirely depends on the potential values of the index variable. Thus, a technique that extracts information about the potential values of variables is required in order to be able to reliably detect buffer overflows. In this thesis, a technique called value set analysis [BRS04, Bal07] is used in order to determine potential values of all locations in the memory and the registers. The value set analysis is covered in section 2.5.

An additional challenge when analysing assembler code is that the information about how the memory is split into individual variables is lost during compilation. Figure 2.4 shows an excerpt from a C program and the corresponding assembler code. In the C code on the left side it is obvious that the program uses two integer variables and one 20 byte buffer. In the assembler code on the right side, the first two instructions update the base pointer. 28 bytes of memory needed for the local variables are allocated in the third instruction. There is however no information about how these 28 bytes are used. They might e.g. just as well be used for seven integer values. Thus, as there is no obvious information about buffers or buffer sizes in the executable, simply examining buffer accesses and index variables does not work. Also, it is not always obvious which memory access is a buffer access in assembler code. Section 3.1 shows a heuristic approach to detecting buffer overflows despite the lack of information about buffers and buffer sizes.

## 2.4 Reaching definitions analysis

A technique used by the buffer overflow detection is the reaching definitions analysis [ALSU07]. Its aim is to determine for every register and every variable at every position in the program, where its current value was possibly set.

Every instruction has sets of *defined*, i.e. changed, registers and memory locations. E.g. the `add` instructions, which adds the value of the second operand to the one of the first and stores the result in the first operand, defines the

```
1  void func(int a){          1  push ebp
2    int i, j;                2  mov ebp, esp
3    char buffer[20];         3  sub esp, 28
4    ...                      4  ...
```

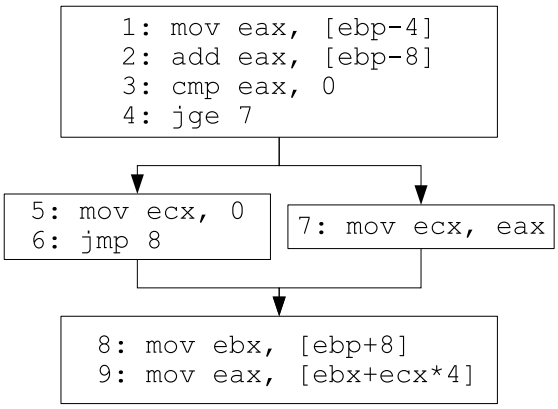
Figure 2.4: The initial lines of a function both in C code and in assembler code. While the C code contains detailed information about the usage of the memory, this information is missing in the assembler code. Line two and three of the C code are merged into just one instruction in line three of the assembler code.

first operand. The `xchg` instruction exchanges the values of its first and its second operand and consequently defines both operands. Other instructions may implicitly use or define registers or memory locations that are not given as operands. E.g. the `pop` instruction which pops a value from the stack and stores it in its operand defines its operand and the stack pointer. Each definition of a register or variable kills all previous definitions of that register or variable. A definition *reaches* a position in the CFG, if a path from the definition to the position exists on which the definition is not killed. Formally, the aim of the reaching definitions analysis is to identify for each location in the program the set of reaching definitions of each register and variable.

In the course of the buffer overflow detection, reaching definitions analysis is performed for registers exclusively and only intraprocedurally, i.e. based on the intraprocedural CFG. Each call instruction is conservatively considered to define all registers.

A simple algorithm [ALSU07] is used for reaching definitions analysis. The algorithm traces the control flow graph and updates the sets of reaching definitions according to the visited instructions. Two sets of reaching definitions are computed for each instruction. One reflects the situation before and one the situation after the instruction. If a definition of a register reaches the position after a given instruction, it obviously reaches the position before each of its successors as well. Based on this observation, the algorithm computes the set of definitions of every register before an instruction as the union of the sets of reaching definitions after all predecessors of that instruction. The set of reaching definitions after an instruction for a given register depends on whether the instruction defines that register. If the instruction does define the register, then the set of reaching definitions after the instruction is a set only containing the instruction in question. If in contrast the instruction does not define the register, then the set of reaching definitions after the instruction is the same as the one before the instruction. Based on these observations, the algorithm continues to set the set of reaching definitions before each instruction to the union of the ones after each predecessor. Then it computes the sets of reaching definitions after each instruction by updating the sets before according to the instruction until a fixpoint is reached, i.e. until further analysis does not yield any changes anymore.

Figure 2.5 shows a short program (a) and the results of the reaching definitions analysis (b). At the beginning of the program, no definitions reach the current position and thus the reaching definitions sets are empty. Instruction 1 in the figure defines `eax`. Consequently, the reaching definitions set is {1}




(a)

| Instruction | Defined | Reaching definitions | | | |
|---|---|---|---|---|---|
| | | Position | eax | ebx | ecx |
| 1 | eax | before | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| | | after | {1} | $\emptyset$ | $\emptyset$ |
| 2 | eax | before | {1} | $\emptyset$ | $\emptyset$ |
| | | after | {2} | $\emptyset$ | $\emptyset$ |
| 3 | - | before | {2} | $\emptyset$ | $\emptyset$ |
| | | after | {2} | $\emptyset$ | $\emptyset$ |
| 4 | - | before | {2} | $\emptyset$ | $\emptyset$ |
| | | after | {2} | $\emptyset$ | $\emptyset$ |
| 5 | ecx | before | {2} | $\emptyset$ | $\emptyset$ |
| | | after | {2} | $\emptyset$ | {5} |
| 6 | - | before | {2} | $\emptyset$ | {5} |
| | | after | {2} | $\emptyset$ | {5} |
| 7 | ecx | before | {2} | $\emptyset$ | $\emptyset$ |
| | | after | {2} | $\emptyset$ | {7} |
| 8 | ebx | before | {2} | $\emptyset$ | {5, 7} |
| | | after | {2} | {8} | {5, 7} |
| 9 | eax | before | {2} | {8} | {5, 7} |
| | | after | {9} | {8} | {5, 7} |

(b)

Figure 2.5: An example program (a) and the results of the reaching definitions analysis (b).

afterwards. Then, the sets of reaching definitions before instruction 2 is set to the unions of the sets after all predecessors. As instruction 2 is the sole predecessor, the sets before instruction 2 are equal to the ones after instruction 1. Instruction 2 redefines `eax` and thus changes the set of reaching definitions of `eax` to $\{2\}$ . Instructions 3 and 4 do not define any registers and thus do not change the sets of reaching definitions. After instruction 4, the control flow is split into two alternating paths. The sets of reaching definitions before each of the alternating successors 5 and 7 is set to the ones after instruction 4. The left path defines `ecx` in instruction 5 and thus the set of reaching definitions of `ecx` is updated accordingly. Analogously, `ecx` is defined in instruction 7 in the right path. Before instruction 8, the control flow paths meet again. Thus, the sets of reaching definitions are set to the unions of the sets after instructions 6 and 7. As a result, the set of reaching definitions of `ecx` now contains both instruction 5 and instruction 7. Instruction 8 defines `ebx` and instruction 9 `eax` and the sets of reaching definitions are updated accordingly. Now, all instructions have been analysed and as the program does not contain any loop, a fixpoint is reached. If however the program did contain a loop, multiple analysis iterations of the loop might be necessary.

## 2.5 The value set analysis

Whether or not a given buffer access is a buffer overflow entirely depends on the index value used. Thus, buffer overflows can not reliably be detected without information about the potential values of the variables or, in case of assembler code, the operands (cf. section 2.3.3). In order to retrieve this information, a technique called value set analysis [Bal07, BRS04] which computes the sets of potential values for all registers and all locations in the memory is used. Section 2.5.1 explains the basic idea behind the value set analysis. Section 2.5.2 describes how conditional jumps are analysed by the value set analysis. Special treatment is required to speed up the analysis of loops. The techniques used for this purpose are discussed in sections 2.5.3 and 2.5.4. Sections 2.5.5 and 2.5.6 describe the abstract memory model used by the value set analysis and the representation of value sets. Finally, the treatment of function calls is discussed in section 2.5.7

For this thesis, not the original implementation of the value set analysis was used. The used implementation of the value set analysis only supports a basic set of features and not the full set described in [Bal07, BRS04].

### 2.5.1 Basic principle

The value set analysis (*VSA*) tries to compute a value set, i.e. a set of potential values, for every register and every variable. The value sets can contain both integer and pointer values.

The VSA is based on the observation that the possible effects of an instruction can be estimated if the sets of potential values for all used operands are known. E.g. it might be known that `eax` is always either 1 or 2 and `ebx` is always either 3 or 4 at a given position in the program. The instruction at that position might be `add eax, ebx` which stores the sum of `eax` and `ebx` in `eax`. Then, the set of potential values of `eax` after the instruction would contain all

sums of possible values of `eax` and `ebx` before the instruction. Thus, the value of `eax` after the instruction would be element of $\{1+3, 1+4, 2+3, 2+4\} = \{4, 5, 6\}$.

The value sets before the starting point of the analysis are initialized as $\top$ which is the set of all possible values. Similarly to the reaching definitions sets in section 2.4, the value sets before any other instruction simply are the unions of the value sets after all predecessors of the instruction. The VSA traces the CFG and alternately updates the value sets according to the current instruction and the propagates the updated value sets to all successors. This is continued until a fixpoint is reached.

Figure 2.6 shows a simple assembler program and its CFG with the value sets computed by the VSA for `eax` and `ebx`. Initially, both value sets are set to $\top$. Instruction 1 is a move instruction and assigns the value 3 to `ebx`. Consequently, the value set of `ebx` is $\{3\}$ after the instruction. Instruction 2 and 3 are one compare and one conditional jump instruction and do not affect the value sets of `eax` and `ebx`. Instruction 3 however splits the control flow into two alternating paths. One path changes the value of `eax` to 1 and one path changes it to 2. The alternating control flow paths meet at instruction 7. Thus the value sets are set to the union of the value sets after instruction 5 and 6. Consequently, the value set of `eax` is $\{1, 2\}$ before instruction 7. Finally, instruction 7 stores the sum of `eax` and `ebx` in `ebx`. Hence, the value set of `ebx` becomes $\{3+1, 3+2\} = \{4, 5\}$.

```
1  mov ebx, 3
2  cmp eax, 1
3  je  6
4  mov eax, 1
5  jmp 7
6  mov eax, 2
7  add ebx, eax
```

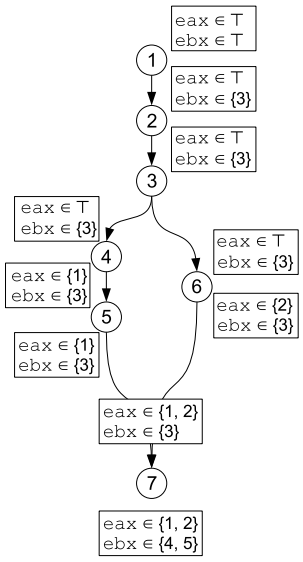



Figure 2.6: An assembler program, its CFG and the value sets computed by the VSA.

### 2.5.2 Conditional jumps

Despite the fact that conditional jumps do no modify any values, the VSA still uses them to update value sets. Whether or not a given conditional jump performs a jump depends on whether or not the condition is fulfilled. The value sets of the operands used by the conditional jump can be split into a subset

of values that potentially fulfil the jump condition and a subset of values that potentially do not fulfil the condition. Only the values that potentially fulfil the condition have to be propagated to the location to which the control flow jumps if the condition is fulfilled. Analogously, only the values that potentially do not fulfil the condition have to be propagated to the instruction at which the control flow continues if the condition is not fulfilled.

Figure 2.7 shows two conditional jumps and their effects on the value sets of the used operands. The conditional jump in subfigure (a) performs a jump only if the value of `eax` is greater than or equal to zero. Thus, only the values below zero remain in the value set of `eax` at the no-jump path in the CFG. Analogously, only the values that are greater or equal zero remain inside the value set at the jump path.

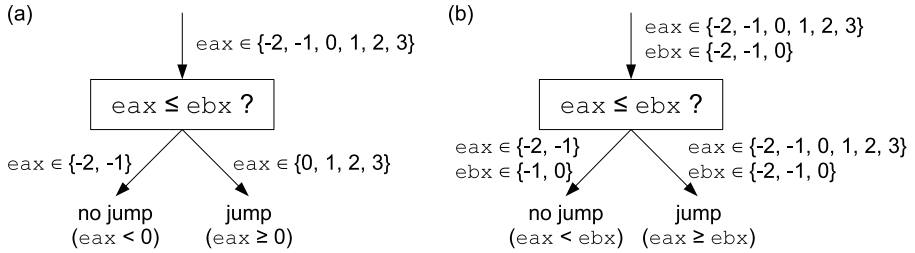



Figure 2.7: Two conditional jumps. The value sets illustrate, how the VSA can use conditional jumps to restrict the value sets.

The conditional jump in subfigure (b) performss a jump only if the value of `eax` is greater than or equal to the value of `ebx`. Thus, this time `eax` is not compared with a constant but with `ebx` which also has more than one possible value. Still it is possible to split the value sets into values that potentially fulfil the condition and ones that potentially do not. Again, the values at the left control flow path are the ones that potentially do not fulfil the condition, i.e. ones that make it possible that the value of `eax` is smaller than the one of `ebx`. If e.g. `eax` was -1 and `ebx` was 0 then `eax` would be smaller than `ebx`. Consequently, -1 is in the value set of `eax` and 0 is in the value set of `ebx` at the left path. If in contrast `eax` is 1, there is no value in the value set of `ebx` that allows `eax` to be smaller than `ebx`. Consequently, 1 is not in the value set of `eax` at the left path.

As -2 is in the value set of `ebx` and every value in the value set of `eax` is greater than or equal to -2, the value set of `eax` at the right path is the same value set as the one before the conditional jump. Analogously, as every value in `ebx` is smaller than or equal to 3, which is in the value set of `eax`, the full value set of `ebx` is propagated along the right path. In general, if the condition of the indirect jump is `eax` $\geq$ `ebx`, all values that are greater than or equal to the upper bound of `ebx` are removed from the value set of `eax` at the no-jump path. Analogously, all values that are smaller than the lower bound of `ebx` are removed at the jump path and the value set of `ebx` is restricted in a similar way.

In some cases there are values in the original value sets that, depending on the value of the other operand, may or may not fulfil the condition. Such values like e.g. -1 in the value set of `eax` in the example are then propagated along both outgoing control flow edges of the conditional jump.

Also, in some cases restricting a value set for one path results in an empty set. If e.g. the value set of `eax` in example (b) was $\{1, 2, 3\}$ , every possible value of `eax` would be greater than every possible value of `ebx`. Hence, the value set of `eax` at the no-jump path would be empty. This in general indicates that the path is infeasible, i.e. if `eax` is either 1, 2 or 3 and `ebx` is either -1 or 0 then the jump will always be performed and the left CFG path will never be followed.

### 2.5.3 Widening

The VSA continues until a fixpoint is reached. As the program in the first example in figure 2.6 does not contain any loops, a fixpoint is reached when every instruction has been analysed once. In programs that do contain loops however, multiple analysis iterations of each loop may be required until a fixpoint is reached.

The example in figure 2.8 illustrates that in some cases a very large number of analysis iterations may be required until a fixpoint is reached. The program contains a loop consisting of three instructions that is at runtime executed ten times. If the VSA is performed as described so far, 33 analysis steps are required in order to reach a fixpoint. Also, the number of analysis steps increases linearly with the number of loop iterations performed at runtime. If e.g. not ten but 100 iterations were performed, 303 iterations would be required.

In order to reduce the number of analysis steps required, the VSA uses a technique called widening. Widening does however not only reduce the number of analysis steps required but also makes the number of analysis steps independent of the number of loop iterations executed at runtime.

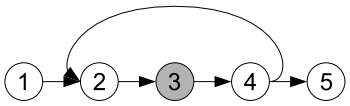
Before the VSA starts, a widening point selection algorithm is executed. The widening point selection algorithm chooses a set of widening points in the CFG in a way that at least one widening point is in every cycle of the CFG [BRS04]. Whenever a widening point is visited at least for the second time during analysis, the value sets of the previous visit and the ones of the current visit are compared and the results of future analysis iterations are estimated. If the upper bound of a value set after the current analysis iteration is larger than the one at the previous visit, the upper bound is set to $\infty$. Similarly, if the lower bound at the current visit is lower than the one at the previous visit, the lower bound of the value set is set to $-\infty$.

Figure 2.8 shows a simple program and the intermediate results of the VSA both with and without widening. Subfigure (a) shows the assembler code of the program and subfigure (b) its CFG. Instruction 1 initializes `eax` to 0. Instructions 2 to 4 contain a loop that increases `eax` by one in each iteration in instruction 2. The `jb` ("jump if below") instruction in line 4 performs a jump back to the loop start if the value of `eax` is smaller than ten.

Subfigure (c) shows the intermediate results of the VSA without widening. Initially, the value set of `eax` is $\top$. In the first step, the VSA analyses the instruction `mov eax, 0` in line 1 and the value set of `eax` becomes $\{0\}$ . The second step analyses the second instruction which increases the value of `eax`. Instruction 3 is a compare instruction which does not modify the value of `eax`. In the fourth analysis step, the conditional jump is analysed. At this position, the control flow jumps back to instruction 2 if the value of `eax` is less than ten, and steps to instruction 5 otherwise. As currently the value set of `eax` is $\{1\}$ and thus only consists of values below 10, the control flow path leading

```
1  mov eax, 0
2  add eax, 1
3  cmp eax, 10
4  jb 2
5  add eax, 1
```

(a)




(b) CFG

| Step | Position | | Value set of `eax` |
|---|---|---|---|
| 1 | 1 | before | $\top$ |
| | | after | $\{0\}$ |
| 2 | 2 | before | $\{0\}$ |
| | | after | $\{1\}$ |
| 3 | 3 | before | $\{1\}$ |
| | | after | $\{1\}$ |
| 4 | 4 | before | $\{1\}$ |
| | | after | $\{1\}$ |
| 5 | 2 | before | $\{0,1\}$ (union) |
| | | after | $\{1,2\}$ |
| 6 | 3 | before | $\{1,2\}$ |
| | | after | $\{1,2\}$ |
| 7 | 4 | before | $\{1,2\}$ |
| | | after | $\{1,2\}$ |
| 8 | 2 | before | $\{0,1,2\}$ (union) |
| | | after | $\{1,2,3\}$ |
| 9 | 3 | before | $\{1,2,3\}$ |
| ... | | | |
| **27** | 3 | after | $\{1,2,\ldots,9\}$ |
| 28 | 4 | before | $\{1,2,\ldots,9\}$ |
| | | after | $\{1,2,\ldots,9\}$ |
| 29 | 2 | before | $\{0,1,\ldots,9\}$ (union) |
| | | after | $\{1,2,\ldots,10\}$ |
| 30 | 3 | before | $\{1,2,\ldots,10\}$ |
| | | after | $\{1,2,\ldots,10\}$ |
| 31 | 4 | before | $\{1,2,\ldots,10\}$ |
| | | after | Depends on successor. |
| 32 | 2 | before | $\{0,1,\ldots,9\}$ (union) |
| | | | *No changes anymore* |
| 33 | 5 | before | $\{10\}$ |
| | | after | $\{11\}$ |

(c) Analysis without widening

| Step | Position | | Value set of `eax` |
|---|---|---|---|
| 1 | 1 | before | $\top$ |
| | | after | $\{0\}$ |
| 2 | 2 | before | $\{0\}$ |
| | | after | $\{1\}$ |
| 3 | 3 | before | $\{1\}$ |
| | | after | $\{1\}$ |
| 4 | 4 | before | $\{1\}$ |
| | | after | $\{1\}$ |
| 5 | 2 | before | $\{0,1\}$ (union) |
| | | after | $\{1,2\}$ |
| 6 | 3 | before | $\{1,2,\ldots,\infty\}$ |
| | | after | $\{1,2,\ldots,\infty\}$ |
| 7 | 4 | before | $\{1,2,\ldots,\infty\}$ |
| | | after | Depends on successor |
| 8 | 2 | before | $\{0,1,\ldots,9\}$ (union) |
| | | after | $\{1,2,\ldots,10\}$ |
| 9 | 3 | before | $\{1,2,\ldots,\infty\}$ |
| | | | *No changes anymore* |
| 10 | 5 | before | $\{10,11,\ldots,\infty\}$ |
| | | after | $\{11,12,\ldots,\infty\}$ |

(d) Analysis with widening

Figure 2.8: A simple assembler program (a), its CFG (b) and the intermediate results of the VSA with (d) and without (c) widening.

to instruction 5 seems infeasible based on the current results. The value set of `eax` is propagated back along the jump edge in the CFG and merged with the one coming from instruction 1 before instruction 2. As a result, the value set before instruction 2 now becomes $\{0, 1\}$. Then, the VSA analyses the loop again in step 5 to 7. When the analysis reaches the conditional jump again in step 7, the value set of `eax` is $\{1, 2\}$ and thus all values are still smaller than ten. Hence, again only the path back to instruction 2 seems feasible. The new value set before instruction 2 is $\{0, 1, 2\}$ . The subsequent analysis iterations proceed in the same way. After each iteration, the upper bound of the value set of `ebx` is increased by one. Eventually, the value set before instruction 2 becomes $\{0, 1, \ldots, 9\}$ in analysis step 29. When the conditional jump in instruction 4 is again reached in step 31, the value set of `eax` is $\{1, 2, \ldots, 10\}$ and thus contains both values below ten and ones that are greater than or equal to ten. Consequently, both outgoing CFG edges seem feasible now. The value set is split into $\{1, 2, \ldots, 9\}$ , which is propagated back to instruction 2 and $\{10\}$ which is propagated to instruction 5 (cf. section 2.5.2). The resulting value set before instruction 2 again is $\{0, 1, \ldots, 9\}$ . Thus, as the new value set is equal to the old one, continuing the VSA at instruction 2 would not lead to new value sets in the subsequent instructions either. Thus, the VSA stops analysing the current CFG path and continues at instruction 5 instead. The value set $\{10\}$ before instruction and is updated to $\{11\}$ after the instruction. Now, a fixpoint is reached at last.

Although the program consists only of five instructions and the loop even consists of only three instructions, 33 analysis steps have to be executed before a fixpoint is reached. Also, if the compare instruction in line 3 was `cmp eax, 100` instead of `cmp eax, 10`, then 303 analysis steps would be required. In general, the number of analysis steps required grows linearly with the upper bound of `eax`. This leads to unacceptable time requirements in realistic scenarios.

Subfigure (d) shows the intermediate results of the VSA with widening. It is assumed, that instruction 3 was chosen as widening point. The first five analysis steps lead exactly to the same results as in the first example without widening. In step 6 however the widening point at instruction 3 is reached for the second time. The old value set from the first time instruction 3 was analysed in step 3 is $\{1\}$ . The new value set is $\{1, 2\}$ . Thus, as the new upper bound is larger than the old one, the upper bound of the value set is set to $\infty$. Hence, the new value set is $\{1, 2, \ldots, \infty\}$ which still is the value set of `eax` when the conditional jump in instruction 4 is analysed the next time in step 7. Unlike in the first example, the value set of `eax` now already contains values both below and above ten. Thus, like in step 31 of the first example, the value set is split into two subsets. The set $\{1, 2, \ldots, 9\}$ is propagated back to instruction 2 and the value set $\{10, 11, \ldots, \infty\}$ is propagated to instruction 5. In the next step, instruction 2 is analysed and the value set is updated from $\{0, 1, \ldots, 9\}$ to $\{1, 2, \ldots, 10\}$ according to the instruction. This value set is then propagated to instruction 3 and there merged with the old widened value set $\{1, 2, \ldots, \infty\}$ resulting in the value set $\{1, 2, \ldots, \infty\}$ again. Thus, there are no changes at the current control flow path any more and the VSA stops tracing it and continues at instruction 5. Here, the value set before the instruction is $\{10, 11, \ldots, \infty\}$ which is updated to $\{11, 12, \ldots, \infty\}$ after the instruction.

With widening, the VSA only needs ten analysis steps instead of 33. Even more important is that this number does not depend on the upper bound of

`eax` any more. If the compare instruction in line 3 was `cmp eax, 100` or even `cmp eax, 10000` instead of `cmp eax, 10` the VSA would still only need ten analysis steps. The downside however is that the computed value sets are now less precise. The value sets at instruction 3 and 4 now are $\{1, 2, \ldots, \infty\}$ instead of $\{1, 2, \ldots, 10\}$ . Similarly, the value sets at instruction 10 are much less precise than the one in the VSA version without widening. Section 3.2.1 shows a technique that tries to mitigate the loss of precision at least at instructions 3 and 4 in this example.

### 2.5.4 Widening point selection

As described in section 2.5.3 the VSA uses a widening point selection algorithm that places at least one widening point in each cycle in the CFG. The widening point selection algorithm is based on a technique called hierarchical decomposition [Bou93] which is also essential for the improved widening approach described in section 3.2.1.

Hierarchical decomposition of a program's CFG yields a so-called weak topological ordering. The definition of a weak topological ordering ($WTO$) is based on the definition of a hierarchical ordering. A *hierarchical ordering* of a CFG is a well-parenthesized permutation of the program's instructions that does not contain two consecutive left parentheses. A hierarchical ordering of the CFG in figure 2.9 e.g. is 1(2(43))56. The instructions between matching parentheses in a topological ordering are called a *component*. The first instruction in each component is called its *head*. E.g. the hierarchical ordering 1(2(43))56 of the CFG in figure 2.9 contains the component 243 whose head is 2 and the component 43 whose head is 4. Like instructions 3 and 4 in the example, instructions can be members of multiple components. As no two consecutive left parentheses are allowed, each instruction can however be the head of at most one component.

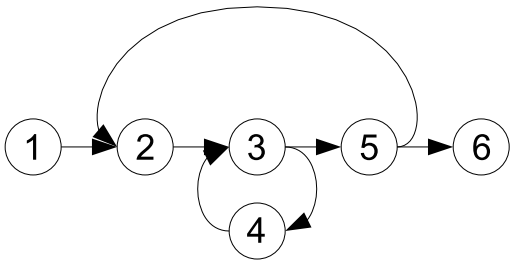



Figure 2.9: The CFG of two nested loops.

A *feedback edge* in a hierarchical ordering is an edge in the CFG whose head is either equal to its tail or is located left of its tail in the hierarchical ordering. In case of the example hierarchical ordering 1(2(43))56, the CFG edge $(3, 4)$ is a feedback edge as 3 is located left of 2 in the hierarchical ordering. Similarly, the edge $(5, 2)$ is a feedback edge. A hierarchical ordering is a *weak topological ordering* (WTO) if the head of every feedback edge is also head of a component and the tail of every feedback edge is a member of the edge's head's component. In the example hierarchical ordering 1(2(43))56, both conditions are fulfilled for the feedback edge $(3, 4)$ as 4 is head of the component 43 and 3 is a member

of that component. The conditions are however not fulfilled for the feedback edge $(5, 2)$ as instruction 5 is not a member of the component of 243 of which instruction 2 is the head.

A hierarchical ordering of the example CFG that does fulfil the conditions and thus is a WTO is 1(2(34)5)6. This WTO contains the feedback edges $(5, 2)$ and $(4, 3)$. The conditions that make a hierarchical ordering a WTO are fulfilled for both feedback edges as instruction 5 is member of instruction 2's component 2345 and 4 is a member of instruction 3's component 34.

Every cycle in a CFG contains in every WTO of that CFG at least one feedback edge, namely the edge from its instruction that is located rightmost in the WTO to its successor in the cycle. Thus, as every head of a feedback edge is also head of a component in the WTO, every cycle contains a component head of the WTO. Hence, selecting all component heads in an WTO as widening points guarantees that at least one widening point is located in every CFG cycle [Bou93]. The widening point selection algorithm used by the VSA is based on this observation.

Whether or not the widening point selection is a good one however strongly depends on the used WTO. E.g. (6(5(4(3(2(1)))))) is a valid WTO of the CFG in figure 2.9 as well. In this WTO however, all instructions are heads of components and thus all instructions would be selected as widening points which obviously is a bad selection. Fortunately, the WTOs computed by the hierarchical decomposition algorithm usually resemble the loops in the CFG very closely, i.e. each loop forms a component in those WTOs. Hence, usually only one widening point per loop is selected. E.g. the WTO computed for the CFG in figure 2.9 would be 1(2(34)5)6. This WTO contains the component 2345 corresponding to the outer loop and the component 34 corresponding to the inner loop. Based on this WTO, instruction 2 and 3 would be chosen as widening points.

As the hierarchical decomposition of large graphs takes too long, it is only used intraprocedurally and a different algorithm is used in order to place at least one widening point in every interprocedural loop [Bal07].

### 2.5.5 The abstract memory model

So far, only the value sets of registers have been discussed. Aside from some special challenges like aliases, the same approach could also be used to compute value sets for high level language variables. In assembler code however there are no variables. Instead, the locations where the values are stored are specified by their address. The address of a local variable may however vary during execution as the function's stack frame may start at different locations depending on where the function was called. Still, it is desirable to treat each local variable as one entity. In order to make this possible, the VSA uses an abstract memory model [BR06, Bal07, BRS04].

The abstract memory model splits the memory into several logical regions called memory regions. There are two main types of memory regions: stack regions and the global memory region. Each function has its own stack region denoted by $stack\_frame_{function\_name}$. Each function's stack region corresponds to its stack frame and thus consists of the function's arguments, the return address, and the local variables. The global memory region consists of the memory used for global variables. No assumptions about where exactly each

memory region lies relative to each other memory region are made apart from when call and return instructions are analysed (cf. section 2.5.7)

In the abstract memory model, locations are not described by their address but by a pair consisting of the memory region in which the location lies and its offset inside that memory region. In the global memory region, the offset of a location simply is its address. Therefore, the operand [1234] corresponds to the location $(global\_memory\_region, 1234)$. In stack regions of functions using the base pointer (cf. section 2.2.4), the offset of the location to which the base pointer points usually is 0 inside the current function's stack memory region. Thus, the operand $[ebp + x]$ corresponds to the location $(stack\_frame_{func}, x)$ in such stack frames. In functions that do not use the base pointer, the offsets are at least consistent, meaning that the difference between the offsets of the operands `[esp+x]` and `[esp+y]` is equal to the difference between `x` and `y`.

The abstract memory model uses variable-like entities called abstract locations (*a-locs*). An a-loc specifies where in the memory a value has been stored. As values in the memory often use several bytes, an a-loc is a pair of a memory region and a range of offset. E.g. the instruction `mov [ebp-4], 1` in the function `func` stores the four byte value 1 at the a-loc $(stack\_frame_{func}, [-4, -1])$.

In order to be able to express every location at which values can be stored as a-loc, a special "memory" region for registers is introduced as well. This way, each register can be specified as a pair of the register region and the register ID as offset. Also, special memory regions can be used to handle heap variables. The VSA implementation used for the buffer overflow detection however does not support these heap regions.

### 2.5.6 The representation of value sets

So far, value sets were used in several examples. This section describes how the integer and pointer values are modelled and how large value sets are stored efficiently.

As described in the previous section, locations in the abstract memory model are pairs of a memory region and an offset. Thus, as pointer values specify locations in memory, they are represented as pairs of a memory region and an offset as well. E.g. the instruction `lea` loads a pointer to its second operand into its first operand. Thus, the value set of `eax` after the instruction `lea eax, [ebp-8]` in the function `func` is $\{(stack\_frame_{func}, -8)\}$ .

The representation of integer values is based on the observation that integer values differ from pointers to global variables only in their usage. If e.g. the value 1234 is assigned to `eax`, `eax` might be used as pointer to the global variable at address 1234 later. `eax` might however be used for numeric calculations using the integer value 1234 as well. Thus, integers are simply represented as pairs of the global memory region and their value as offset. Consequently, $(global\_memory\_region, 1234)$ can be both the integer value 1234 and the value of a pointer to the address 1234.

Value sets in the VSA are pairs of memory regions and offsets. If e.g. an a-loc at a given position in the executable may contain either the values 1 or 2 or a pointer to the variable at offset -4 in the stack region of the function `func` then its value set is $\{(global\_memory\_region, 1), (global\_memory\_region, 2), (stack\_frame_{func}, -4)\}$. However, value sets are usually noted as mappings from memory regions to the set of potential offsets in those memory regions.

Noted this way, the value set just mentioned becomes $\{(global\_memory\_region \mapsto \{1, 2\}), (stack\_frame_{func} \mapsto \{-4\})\}$.

Value sets can become quite large. While the total number of memory regions is limited, the offset sets for one memory region can contain a large number of values. Thus, an efficient representation of sets of offsets is required. As long as the number of offsets in a set stays below a certain user-defined threshold, all values in the set are stored explicitly. If however the size of a set exceeds the threshold, the set is converted into a reduced intervall congruence ($RIC$) [BRS04]. RICs are noted expressions in the form of $a[b, c] + d$ corresponding to $\{a \times x + d \mid b \leq x \leq c\}$ . Of course, many sets can not be represented as RICs. In such cases, the smallest superset representable as RIC is used. E.g. the offset set $\{3, 7, 9, 13\}$ would be converted to $2[1, 6] + 1 = \{3, 5, 7, 9, 11, 13\}$.

### 2.5.7 Function calls

As described in section 2.2.4, call instructions push the return address onto the stack and then jump to the first instruction of the called function. Also, the arguments are moved from the caller's stack frame to the callee's one. During program execution, stack frames only are a logical partition of the stack. In the VSA however, the stack frame is part of the specification of each location inside the frame. Thus, the arguments have to be moved actively from the caller's stack memory region to the callee's one. This is also illustrated in figure 2.10. The left side of the figure shows the memory region of the caller before the function call and the right side shows both memory regions after the call. In the transition from the left situation to the right one, the arguments have to be moved to the new stack frame. Analogously, the values of the arguments have to be moved back to the caller's stack frame when the function returns.

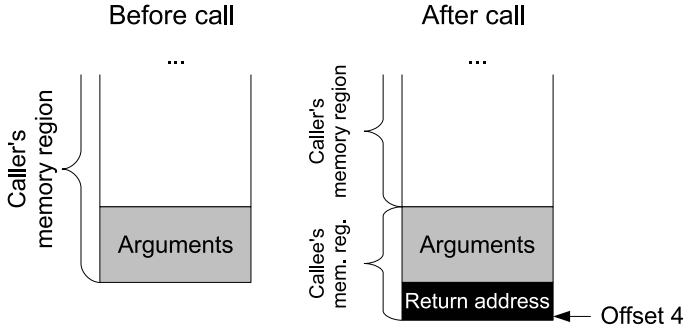



Figure 2.10: The caller's memory region before a call and the caller's and the callee's stack frame after the call.

Not only the arguments but also the value of the stack pointer has to be updated to point to the memory region of the called function after the call. In principle, the stack pointer could be set to any offset in the memory region of the called function as long as the offset is always the same and the offsets of the arguments are consistent with the value of the stack pointer. In practice, the stack pointer is set to offset 4 after the return address has been pushed onto the stack. This way, the base pointer points to offset 0 if the base pointer

is updated as described in section 2.2.4. Thus, operands in the form `[ebp+x]` correspond to the offset x in the current stack memory region which simplifies manual evaluation of the results. If however the base pointer is updated in a different way, it may also point to a different offset.

# Chapter 3

# Methods

The buffer overflow detection approach consists of two main parts. First, section 3.1 describes four heuristics which try to identify memory accesses that are likely buffer overflows based on the results of the VSA. The second part of the approach introduces two modifications made to the VSA in order to adapt it to the needs of the buffer overflow detection.

## 3.1 Heuristics

The VSA provides a value set for each a-loc at each position in the program. These value sets can be used to determine which locations a given instruction operand potentially accesses. In order to determine whether a given operand is probably a buffer overflow, several heuristics are used. Section 3.1.1 discusses their application in more detail. The heuristics themselves are described in sections 3.1.2 to 3.1.5. They aim primarily for the detection of buffer overflows on the stack.

### 3.1.1 Basic approach

Three different types of instruction operands exist: Immediate operands, which are constant values, register operands and memory operands. Memory operands can be any combination of of a base register, an index register whose value is multiplied by a scale and a fixed displacement [Cor99a]. An example of a memory operand is `[eax+ebx*4+42]` where `eax` is the base register, `ebx` is the index register, 4 is the scale and 42 the displacement. Other examples of memory operands are `[ecx]`, `[edx-42]` or `[1234556]`. When a memory operand is used, the value at the address in memory that yields from evaluating the expression inside the square brackets is read or written.

Buffer overflows can only occur when the memory is accessed. Hence, every buffer overflow in high level code is translated into one or several instructions with memory operands in the executable. Therefore, buffer overflows can be detected in executables by identifying suspicious memory operands. There is however no simple way to tell whether a memory operand is suspicious as there is neither any information about the locations and sizes of buffers nor about which, if any, buffer an operand originally was intended to access. Thus, different

heuristics are used to identify operands that show characteristics potentially resulting from buffer overflows.

The VSA allows not only to determine a value set for every operand but also to identify the location sets, i.e. the sets of potentially accessed locations, for register and memory operands. For register operands, the location set contains only a singly element, namely the pair of the register region and the register ID. At runtime, the address accessed by a memory operand is the result of the expression in between the square brackets, i.e. the sum of the base register, the offset and the product of index register and scale. Using the value sets of base and index register it is possible to determine the potential values of that expression and thus to calculate the location set of the memory operand. If e.g. the value set of `esp` is $\{(stack\_frame_a \mapsto \{-20\})\}$ and the one of `ebx` is $\{(global\_memory\_region \mapsto \{0, 1\})\}$, then the location set of the operand `[esp+ebx*4+8]` is $\{(stack\_frame_a \mapsto \{-12, -8\})\}$. The heuristics are applied to these location sets and determine whether any locations are contained that are unlikely to be accessed by sound code.

Special treatment is however required in situations where a location set is $\top$, i.e. the value of all possible values. E.g. A-locs can have $\top$ as value set for several reasons. E.g. they might be uninitialised (cf. section 2.5.1) or they might contain the result of an operation for which no reasonable result exists, e.g. the multiplication of two pointer values. If $\top$ is the value set of a register and if that register is used either as base or as index register in a memory operand, the resulting location set of the memory operand is $\top$ as well. Hence, the operand might access every location and all applicable heuristics should detect a buffer overflow. In practice however, treating $\top$ this way would results in a large number of false positives. In general, considering operands that have $\top$ as location sets safe has proven to be a much better guess than considering them buffer overflows. Consequently, operands are entirely ignored if their location set is $\top$.

The VSA may potentially visit every location multiple times before a fixpoint is reached. Of course, later visits of the same location in the executable may change the value sets computed at earlier visits. Whenever a value set from an earlier visit of one location is replaced with a new one, the old value set is a subset of the newer one. Still, the heuristics are used not only based on the final results but based on all intermediate ones instead, i.e. after each analysis step of the VSA. The reason is that location sets that are not $\top$ in early intermediate results may become $\top$ later. Hence, buffer overflows may be detectable based on intermediate results but not on the final ones. In general, the increased precision of the buffer overflow detection justifies the larger time requirements caused by the more frequent use of the heuristics.

### 3.1.2 The return address heuristic

Each stack frame contains the return address, i.e. the address to which the control flow jumps after the function returns. The return address is pushed onto the stack when a function is called and popped from the stack when it returns. Other instructions usually neither read nor write the return address. According to this observation, the return address heuristic considers every memory access that is neither part of a call nor of a return and that still reads or writes the return address a buffer overflow. Due to the way function calls are handled, the

return address is always stored in the bytes at offsets four to seven in each stack memory region (cf. section 2.5.7). Based on the location set, one can therefore easily determine, whether or not a given memory operand potentially accesses a return address.

Figure 3.1 illustrates a situation in which the return address would detect a buffer overflow. The displayed memory access `[eax]` might access a range of addresses that overlaps the return address and thus is probably a buffer overflow.

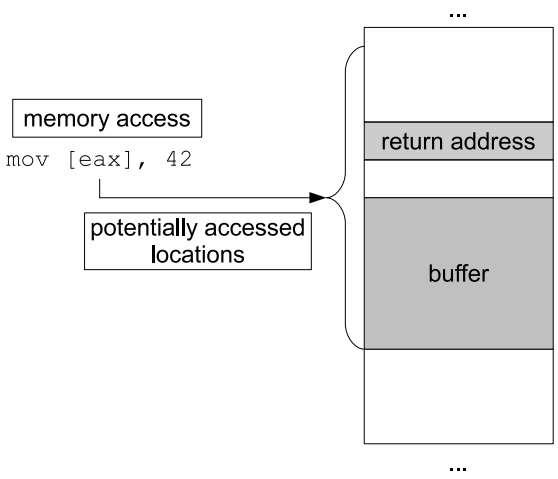



Figure 3.1: Illustration of the return address heuristic. A memory operand is considered a buffer overflow if it potentially overwrites a return address.

While code that reads or writes the return address usually is suspicious, there are code patterns frequently used by the compiler that safely read or write the return address. One example is code that aligns the stack, i.e. that makes the base address of a stack frame divisible by a certain constant in order to reduce the time required to load values from the stack. GCC version 4.2.3 for instance makes the start of the current stack frame divisible by 16 in the beginning of the main function. In order to do this, zero to 15 otherwise unused bytes are added to the stack. Afterwards, the return address is shifted to the new offset 4 relative to the now aligned base pointer. As in this process the original return address is read, the return address heuristic would issue a reading buffer overflow. In order to prevent this, it is possible to specify certain code patterns that are considered harmless and in which the return address heuristic is not applied.

### 3.1.3 The jump heuristic

Buffers are often accessed by loading the value of the index variable into a register and the base address of the buffer into another. Then a memory operand using the loaded registers and the size of one buffer element as scale is used for the actual buffer access. E.g. a buffer that contains 4 byte integer values might be accessed by loading its starting address into `eax`, the index value into `ebx` and then using the operand `[eax+ebx*4]`. In other cases, not only the base register but also the offset is used to specify the base address of the buffer. A

buffer that starts 20 bytes below the base pointer might for instance be accessed using the operand `[ebp-20 + ebx*4]` after loading the index value into `ebx`.

The jump heuristic is based on the assumption, that a memory operand that uses both base and index register is an access to a buffer and that the sum of the base register and the offset is the start address of the buffer or at least an address inside the buffer. In order to tell whether the memory access is a buffer overflow, assumptions about the size of the buffer have to be made. While more restrictive assumptions about the size of a buffer are hard to make, every buffer definitely occupies at most the space up to the next return address. Consequently, if the sum of base address and offset points to a location on the stack below a given return address, the memory operand should access a location below the return address as well. Analogously, the if the sum of base register and offset points to a location above the return address, the location accessed by the operand should be above the return address.

If any of these constraints is violated, i.e. if the sum of base register and offset is potentially not on the same side of the return address as the accessed location in the memory, a buffer overflow is issued by the jump heuristic. This situation is illustrated in figure 3.2.

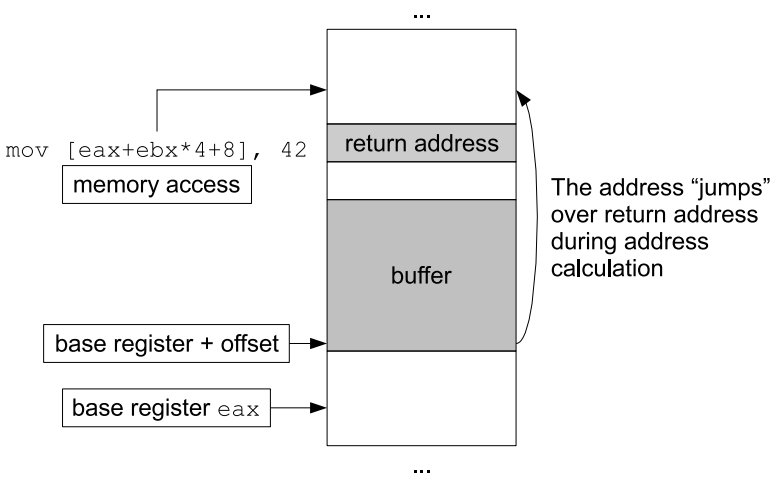



Figure 3.2: An illustration of the jump heuristic. The address that is accessed "jumps" over the return address when the scaled index register is added to the sum of base register and offset.

In some cases, the sum of base register and offset potentially overlaps the return address. As this contradicts the assumption that the sum of base register and offset points to a location inside a buffer, a buffer overflow is issued in this case as well.

### 3.1.4 The argument heuristic

The third heuristic is the argument heuristic which is based on the observation that large buffer overflows may result in a memory access inside a stack frame that does not belong to the current function. This situation is shown in figure 3.3. A buffer overflow is however not the only possible cause for such an

access to stack frame of another function. Another possible cause is that the function which the stack frame belongs to passed a pointer to one of its local variables as argument and that pointer is now used.

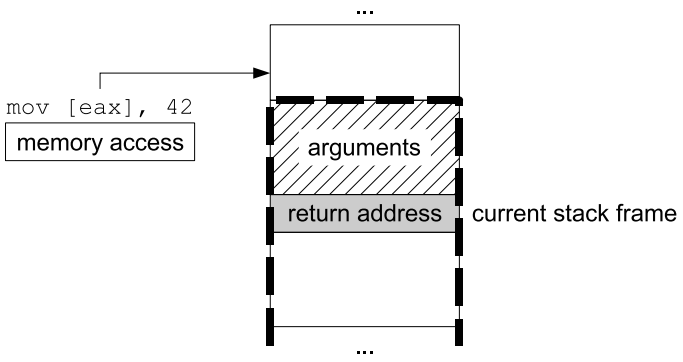



Figure 3.3: A memory operand accesses a location above the current function's stack frame.

Fortunately, these two causes are rather easy to distinguish. If the stack frame owner passes a pointer to one of its local variables, the memory region of the pointer value in the abstract memory model of the VSA will be the stack memory region of the stack frame owner. If in contrast the access to another function's stack frame is caused by a buffer overflow, then the accessed location will be one in the current function's stack memory region with an offset that is larger than the offset of the highest argument byte. Hence, an access in a memory region that uses a larger offset than the one of the function's highest argument byte is considered a buffer overflow by the argument heuristic.

Unfortunately, determining the exact number of bytes of arguments a function receives is difficult. Therefore, the argument heuristic is only applied inside functions that use calling conventions that allow to reliably determine the number of argument bytes.

The calling conventions are the rules by which arguments and return values are passed between functions. Among other things, the calling conventions differ in whether the caller or the callee is responsible for removing the arguments from the stack. Calling conventions in which the caller removes the arguments from the stack make it hard to determine the number of arguments passed to a function as the compiler does not necessarily release the memory used for the arguments immediately after the call. Instead, it might be reused for other purposes immediately. Also, the arguments are not necessarily pushed onto the stack immediately before the call.

Calling conventions in which the callee is responsible for removing the arguments from the stack in contrast allow to determine the number of argument bytes much easier. For such functions, the number of argument bytes is the difference of the stack pointer at the function entry from the one at the return. Also, the arguments are usually removed from the stack using a return instruction with an operand. Such a return instruction does not only remove the return address from the stack and jump to the specified location but also removes the number of bytes specified by the operand from the stack. Thus, in

functions where the callee removes the arguments from the stack, the number of argument bytes can often be determined by looking at the return instructions' operands.

Due to the lack of a reliable technique for detecting the number of argument bytes other than the one mentioned above, the argument heuristic is only applied inside functions that use calling conventions in which the callee is responsible for removing the arguments from the stack.

### 3.1.5 The index heuristic

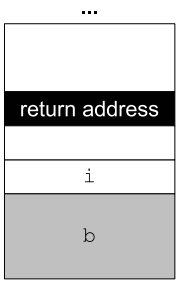
So far, all shown heuristics are only able to identify buffer overflows that reach at least up to the next return address. In many cases however smaller overflows can be exploited as well. The main challenge in detecting them is that no information about how the memory is partitioned into different variables and hence no information about buffer boundaries is available. Also, identifying the variable boundaries based on the use of the memory is very challenging as every variable's memory might be used for a different variable when the original one is not needed any more without any indication in the assembler code.

The index heuristic tries to identify buffer overflows that do not reach up to the next return address. For this purpose, the heuristic first tries to identify "index" variables using a simple algorithm based on reaching definitions analysis. Then, based on the assumption that a buffer access never overwrites an index variable used in the access, likely buffer overflows are identified.

**The heuristic** In many cases buffers are accessed with an expression that uses an index variable like `b[i] = 'A'`. As `b` and `i` are different variables, the assignment should leave `i` unchanged. If `i` is still changed, it is likely that `i` is located after the buffer `b` an that a buffer overflow occurred. This situation is illustrated in figure 3.4. The left side of the figure shows a simple C program with a buffer overflow. The loop in lines 3 to 6 of the program first overwrites all ten elements of the buffer and then continues to write four bytes past the buffer boundaries. The right side of the figure shows the stack layout of the C program. The index variable `i` is located immediately above the buffer. Consequently, the lowest byte of `i` is overwritten by the expression `b[i] = 'A'` in the eleventh iteration of the loop.

```
1  char b[10];
2  int i = 0;
3  while(i < 14){
4    b[i] = 'A';
5    i = i + 1;
6  }
```

(a) The program




(b) Its stack layout

Figure 3.4: A C program with a buffer overflow that overwrites the index variable `i`.

Another common way to access buffers is by using a pointer to a location inside the buffer, e.g. in an expression like `*p = 42`. Similar to the first example, this expression should usually not modify the value of the pointer variable `p`. What `i` and `p` have in common is that they are variables that are used for the calculation of the address at which the right-hand value of the assignment is stored.

The general assumption is that a buffer access that uses a pointer or index variable does not change that pointer or index variable. Similarly, it is assumed that the pointer or index variable is only read during address calculation but not by the buffer access itself. Thus, a memory access is considered a buffer overflow, if it potentially points to a variable used for its address calculation. This situation is illustrated in figure 3.5: The shown memory access may not only point to locations inside the buffer but also to the index variable. Consequently, the index heuristic would report a buffer overflow.

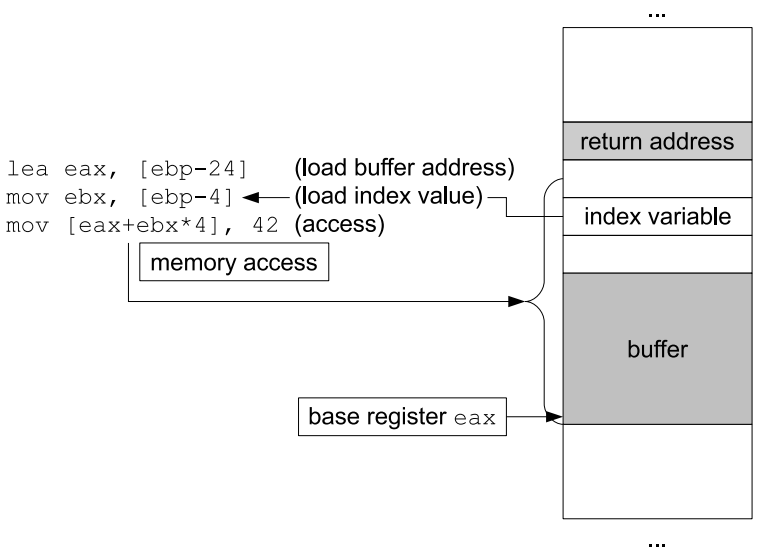



Figure 3.5: The memory access potentially overwrites the value of an index variable used for the address calculation of the memory access. This would be considered a buffer overflow by the index heuristic.

In order to be able to apply the index heuristic, a technique allowing to identify all locations from which values that are incorporated in a certain address calculation is required.

**Identification of "index" variables** A necessary prerequisite for the application of the index heuristic is the ability to identify the variables used for calculating the address of a given memory operand. For this purpose, an approach based on [CSF98] is used.

Statements like `b[i] = 'A'` are usually translated into multiple instructions during compilation. A first instruction might for instance load the address at which the buffer `b` begins, a second one might load the value of `i` into a register. Then the third instruction could access the buffer. An example of such

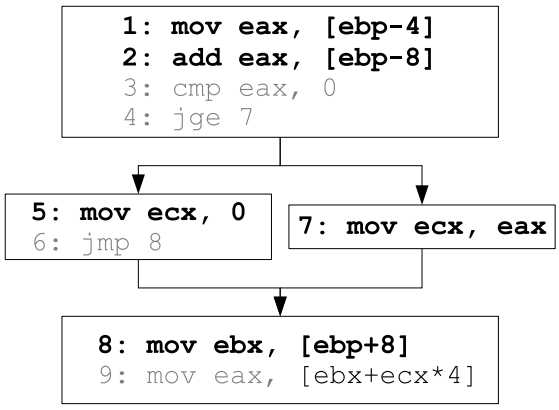
a sequence of instructions is displayed in the left part of figure 3.5.

Based on the observation that the address of a buffer access is usually computed in multiple instructions, the index heuristic first tries to identify the instructions used for calculating the address of a given memory address using the reaching definitions analysis (cf. section 2.4). Then, all variables used in those instructions are considered "index variables".

The address of a memory access is computed using base register, index register, scale and offset (cf. section 2.2.2). Thus, as scale and offset are fixed, the variables used for calculating the address of a memory access are the variables used for calculating the values of base and index register. For every memory operand, the index heuristic uses the results of the reaching definitions analysis in order to determine which instructions potentially defined base and index register of the operand. Then, these instructions are examined. If they use the values of memory operands, these memory operands are variables used for calculating the address of the memory access in question. If the instructions use registers, the locations where these registers were potentially defined are part of the calculation of the memory address in question and thus are examined as well. Again, if they use memory operands, these operands are added to the set of variables used for address calculation. If they use registers, the positions where they were defined are examined. This process is repeated recursively until no more instructions used for address calculation are found.

Figure 3.6 again shows the example from figure 2.5 which was used to illustrate the reaching definitions analysis algorithm. Subfigure (a) shows a short program and subfigure (b) shows the results of the reaching definitions analysis. Compared to figure 2.5, the column containing the defined operands of each instruction has been removed. Instead, the last column indicates which register and memory operands used by the instruction.

The memory operand `[ebx+ecx*4]` of instruction 9 uses both a base and an index register. Thus, in order to identify the variables used for address calculation, the potential defining positions of these registers must be examined. The set of reaching definitions of the base register `ebx` is $\{8\}$ . Thus, instruction 8 is examined. It is a `mov` instruction that assigns the value of the second operand to the first operand. Thus, instruction 8 only uses the value of the second and not the value of the first operand. The second operand is `[ebp+8]`. As the second operand is a memory operand, it is added to the set of variables used for address calculation. As now the index heuristic has completed examining the reaching definitions of the base register `ebx`, it is time to examine the reaching definitions of the index register `ecx`. `ecx` has two reaching definitions: instruction 5 and 7. Instruction 5 is a `mov` instruction again and thus only uses the second operand. This operand however is the immediate value `0` and thus not a variable used for address calculation. Instruction 7 is a `mov` instruction as well and uses the register operand `eax`. Thus, the index heuristic continues to examine the definitions of `eax` that reach instruction 7, i.e. instruction 2. Instruction 2 is an `add` instruction which stores the sum of its two operands in the first one. Thus, both operands are used. The second operand `[ebp-8]` is a memory operand and is thus also added to the set of variables used for calculating the address of the memory operand of instruction 9. The first operand of the `add` instruction in line 2 is the register operand `eax`. Thus, the index heuristic also examines the reaching definitions of `eax` at instruction 2 and thus continues at instruction 1. Instruction 1 only uses its second operand. This operand is




(a)

| Instruction | Reaching definitions | | | | Used |
|---|---|---|---|---|---|
| | Position | eax | ebx | ecx | |
| 1 | before | $\emptyset$ | $\emptyset$ | $\emptyset$ | [ebp-4] |
| | after | {1} | $\emptyset$ | $\emptyset$ | |
| 2 | before | {1} | $\emptyset$ | $\emptyset$ | eax, [ebp-8] |
| | after | {2} | $\emptyset$ | $\emptyset$ | |
| 3 | before | {2} | $\emptyset$ | $\emptyset$ | eax |
| | after | {2} | $\emptyset$ | $\emptyset$ | |
| 4 | before | {2} | $\emptyset$ | $\emptyset$ | - |
| | after | {2} | $\emptyset$ | $\emptyset$ | |
| 5 | before | {2} | $\emptyset$ | $\emptyset$ | - |
| | after | {2} | $\emptyset$ | {5} | |
| 6 | before | {2} | $\emptyset$ | {5} | - |
| | after | {2} | $\emptyset$ | {5} | |
| 7 | before | {2} | $\emptyset$ | $\emptyset$ | eax |
| | after | {2} | $\emptyset$ | {7} | |
| 8 | before | {2} | $\emptyset$ | {5, 7} | [ebp+8] |
| | after | {2} | {8} | {5, 7} | |
| 9 | before | {2} | {8} | {5, 7} | [ebx+ecx*4] |
| | after | {9} | {8} | {5, 7} | |

(b)

Figure 3.6: An example program (a) and the results of the reaching definitions analysis (b). The operands used for calculating the address of the memory operand of instruction 9 are highlighted.

the memory operand `[ebp-4]` which is also added to the set variables used for address calculation. As no instructions are left, the identification of variables used for address calculation is completed. The instructions that calculate the address are 1, 2, 5, 7 and 8. The variables used for address calculation are `[ebp-8]`, `[ebp-4]` and `[ebp+8]`. Thus, the index heuristic reports a buffer overflow, if the location set of `[ebx+ecx*4]` overlaps any of the locations sets of `[ebp-8]`, `[ebp-4]` and `[ebp+8]`. Of course, the location sets of the variables used for address calculation have to be computed based on the VSA results at the locations where the variables are used. I.e. the location set of `[ebp-4]` has to be computed based on the value sets before instruction 1, the location set of `[ebp-8]` based on the value sets before instruction 2 and the one of `[ebp+8]` based on the value sets before instruction 8.

## 3.2 VSA additions

In addition to the development of several heuristics that try to identify buffer overflows, modifications where made to the VSA. Most notably, the widening technique was modified to favour "good" widening points (cf. section 3.2.1). Furthermore, a simple approach for the improvement of the analysis of conditional jumps was added (cf. section 3.2.2).

### 3.2.1 Delayed widening

The algorithms used in the VSA try to minimize the total number of widening points but have no preferences concerning the position of the widening point. The position of the widening point inside the CFG may however greatly affect the precision of the VSA and the buffer overflow detection. This is shown in section 3.2.1.1.

The approach splits into two parts. First, a heuristic that tries to identify good widening points is described in section 3.2.1.2. The second part of the approach is a technique that tries to perform widening at the good widening points rather than the ones chosen by the widening point selection and is described in section 3.2.1.3. The approach works independently of the heuristic that chooses good widening points and could be used with another heuristic as well.

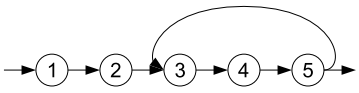
The last two subsections discuss the additional cost caused by the delayed widening. First, the maximum number of additional widening processes is discussed in section 3.2.1.4 and secondly the memory requirements are discussed in section 3.2.1.5. The additional time required for delayed widening was examined experimentally only (cf. section 4.1.6 and section 4.2.4).

#### 3.2.1.1 Good and bad widening points

The VSA uses widening to reduce the time required for the analysis of loops (cf. section 2.5.3). Before analysis, a widening point selection algorithm places widening points inside the CFG such that at least one is in every loop. Then, the value sets are widened whenever the analysis reaches a widening point for the second time. Every widening process reduces the precision of the results. Thus, the widening point selection algorithm should keep the number of widening

```
1  char a[10];
2  int i = 0;
3  while(i < 10){
4     a[i] = '\0';
5     i++;
6  }
```

(a) The program




(b) Its CFG

| Step | Position | | Value set of `i` Widening point in line 5 | Widening point in line 4 |
|---|---|---|---|---|
| 1 | 1 | before | $\top$ | $\top$ |
| | | after | $\top$ | $\top$ |
| 2 | 2 | before | $\top$ | $\top$ |
| | | after | $\{0\}$ | $\{0\}$ |
| 3 | 3 | before | $\{0\}$ | $\{0\}$ |
| | | after | $\{0\}$ | $\{0\}$ |
| 4 | 4 | before | $\{0\}$ | $\{0\}$ |
| | | after | $\{0\}$ | $\{0\}$ |
| 5 | 5 | before | $\{0\}$ | $\{0\}$ |
| | | after | $\{1\}$ | $\{1\}$ |
| 6 | 3 | before | $\{0,1\}$ (union) | $\{0,1\}$ (union) |
| | | after | $\{0,1\}$ | $\{0,1\}$ |
| 7 | 4 | before | $\{0,1\}$ | $\{0,1,\ldots,\infty\}$ (widened) |
| | | after | $\{0,1\}$ | $\{0,1,\ldots,\infty\}$ |
| 8 | 5 | before | $\{0,1,\ldots,\infty\}$ (widened) | $\{0,1,\ldots,\infty\}$ |
| | | after | $\{1,2,\ldots,\infty\}$ | $\{1,2,\ldots,\infty\}$ |
| 9 | 3 | before | $\{0,1,\ldots,\infty\}$ (union) | $\{0,1,\ldots,\infty\}$ (union) |
| | | after | depends on successor | depends on successor |
| 10 | 4 | before | $\{0,1,\ldots,9\}$ | $\{0,1,\ldots,\infty\}$ (widened) |
| | | after | $\{0,1,\ldots,9\}$ | *No changes anymore* |
| 11 | 5 | before | $\{0,1,\ldots,\infty\}$ (widened) | |
| | | | *No changes anymore* | |

(c) VSA results

Figure 3.7: A simple loop that fills a buffer with zeros. If the widening point is chosen in a disadvantageous position, a buffer overflow might erroneously be detected in line 4.

points low. However, not only the number of widening processes affects the precision of the results but also the exact location of the widening points in the CFG. This is shown using figure 3.7.

Figure 3.7 shows a short program that consists only of one loop which fills a buffer with zeros. For simplicity, the C code of the program is used instead of the assembler code. Whether or not a buffer overflow is erroneously reported in line 4 depends of the choice of the widening point. The table in the figure shows how the choice of the widening point affects the results of the VSA.

First, it is assumed that the WP selection algorithm placed the WP in line 5. Initially, the value set of `i` is $\top$ . The variable initialization in line 2 changes the value set to $\{0\}$ . During the first analysis iteration of the loop body in steps 3 to 5, widening is not performed yet. The value set for `i` at the end of the loop body after step 5 is $\{1\}$ . When the value set is propagated along the CFG edge from line 5 back to line 3, it has to be merged with the memory set from line 2. Hence, in the beginning of the second analysis iteration before step 6, the value set is $\{0,1\}$ . The widening point in line 5 is reached again in step 8 and widening is performed. Thus, the value set of `i` becomes $\{0,1,\ldots,\infty\}$. When the loop condition is reached again in step 9, the value set is split, based

on the loop condition. The value set for `i` on the edge that stays inside the loop is $\{0, 1, \ldots, 9\}$ and the one on the edge that leaves the loop is $\{10, 11, \ldots, \infty\}$. In step 10, the buffer access is analysed for the third time. The value set of the index variable is $\{0, 1, \ldots, 9\}$ and hence no buffer overflow is detected. When in step 11 the widening point is reached for the third time, there are no changes compared to the previous visit any more. Thus, the VSA does not trace the current CFG path any further. Instead, the analysis would proceed after the loop if there was more code to analyse.

In the second scenario, the widening point is placed in line 4 instead of in line 5. This does not affect the first six analysis steps. Also, steps 8 to 10 are similar to the last steps in the first scenario. The crucial analysis step is step 7. As the widening point in line 4 is reached the second time in that step, widening is performed. Consequently, the value set of `i` already becomes $\{0, 1, \ldots, \infty\}$ in step 7 immediately before the buffer access in line 4. As the value set of `i` before the buffer access is $\{0, 1, \ldots, \infty\}$ this time, a buffer overflow should be reported. Hence, placing the widening point at line 5 instead of line 4 makes the difference between a correct analysis result and a false positive.

The difference between the first, good widening point and the second, bad one is the order in which widening point, loop condition and buffer access are analysed. Using the first widening point, the analysis reaches the loop condition between the widening point and the buffer access. This makes it possible to restrict the widened value set of `i` before the buffer access is analysed. If in contrast the loop condition is not visited between widening point and buffer access, the widened value set will reach the buffer access which usually leads to a false positive.

A way to make sure that the loop condition is visited between widening and the next buffer access would be to perform widening immediately before the loop condition.

#### 3.2.1.2 Identification of good widening points

This section describes a heuristic that tries to identify good widening points. The heuristic is based on the observation that the location immediately before a loop condition is a good widening point.

Loops that subsequently overwrite multiple elements of a buffer like the loop in the program in figure 3.7 are a common programming idiom. As the example in section 3.2.1.1 illustrates, false positives are likely to occur in such loops, if the loop condition is not located between the position where widening is performed and the next buffer access. A way to guarantee that the loop condition is always located between the position where widening is performed and the next buffer access would be to perform widening immediately before the loop condition. Consequently, loop conditions are considered good widening points.

In order to identify loop conditions, the heuristic needs to differentiate between two different types of conditional jumps in loops: ones branching into alternate paths inside the loop like instruction 3 in figure 3.8 and loop conditions like instruction 7 in that figure. While after instruction 3 both paths stay inside the loop, instruction 7 branches the control flow into one path that leaves the loop. Thus, in order to differentiate an intra-loop branch like the one in instruction 3 from a loop condition like the one in instruction 7, the good widening point heuristic has to estimate which instructions belong to the same

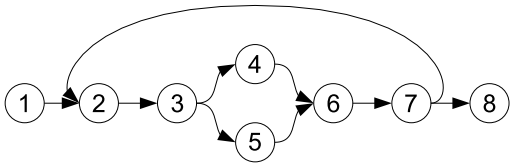



Figure 3.8: The CFG of a loop and two alternating paths inside the loop.

loop.

As described in section 2.5.4, hierarchical decomposition [Bou93] computes a weak topological ordering $WTO$ that usually closely resembles the loop structure of the program. More precisely, the components of the WTO resemble the loops of the program. Thus, if an edge in the CFG leaves a loop, it usually leaves a component in the WTO as well and vice versa. E.g. the WTO computed for the CFG in figure 3.8 by the hierarchical decomposition would be 1(234567)8. In this WTO, the edges $(3, 4)$ and $(3, 5)$ do not leave any component. The edge $(7, 8)$ in contrast does leave the component 234567 which usually is a strong indication that it leaves a loop in the CFG as well.

Based on the observation that the components in a WTO computed by the hierarchical decomposition usually resemble the loops in the CFG rather well, the good widening point heuristic considers every conditional jump that has both an outgoing edge that leaves a component and one that stays inside the component to be a loop condition and thus a good widening point.

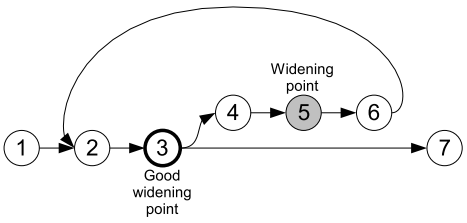
#### 3.2.1.3 Delaying widening

The previous section described a heuristic that tries to identify good widening points. In this section, an approach that tries to perform widening at those good widening points is introduced. The approach could also be used with other heuristics for the identification of good widening points.

The basic idea of delayed widening is not to modify the widening point selection algorithm to prefer good widening points but instead to modify the way widening is performed. When a widening point is visited for the second time, widening is not performed immediately anymore. Instead, the loop is analysed one more time. If during this additional analysis iteration a good widening point is found, widening is performed immediately at that point. If no good widening point is found, widening is performed when the widening point is reached again after the additional analysis iteration.

In order to ensure that widening is performed at least once per loop but still not unnecessarily often, the "widening requested" sets ($WRS$) are introduced. When a given widening point is visited for the second time, i.e. when usually widening would be performed, that widening point is added to the WRS. This indicates that widening should be performed at the next good widening point. Figure 3.9 shows an example CFG with a good widening point and one that is assumed to have been chosen by the widening point selection algorithm. When the original widening point at instruction 5 is visited for the second time it is added to the WRS.

The WRS is propagated along the CFG path. Thus, the widening point is still in the WRS at instruction 6. When several CFG paths meet, the new WRS




| Instruction | WRS at fixpoint |
|---|---|
| 1 | $\emptyset$ |
| 2 | $\{5\}$ |
| 3 | $\emptyset$ |
| 4 | $\emptyset$ |
| 5 | $\{5\}$ |
| 6 | $\{5\}$ |
| 7 | $\emptyset$ |

Figure 3.9: A CFG containing a one widening point ($WP$) chosen by the widening point selection algorithm and one that is considered a good widening point by the good widening point heuristic. The table on the right side shows the widening requested sets ($WRS$) at all positions in the CFG.

is is the union of the WRS of all paths that meet. Consequently, the WRS at instruction 2 is $\{5\}$ at instruction 2 in the example.

What happens when a good widening point in the CFG is reached depends on whether the WRS is empty. If the WRS is empty, nothing special is done. If however the WRS is not empty, widening is performed and WRS is set to $\emptyset$. The fact that the WRS is now empty shows that widening already has been performed. In the example, the WRS is not empty when the good widening point is reached in instruction 3 and thus widening is performed.

The WRS serves several purposes. First, it ensures that widening is only performed if it was performed in the original approach as well due to the fact that the WRS can only be non-empty if a widening point was analysed at least twice before. Second, it ensures that in a loop with multiple good widening points widening is only performed once. If e.g. instruction 4 in the example was a good widening point as well, widening would still not be performed again as the WRS is empty at that instruction. The third purpose of the WRS is to indicate whether or not a good widening point was found during the additional analysis iteration. If no good widening point is found, the widening point will still be in the WPS after the additional analysis iteration. Thus, in order to ensure that widening is performed at least once per cycle in the CFG, widening is performed whenever a widening point is reached with a WPS containing that widening point.

Before the heuristic that tries to identify good widening points is discussed in the next section, the correctness of the approach is proven:

**Theorem 3.1.** *Delayed widening performs widening at least once in every cycle in the CFG.*

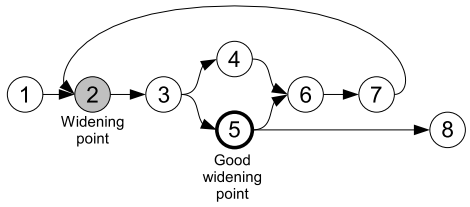
*Proof.* Let C be an arbitrary cycle in the CFG. As the widening point selection algorithm places at least one widening point in every cycle in the CFG, C contains at least one widening point. After a sufficient number of analysis iterations, the widening point will be added to the WRS. While tracing C, the widening point will never be removed from the WRS unless widening is performed at a good widening point. Thus, if widening is not performed in C before the widening point is reached again, the widening point will still be in the WRS. This however means that widening will be performed at the widening

point. Thus, widening is either performed at a good widening point inside C or after a further analysis iteration when the widening point is visited again. □

#### 3.2.1.4 Number of additional widening processes

An interesting aspect of delayed widening is its influence on the number of widening processes. Especially, an interesting question is the following: In which situations additional widening processes are cause by delayed widening and how many additional widening processes are potentially caused?

Unfortunately, there is no upper bound to the number of additional times widening that is performed. This can be illustrated using the CFG in figure 3.10a. The figure shows a modified version of the CFG in figure 3.8. In the modified version, the loop condition is instruction 5 instead of instruction 8. A valid WTO of the CFG is 1(234567)8. Based on this WTO, instruction 2 is chosen as widening point and instruction 5 is considered a good widening point.




(a) CFG

| Instruction | WRS at fixpoint |
|---|---|
| 1 | ∅ |
| 2 | {2} |
| 3 | {2} |
| 4 | {2} |
| 5 | ∅ |
| 6 | {2} |
| 7 | {2} |
| 8 | ∅ |

(b) Widening requested sets

Figure 3.10: A modified version of the CFG in figure 3.8. Widening is performed twice in one loop if delayed widening is used.

When instruction 2 is visited for the second time by the VSA, it is added to the WRS. Thus, when the good widening point at instruction 5 is analysed, the WRS is not empty, widening is performed and the WRS is set to ∅. Immediately before instruction 6 however, the current CFG path meets the one from instruction 4. As no widening was performed there, the WRS at instruction 4 is still {2}. The WRS at instruction 6 is the union of the ones after instruction 4 and 5 and thus again {2}. As no further good widening point is reached before instruction 2, the WRS is still {2} when instruction 2 is reached the next time and widening is performed one more time. Thus, widening is performed twice using delayed widening despite the fact that there is only one widening point in the loop. Also, repeating the pattern consisting of instructions 3, 4, 5 and 6 leads to further unnecessary widening processes This is illustrated in figure 3.11. The loop in the CFG shown in that figure repeats the pattern five times and widening is as a result performed six times in the loop.

The property of the CFG that leads to the surplus widening processes is that the loop contains loop conditions in optional branches. In the usual loops used in high level languages, the loop condition is however evaluated every iteration and thus leads not to additional widening processes.

While there is in theory no upper bound for the number of additional positions where widening is execute, only a moderate increase can be observed in

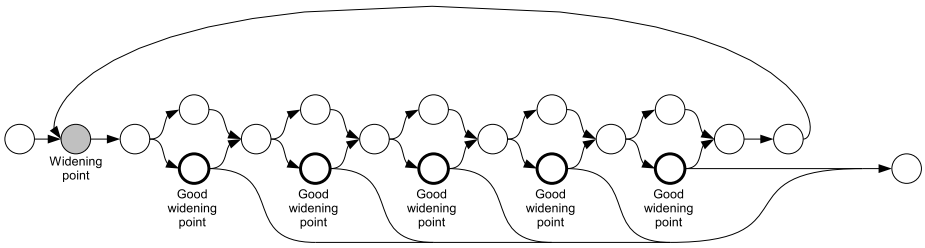



Figure 3.11: A CFG in which widening is performed six times in one loop if delayed widening is used.

practice (cf. section 4.2.4).

#### 3.2.1.5 Memory usage

Another interesting aspect of delayed widening is the additional memory required for the WRS. In practice, two WRS are stored per instruction. Like the abstract environments, one of the WRS represents the situation before and one the situation after the instruction.

If delayed widening was performed as described so far, the WRS might become very large and even contain all widening points of the entire program. Thus, the additional memory usage would be in $O(\#instructions \times \#widening\_points)$. The memory usage can however significantly reduced.

As described in section 2.5.4, the widening point selection algorithm first computes a weak topological ordering ($WTO$) of the program and then chooses all component heads as widening points. Intuitively, each WTO component corresponds to a loop in the CFG and the widening point that is the head of the component is responsible for performing widening inside the loop. Consequently, a widening point is only needed inside the WRS inside its component. This is first illustrated using an example and then be proven.

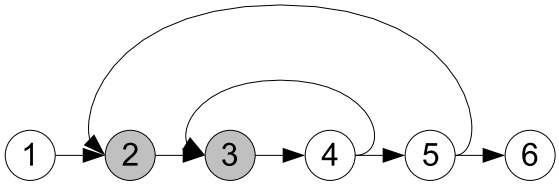



Figure 3.12: A simple CFG containing two nested loops.

Figure 3.12 shows a simple CFG containing two loops. A WTO of the CFG is 1(2(34)5)6. Based on this WTO, instructions 2 and 3 would be chosen as widening points. The widening point at instruction 3 can safely be erased from the WRS outside its component consisting of instructions 3 and 4, i.e. instruction 3 can be erased when the WRS is propagated along the edge $(4, 5)$. Analogously, it is safe to erase widening point 2 from the WRS that is propagated along the edge $(5, 6)$.

In general, every widening point can in every CFG safely be removed from

every WRS outside its component. The following lemma is required for the proof.

**Lemma 3.1.** *Let $H$ be the head of a component in a given weak topological ordering $W$ of a given CFG. Let $C$ be circle in the CFG that contains $H$ but does not entirely lie inside the component of $H$. Then*

1. *there is a feedback edge $(F, H_2)$ in $C$ such that the component of $H$ lies entirely in the component of $H_2$ in $W$ and $H \neq H_2$.*
2. *there is a feedback edge $(F_o, H_o)$ in $C$ such that $C$ lies entirely in the component of $H_o$.*

*Proof.* Figure 3.13 shows the situation described in the lemma. First, the part 1 of the lemma is proven.

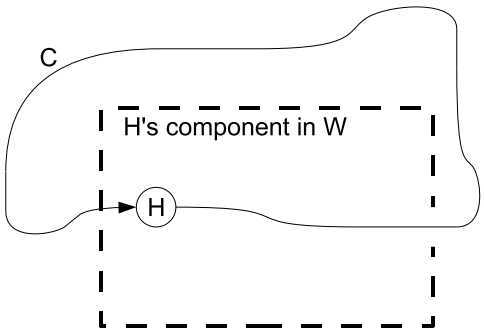
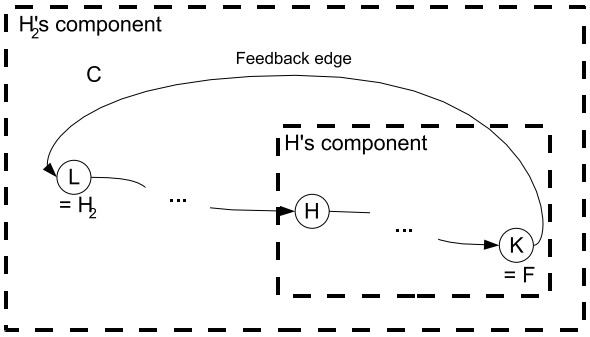


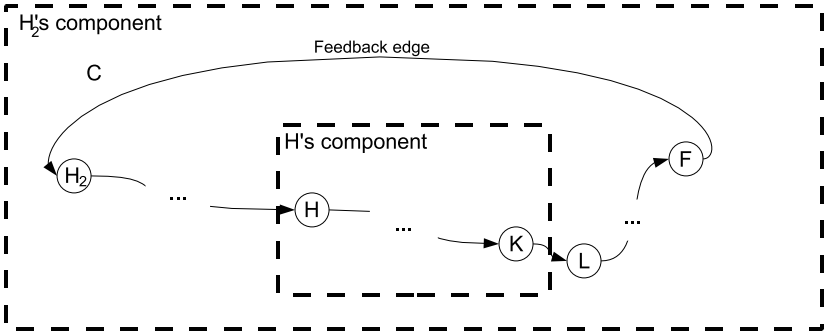

Figure 3.13: The situation described in lemma 3.1.

As $C$ does not lie entirely in component of $H$, there is at least one edge in $C$ that leaves the component of $H$. Let $(K, L)$ be the first edge after $H$ in $C$ that leaves the component of $H$. There are two situations:

- $L$ lies left of the component of $H$. Then $(K, L)$ points leftwards in $W$ and thus *is* a feedback edge. This situation is illustrated in figure 3.14a. Due to the fact that $(K, L)$ leaves the component of $H$, $K$ can not be equal to $H$. By definition of a WTO, $L$ is head of a component and $K$ is member of the component of $L$. That the component of $K$ starts immediately left of $K$ and reaches at least up to $L$. As however each WTO is well-parenthesized and the component of $K$ starts before the component of $H$, the component of $K$ ends after the component of $H$ and thus entirely contains the component of $H$. Thus, choosing $F = K$ and $H_2 = L$ meets all requirements from lemma 3.1.1.
- $L$ lies right of the component of $H$. This situation is illustrated in figure 3.14b. An important observation is that no edge can jump from a location right of the component of $H$ directly back to $H$ as such an edge would be a feedback edge and feedback edges to $H$ must always originate inside the component of $H$. Thus, there must be an edge $(F, H_2)$ in $C$ that jumps from a location right of the component of $H$ to a location left of the component of $H$. This edge of course jumps leftwards and thus is a feedback edge. Also, due to the definition of WTOs, $H_2$ is head of a




(a) C leaves the component of H to the left




(b) C leaves the component of H to the right

Figure 3.14: Two different ways in which the cycle C can leave the component of $H$.

component and $F$ is a member of the component of $H_2$. Thus, the area between $H_2$ and $F$ in which the component of $H$ lies is entirely part of the component of $H_2$ and all requirements from lemma 3.1.1 are met.

Now lemma 3.1.2 will be proven using lemma 3.1.1. As stated in lemma 3.1.1, there is a feedback edge $(F, H_2)$ in $C$ such that the component of $H$ lies entirely in the component of $H_2$. Now there are again two possibilities:

- $C$ lies entirely in the component of $H_2$. Then $H_o$ simply is $H_2$.
- $C$ does not lie entirely in the component of $H_2$. In this case, again applying lemma 3.1.1 yields that there is a feedback edge $(F_2, H_3)$ in $C$ such that the component of $H_2$ lies entirely in component the component of $H_3$. Then, either $C$ lies entirely in the component of $H_3$ or lemma 3.1.1 can be applied again and there is another feedback edge $(F_3, H_4)$ such that the component of $H_3$ lies entirely in the component of $H_4$. Analogously, lemma 3.1.1 can repeatedly be applied until eventually a node $H_o$ is found such that $C$ lies entirely in the component of $H_o$.

□

Lemma 3.1 can now be used to prove that widening is performed at least once in every CFG cycle even if widening points are removed from all WRS outside their components.

**Theorem 3.2.** *Let $W$ be a WTO of a given CFG and every head in $W$ be chosen as widening point. Delayed widening guarantees that widening is performed at least once in every CFG cycle even if every widening point is removed from every WRS outside the widening point's component in $W$.*

*Proof.* Let $C$ be an arbitrary cycle in the CFG. Then, there will be at least one widening point inside $C$. Let $H$ be a widening point in $C$. Lemma 3.1.2 immediately implies that there is a component head $P$ inside $C$ such that $C$ lies entirely inside the component of $P$. If $C$ already lies entirely in the component of $H$, then $P$ equals $H$. If $C$ does not lie entirely inside the component of $H$, then according to lemma 3.1.2, there is a feedback edge $(F, H_o)$ such that $C$ lies entirely in the component of $H_o$. In this case, $P$ equals $H_o$. Either way, there is a component head and thus a widening point $P$ in $C$ such that $C$ lies entirely in the component of $P$. Thus, the only reason due to which $P$ is possibly removed from the WRS inside $C$ is that widening is performed. Hence, analogously to the proof of theorem 3.1, widening is definitely performed once inside $C$. □

Using theorem 3.2, the additional memory required by delayed widening can be reduced to $O(\#instructions \times maximum\_nested\_WTO\_components)$. This is in the worst case equal to the original requirement of $O(\#instructions \times \#widening\_points)$ but in realistic scenarios much lower due to the fact that the maximum number of nested WTO components roughly corresponds to the number of nested loops in high level language source code.

The proof of theorem 3.2 is based on the assumption that all widening points were selected as heads of a WTO component. In practice however, the WTO-based approach is only applied intraprocedurally (cf. section 2.5.4). In order to solve this problem, delayed widening is only performed for widening points selected by the interprocedural widening point selection algorithm. Widening points chosen by the interprocedural algorithm are not added to the WRS but instead widening is performed immediately when such a widening point is visited the second time. Also, due to the fact that WTOs are computed intraprocedurally, the heuristic for finding good widening points does not really make sense for interprocedural widening points.

### 3.2.2 Improved conditional jump analysis

As described in section 2.5.2, conditional jumps can be used to restrict the value sets of the operands compared for the jump condition. It is however in many cases also possible to restrict other value sets than the ones compared for the conditional jump. A situation where this is the case is shown in figure 3.15.

The code shown in figure 3.15 corresponds to a high level expression that compares two variables like e.g. `if(a ≥ b)` where `[ebp-4]` corresponds to `a` and `[ebp-8]` corresponds to `b`. Instruction 2 and 3 form the usual pattern of a compare instruction and then a conditional jump. As the compare instruction however can use at most one memory operand, the value of the variable `[ebp-4]` is loaded into `eax` before the compare instruction. As a result, the value sets of `[ebp-8]` and `eax` instead of those of the two variables are restricted after the conditional jump. The full value set of `[ebp-4]` is propagated along both outgoing CFG edges after the conditional jump. In the high level language example, this means that the expression `if(a ≥ b)` is used to restrict the value

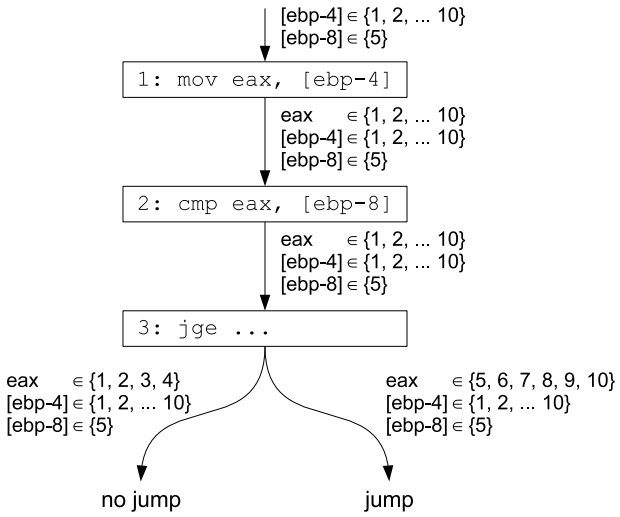



Figure 3.15: A conditional jump and the value sets of the used operands.

set of `b` but not the one of `a`. If `a` is used as index variable at a later buffer access, this may lead to a false positive.

As the high level language expression `if(a ≥ b)` suggests, the conditional jump could also be used to restrict `[ebp-4]` due to the fact that the value of `eax` is always equal to `[ebp-4]` at the jump. The conditional jump is only performed if `eax` is greater or equal to 5. As `[ebp-4]` is always equal to `eax`, `eax` can only be greater or equal to 5 if the `[ebp-4]` is greater or equal to 5 as well. Hence, the value set of `[ebp-4]` can be restricted in the same way as the value set of `eax`. In general, if the value of a conditional jump operand being compared always equals the value of another a-loc, restrictions can be applied not only to the value set of the compared operand but to both operands. Thus an algorithm is used to keep track which registers are equal to which variables.

The algorithm traces the CFG and updates which register is equal to which variable based on the following observations.

- Whenever a `mov` instruction stores the value of a variable in a register, the value of that register will be equal to the one of the variable until either the value of the register or the value of the variable is changed.
- Whenever the value of a register that is equal to a variable is copied to another register, that register will equal the variable as well afterwards.
- There two possibilities at locations where several CFG paths meet. If a register equals the variable at all meeting CFG paths, then it still equals that variable after the CFG paths met. Otherwise the register is not guaranteed to equal any variable after the CFG paths met.
- If a register equals a variable and either the register or the variable is modified, then they are not guaranteed to be equal anymore afterwards.

Figure 3.16 illustrates the algorithm. The figure shows an example program consisting of eleven instructions. The first instruction stores the value of

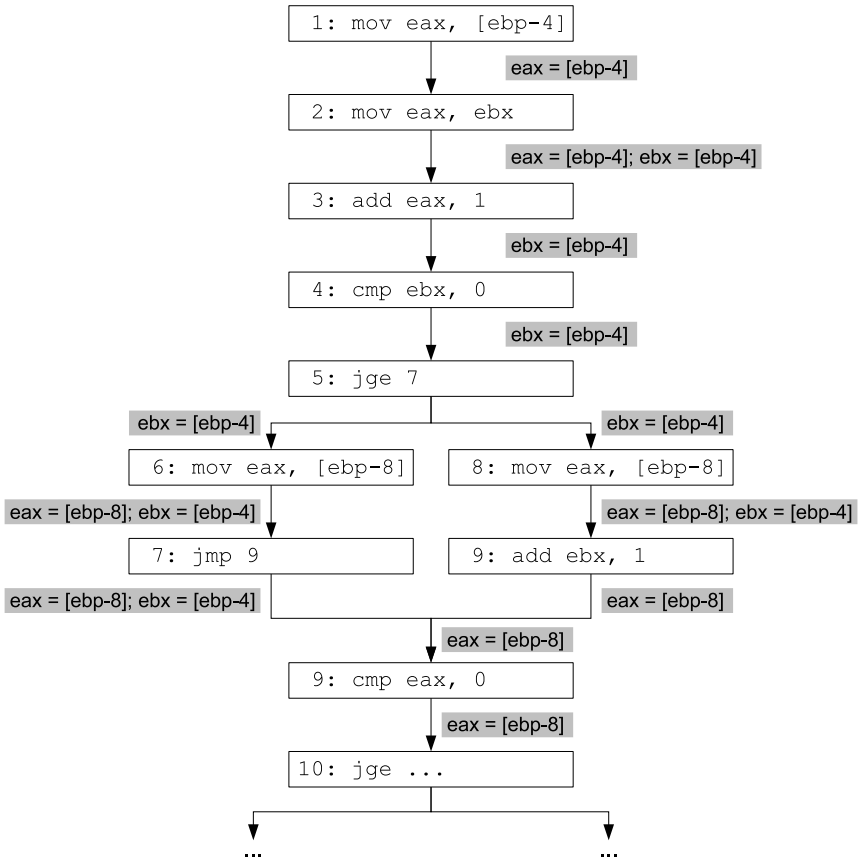



Figure 3.16: A short program with equality information of registers and variables displayed.

`[ebp-4]` in `eax`. Thus, `eax` is equal to `[ebp-4]` afterwards. Instruction 2 copies the value of `eax` to `ebx` and consequently `ebx` is equal to `[ebp-4]` afterwards as well. Instruction 3 modifies `eax` which as a result is not equal to `[ebp-4]` anymore afterwards. Instruction 4 and 5 form a conditional jump. A jump is performed if `ebx` is greater than or equal to 0. As at this point the value of `ebx` is under all circumstances equal to `[ebp-4]`, the VSA can not only restrict `ebx`'s value set but also the one of `[ebp-4]`. After the conditional jump, the control flow splits into two alternating paths. The left control flow path stores the value of `[ebp-8]` in `eax` and then jumps to instruction 10. The right control flow path also stores the value of `[ebp-8]` in `eax` in instruction 8. Instruction 9 modifies `ebx` which consequently does not equal `[ebp-4]` anymore afterwards. Before instruction 10, the two control flow paths meet again. `eax` is equal to `[ebp-8]` at the end of both branches and thus still before instruction 10. Only after the left control flow path is `ebx` guaranteed to be equal to `[ebp-4]` and thus not guaranteed to be equal to `[ebp-4]` before instruction 10. Instruction 9 and 10 again are a compare and a conditional jump instruction. Again, as `eax` is equal to `[ebp-8]` in any case, the VSA can not only restrict the value set of `eax`, which is used by the compare instruction, but also the one of `[ebp-8]`.

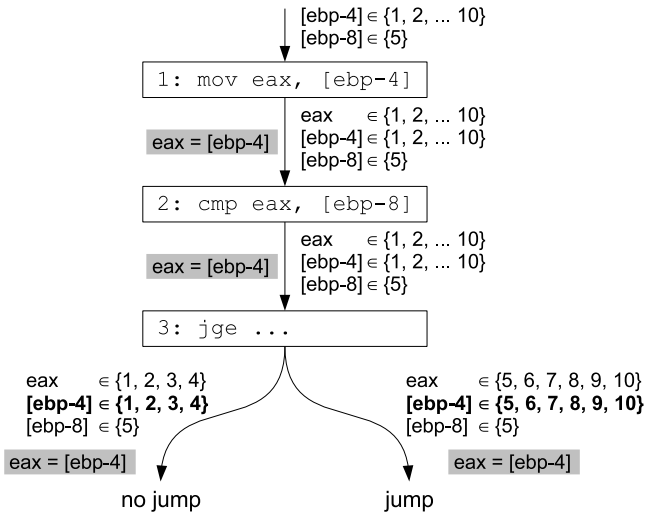



Figure 3.17: The VSA results of analysing the example of figure 3.15 with the improved conditional jump analysis.

Figure 3.17 shows the results of analysing the program of figure 3.15 with the improved conditional jump analysis. After the first instruction, `eax` equals `[ebp-4]` Using this information, the VSA can not only restrict the value set of `eax` at the conditional jump but also the one `[ebp-4]`.

# Chapter 4

# Evaluation

The performance of the buffer overflow detection approach was evaluated both using a set of synthetic test cases and using a real world example. The synthetic test cases allow to systematically determine properties of buffer overflows that lead to wrong buffer overflow detection results. Also, the results of analysing the synthetic test cases were used to compare the approach described in this thesis with tools for the analysis of C code. Furthermore, the media streaming server Icecast [Fou] was analysed as real world example.

## 4.1 Synthetic examples

Synthetic test executables generated by a code generator by Kratkiewicz [Kra05] were used for evaluation. As the generated test cases originally were designed for the evaluation of tools that analyse C code, some modifications described in section 4.1.1 had to be made. The executables were analysed with advantageous (section 4.1.5) and with more common (section 4.1.3) calling conventions. Finally, section 4.1.6 discusses how the VSA improvements described in section 3.2 affect the results of the buffer overflow detection.

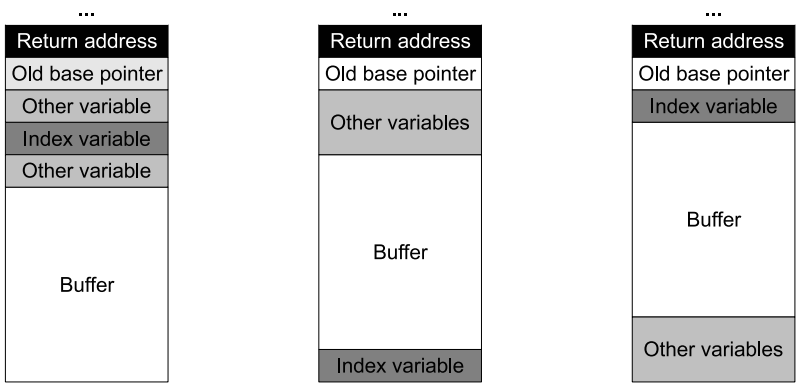
### 4.1.1 The test cases

The test cases used for evaluation were generated using a code generator by Kratkiewicz that was originally developed to compare several tools that detect buffer overflows in C code [Kra05]. With the configuration used, 291 different test cases are generated, each in four different variants. Three variants contain a buffer overflow and one does not. The variants with buffer overflows contain buffer overflows of different magnitude (cf. section 4.1.2). In total, 1164 programs were generated by the code generator.

The exact memory layout, i.e. the exact location of the variables, is not specified in the C code but instead chosen by the compiler. Consequently, the memory layout should not have any effects on the performance of tools that analyse C code. In executables however, the memory layout is specified and thus might affect the result of the buffer overflow detection. In order to determine the impact, three different versions with different stack layouts of each of the original 1164 programs were generated. Specifying the stack layout in the C

code is possible using packed structs which force the compiler to place certain variables in the specified order in the memory.

Figure 4.1 shows the three used stack layouts. The first used stack layout is the one that is chosen by the compiler. This layout might e.g. look like the one displayed in subfigure (a). Every other order of the variables is however possible as well. For the other two layouts, three different kinds of variables are distinguished: index, buffer and other variables. In the second stack layout, the index variables are located at the bottom of the stack, the buffer is located immediately above and all other variables are located at the top of the stack frame. This layout shown in subfigure (b) represents the worst case order as it maximizes the number of bytes between the buffer and the return address. This potentially prevents detection with the return address heuristics or the jump heuristic. Also, detecting the buffer overflow with the index heuristic is impossible using this layout. In the third layout, the order of the variables is reversed compared to the second layout: the index variables are located at the top of the stack frame, the buffers immediately below and all other variables at the bottom. This is shown in subfigure (c). This order especially benefits the index heuristic. In test executables that contain multiple functions, the stack layout modifications were applied to each one. As three variants were generated of each of the of the original 1164 test executables, 3492 test executables were generated in total.




(a) A stack layout possibly chosen by the compiler

(b) The others-buffer-index stack layout

(c) The index-buffer-others stack layout

Figure 4.1: The three different stack layouts used in the test executables.

In addition to the modification of the stack layout, three special instructions are added to the executable to permit fully automated evaluation. This is illustrated in figure 4.2. Subfigure 4.2a shows the original version and subfigure 4.2b the version with marking instructions. The additional instruction in line 2 marks the start of the analysis. The two other marking instructions in line 5 and line 7 are located immediately before and after the buffer overflow. A single alleged buffer overflow between those two instructions is considered a correct detection. The marking instructions are compare instructions which modify only the processor flags but nothing else. Thus, they should not influence the VSA or buffer overflow detection were used as marking instructions.

The test executables were compiled using GCC version 4.2.3 on Ubuntu

```
1  int main() {
2    char buf[10];
3
4    buf[4105] = 'A';
5
6    return 0;
7  }
```

(a) A test program

```
1  int main() {
2    asm("cmp $0x606060, %edi");
3    char buf[10];
4
5    asm("cmp $0xBAD, %edi");
6    buf[4105] = 'A';
7    asm("cmp $0x600D, %edi");
8
9    return 0;
10 }
```

(b) The test program with marking instructions

Figure 4.2: A test executable without and with marking instructions. The assembler instructions are written in AT&T syntax used by GCC while the rest of this thesis uses Intel syntax.

Linux with kernel version 2.6.24. The option `-fno-stack-protector` which disables the GCC's dynamic buffer overflow detection was used. The executables were analysed both using the cdecl and stdcall calling conventions. The caller is responsible for removing the arguments from the stack in the cdecl calling conventions. Thus, the argument heuristic can not be used (cf. section 4.1.5). In the stdcall calling conventions in contrast, the callee is responsible for removing the argument from the stack and thus the argument heuristic can be used.

### 4.1.2 Types of buffer overflows

Each test case is characterized by 22 attributes defined by Kratkiewicz [Kra05]. One of the attributes e.g. indicates whether the buffer overflow is a reading or a writing one. There are two attributes that have large impact on the precision of the buffer overflow detection: the magnitude and whether the buffer overflow is a continuous or a discrete one. The magnitude of a buffer overflow specifies by how many elements the used index value exceeds the range of valid index values. Figure 4.3 shows a short program in four different variants. The variants in subfigures (a), (b) and (c) do contain a buffer overflow and the one in subfigure (d) does not. Each of the programs consists of the declaration of one array variable with ten elements and an access to that variable. In the program in subfigure (a), the index value used in the buffer access is 4105. As the buffer consists of ten elements, the highest legal buffer index is 9. Thus, the program contains a buffer overflow of magnitude $4105 - 9 = 4096$. Analogously, the program in subfigure (b) contains a buffer overflow of magnitude 8 and subfigure (c) contains a buffer overflow of magnitude 1. Each of the basic 291 test cases is generated in the four variants shown in figure 4.3, i.e. in one variant without buffer overflow and three variants with buffer overflows of magnitudes 1 ("small"), 8 ("medium") and 4096 ("large").

The second important attribute of a buffer overflow indicates, whether it is a continuous or a discrete buffer overflow. Continuous buffer overflows consist of a loop that subsequently overwrites or reads multiple elements of the buffer

```
char buf[10];
buf[4105] = 'A';
```

(a) Large buffer overflow

```
char buf[10];
buf[17] = 'A';
```

(b) Buffer overflow of medium magnitude

```
char buf[10];
buf[10] = 'A';
```

(c) Small buffer overflow

```
char buf[10];
buf[9] = 'A';
```

(d) No buffer overflow

Figure 4.3: Four variants of a short program. One does not contain a buffer overflow and three contain buffer overflows of different magnitude.

and that continues to write or read past the boundaries of the buffer. Discrete buffer overflows in contrast consist of a single memory access that is not part of a loop. Thus, continuous buffer overflows access a consecutive section in the memory while discrete buffer overflows access only one location in the memory.

Figure 4.4 displays one program with a continuous and one with a discrete buffer overflow. The program in subfigure (a) allocates an array of ten elements in line 1. The loop in lines 3 to 6 overwrites the elements of the buffer starting from element 0. The loop does however not stop when the maximum legal index 9 is reached. Instead, the loop continues to overwrite 4096 further elements. Thus, the program contains a continuous buffer overflow of magnitude 4096 ("large").

```
1 char buf[10];
2 int i = 0;
3 while(i < 4106){
4   buf[i] = 'A';
5   i = i + 1;
6 }
```

(a) Continuous

```
1 char buf[10];
2 buf[4105] = 'A';
```

(b) Discrete

Figure 4.4: A continuous and a discrete buffer overflow.

The program in subfigure (b) also uses an array variable of ten elements. The program does however not contain a loop but only the single buffer access in line 2. This buffer access uses the index 4105 which is larger than the maxmimum legal index of 9. Thus, the program contains a discrete buffer overflow of magnitude $4105 - 9 = 4096$.

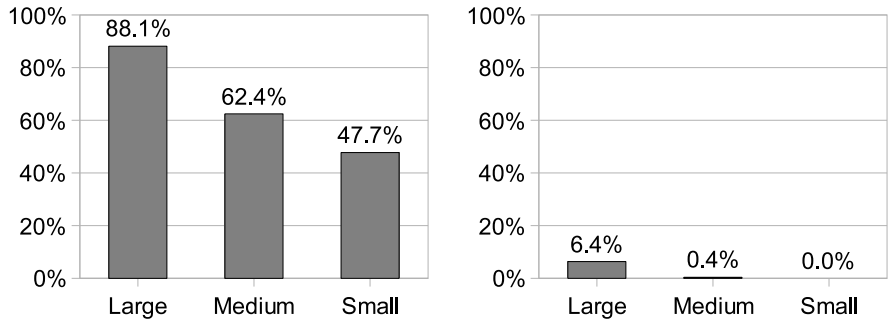
Continuous buffer overflows tend to be easier to exploit as they allow to overwrite an entire section in the memory. This makes it much more likely that a return address is affected. Furthermore, it is in many cases possible to redirect the control flow and to send code that is executed after the control flow has been redirected in a single exploit of a continuous buffer overflow. This is not possible when exploiting discrete buffer overflows. Thus, discrete buffer overflows might be considered less severe. 118 of the 291 basic test cases contain continuous buffer overflows and 173 contain discrete ones.

### 4.1.3 Results with disadvantageous calling conventions.

There are two possible types of incorrect results of the buffer overflow detection: false positives and false negatives. A false positive occurs when the analysis reports a buffer overflow at a location in the program where no buffer overflow exists. A false negative occurs when the analysis fails to detect an existing buffer overflow.

The metrics used for evaluation of the buffer overflow detection are the detection rate and the false alarm rate. The detection rate is the number of correctly detected buffer overflows divided by the number of executables with buffer overflows. The false alarm rate is the number of false positives in executables without buffer overflow divided by the number of executables without buffer overflows. The programs were compiled with the default cdecl calling conventions which do not allow to use the argument heuristic (cf. section 3.1.4).

The buffer overflow detection did not report a single false positive. Thus, the false alarm rate was zero. The reason why no false positives were reported is that the value sets computed by the VSA tend to be very precise for small and simple examples like the synthetic test cases. Figure 4.5 shows the detection rates for different types of buffer overflows.




(a) Continuous (b) Discrete

Figure 4.5: The detection rates of the buffer overflow detection when the test cases are compiled with disadvantageous calling conventions.

Subfigure (a) shows the detection rates for continuous buffer overflows of different magnitudes. 88.1% of the large continuous buffer overflows were detected. However, only 62.4% of the buffer over flows of medium magnitude and only 47.7% of the small ones were detected. The reason why the detection rate is smaller for buffer overflows of smaller magnitude is that only the index heuristic (cf. section 3.1.5) is suited for detecting buffer overflows of small magnitude. The return address heuristic (cf. section 3.1.2) can only detect buffer overflows that potentially overwrite the return address. Similarly, the jump heuristic (cf. section 3.1.3) can only detect buffer overflows that reach up to a location above the return address. The index heuristic however only detects the buffer overflows in the advantageous stack layouts. The reason why also some continuous buffer overflows of large magnitude were not detected are explained at the end of this section.

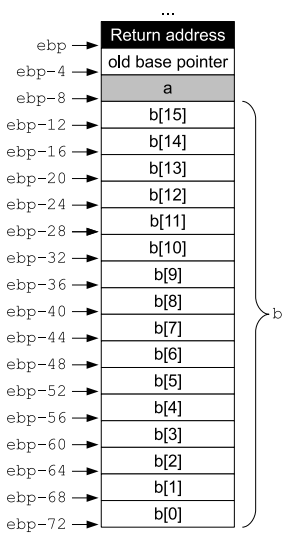
While many of the continuous buffer overflows could be detected, the de-

```
1  int main(){
2    int a;
3    int b[16];
4    a = 42;
5  }
```

(a) No buffer overflow

```
1  int main(){
2    int a;
3    int b[16];
4    b[16] = 42;
5  }
```

(b) With buffer overflow




(c) The stack layout

```
1   lea     ecx, [esp+arg_0]
2   and     esp, 0FFFFFFF0h
3   push    dword ptr [ecx-4]
4   push    ebp
5   mov     ebp, esp
6   push    ecx
7   sub     esp, 50h
8   mov     dword ptr [ebp-8], 2Ah
9   add     esp, 50h
10  pop     ecx
11  pop     ebp
12  lea     esp, [ecx-4]
13  retn
```

(d) Disassembly of the program both with and without buffer overflow

Figure 4.6: Although one of the shown programs contains a buffer overflow and the other one does not, the resulting binary code after compilation is the same.

tection rates for discrete buffer overflows which are shown in subfigure (b) are significantly lower. Only 6.4% of the buffer overflows of large magnitude and only 0.4% of the ones of medium magnitude were detected. Not a single buffer overflow of small magnitude was detected. Discrete buffer overflows are very difficult to detect in assembler code due to the fact that there is no information about variable boundaries. Thus, it is often difficult to tell whether a given memory access really was intended to access the location it does access. This is illustrated in figure 4.6. Subfigure (a) shows a short program that does not contain any buffer overflow. Despite the fact that the program in subfigure (b) *does* contain a buffer overlow, both programs result in the same executable after compilation with GCC. Subfigure (d) shows the disassembly of the main function of both programs. Both programs use an integer variable `a` and an array of 16 elements called `b`. The two programs differ in that the version without buffer overflow accesses `a` in line 4 while the one with buffer overflow accesses `b`. For this access to `b`, the index 16 is used. As `b` has 16 elements, only indices ranging from 0 to 15 would be legal. Thus, the buffer access in line 4 of subfigure (b) is a discrete buffer overflow.

Subfigure (c) shows the stack layout of the main function in both programs. `a` is located immediately above `b` on the stack. Thus, the assignments `b[16] = 42;` and `a = 42;` access the same memory address and thus both result in the same instruction in line 8 in the assembler code.

As the executables of both programs are exactly the same, every buffer over-

flow detection approach that analyses executables would either detect a buffer overflow in both or in none of the programs. This illustrates that the reliable detection of discrete buffer overflows can be very hard and even impossible in some cases when analysing executables. The buffer overflow detection is however able to detect some discrete buffer overflows. The reason is that discrete buffer overflows that are more complicated than the one in figure 4.6b can in some cases be detected using the index or jump heuristic. Also, some of the discrete buffer overflows access the return address and can thus be detected using the return address heuristic.

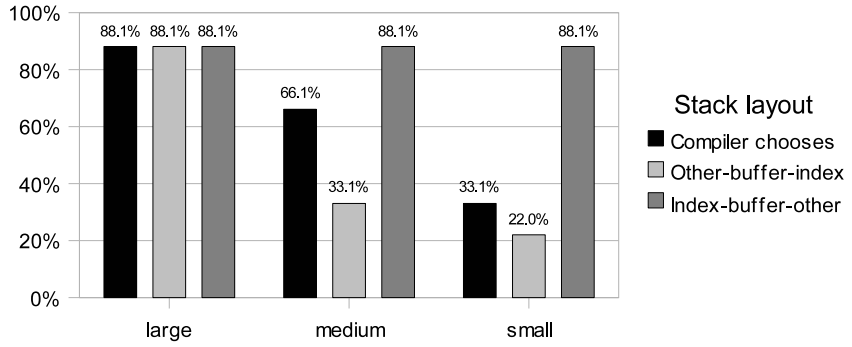
Discrete buffer overflows and small magnitude cause many false negatives. There are however some additional properties that may lead to false negatives even in executables that contain a large, continuous buffer overflow:

- **Library functions.** In fourteen of the basic 291 test cases, the buffer overflow occurs not in the executable itself but in a function that is loaded from a library at runtime. As our implementation of the VSA skips these functions during analysis, such buffer overflows are not detected.
- **Buffer overflow not on stack.** In six test cases, the buffer is not located on the stack. Instead, the buffers in which the overflows occur are global variables or are located on the heap or in shared memory. In these cases detection fails as buffer overflow detection aims for overflows on the stack exclusively.
- **Index value set is $\top$ .** In four test cases, no other value set than $\top$, i.e. the set of all potential values, could be calculated for the index variable. As a results, the location set of the buffer access also is $\top$ and no buffer overflow is detected (cf. section 3.1.1).
- **Multithreaded programs.** Two of the test cases, are multithreaded and the buffer overflow does not occur in the main thread. In this case no path exists from the position where the analysis starts to the location where the buffer overflow occurs in the CFG. Thus, that position is not analysed and the detection fails.
- **Lower bound overflows.** In one test case, not a location behind the upper bound of the buffer but one below the lower bound is accessed. The only heuristic that theoretically could detect such a buffer overflow is the index heuristic. This however does not happen since the buffer overflow is also a discrete one and therefore no index variable is involved.

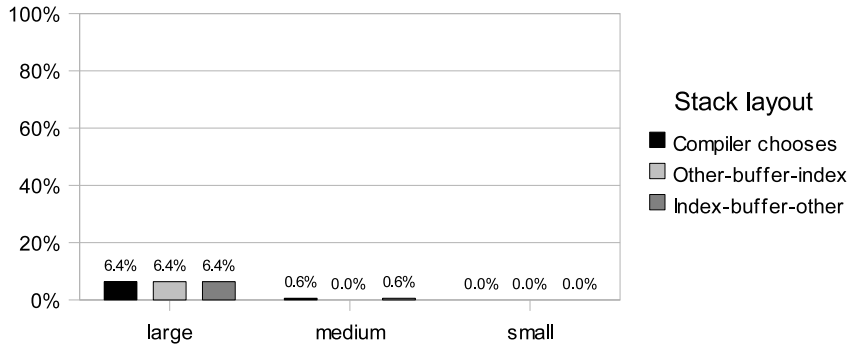
### 4.1.4 Influence of the stack layout

As described in section 4.1.1, three different variants with different stack layouts are generated of each of the original 1164 programs. In the first variant, the compiler chooses the stack layout. The second variant is the others-buffer-index layout in which the index variable is located at the bottom of the stack frame. The buffer is located immediately above the index variable and all other variables are located above the buffer immediately below the old base pointer. This makes detection of buffer overflows rather difficult. The third variant is the index-buffer-others layout in which the index variable is located at the top of the stack frame, the buffer is located below and all other variables are located

at the bottom of the stack frame. The index heuristic especially benefits from this layout.




(a) Continuous




(b) Discrete

Figure 4.7: The detection rates in the different stack layouts.

Subfigure (a) shows the detection rates of continuous buffer overflows by stack layout. The stack layout does not affect the detection rate in executables with large continuous buffer overflows. The reason is that the continuous buffer overflows of magnitude 4096 are sufficiently large with all stack layouts to reach up to the next return address. Thus, unless there are other difficulties that prevent detection, the large continuous buffer overflows are detected by the return address heuristic in all stack layouts.

The continuous buffer overflows of medium or small magnitude do not necessarily reach up to the next return address. Thus, the stack layout does make a difference in these executables. As expected, the detection rate is the highest in the executables with the index-buffer-other stack layout. The index variable is located immediately above the buffer in this layout. Hence, even buffer overflows of magnitude one modify the index variable and they can be detected by the index heuristic. As a result, with this layout the detection rate does not depend on the magnitude at all.

The other-buffer-index layout places the index variable below the buffer on the stack. Hence, the index heuristic can only detect lower bound overflows in this layout. Also, the distance between buffer and return address is maximised

in this layout which in many cases prevents detection by the return address heuristic and the jump heuristic. Consequently, the detection rate of buffer overflows of small and medium magnitude is significantly lower than with the index-buffer-others layout. The detection rate ranges between the one of the others-buffer-index layout and the one of index-buffer-others layout when the compiler chooses the stack layout.

Subfigure (b) shows the detection rate of discrete buffer overflows by stack layout. The results are similar to the one of the continuous buffer overflows. The detection rate does not depend on the stack layout if the magnitude is large. Again, the detection rate is much lower with the other-buffer-index than with the other layouts in the executables with medium magnitude buffer overflows. Small discrete buffer overflows are not detected with any layout.

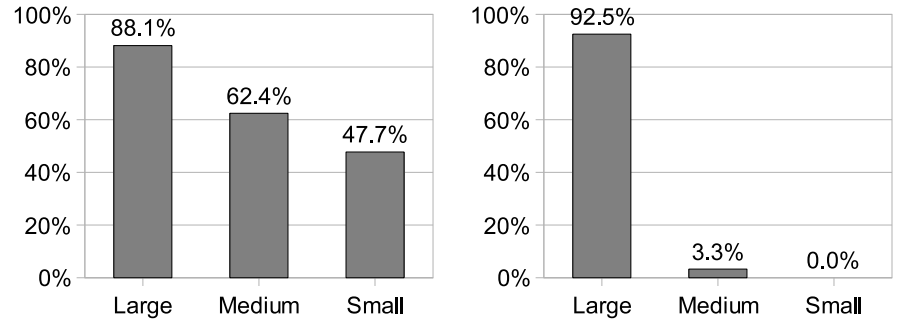
### 4.1.5 Results with advantageous calling conventions

As second evaluation scenario the test cases were compiled with the stdcall calling conventions in which the callee is responsible for removing the arguments from the stack. Consequently, the exact number of argument bytes can be determined and the argument heuristic can be used (cf. section 3.1.4).

Unlike when compiling with disadvantageous calling conventions, false positives occurred in the executables with advantageous calling conventions. One false positive occurs in all versions of one basic test case. The cause of the false positive is the way library functions are treated by the VSA implementation used for this thesis. Unlike the original implementation, this implementation does not analyse external library functions. Instead, they are simply ignored. The only exception is the stack pointer which is updated according to the function. For this purpose, the function's effects on the stack pointer have to be known. A special heuristic is used in order to determine the effects of external functions on the stack pointer. If the heuristic fails to determine the effects, the results of a the stack pointer analysis performed by IDA Pro are used. The false positives are caused by the fact that both the heuristic fails and IDA Pro computes a wrong stack pointer change for one call of an external function. As a result, the stack pointer value computed by the VSA is wrong after that call and an ordinary access to a local variable is erroneously considered a buffer overflow. Thus, the false positives are not directly caused by the use of the advantageous calling conventions but by the fact that determining the effects of an external library function on the stack pointer fails. The test case in which the false positives occur contains a discrete buffer overflow. Consequently, false positives occur in 0.3% off all test executables, 0.6% of the discrete test cases and 0.0% of the continuous ones.

Figure 4.8 shows the detection rates when using the advantageous calling conventions The detection rates of the continuous buffer overflows shown in subfigure (a) did not change compared to the detection rate with the disadvantageous calling conventions in figure 4.5a. The argument heuristic is able to detect buffer overflows that access a location above the highest argument byte (cf. section 3.1.4). As however continuous buffer overflows overwrite a consecutive section in the memory starting inside the stack frame, they will overwrite the return address before they reach a location above the highest argument byte. Hence, every buffer overflow that can be detected by the argument heuristic can also be detected by the return address heuristic. This explains why in addition

using the argument heuristic yields no improvement for the continuous buffer overflows.




(a) Continuous (b) Discrete

Figure 4.8: The detection rates of the buffer overflow detection when the test cases are compiled with advantageous calling conventions.

The detection rates of discrete buffer overflows are shown in figure 4.8b. The biggest difference compared to the detection rates in figure 4.5b is that 92.5% of the buffer overflows of large magnitude are correctly detected. The argument heuristic is able to detect most of the large buffer overflows due to the fact that every large buffer overflow accesses a location above the highest argument byte. Also, the detection rate for buffer overflows of medium magnitude is slightly increased. As however a magnitude of eight does in most cases not reach a location above the highest argument byte, still only 3.3% of the discrete buffer overflows of medium magnitude are detected. Buffer overflows of small magnitude never reach a location above the highest argument byte and thus still are never detected.

### 4.1.6 The influence of the VSA improvements

Section 3.2 describes two additions made to the VSA: delayed widening and an improvement to the analysis of conditional jumps. In order to estimate their influence, the executables with the disadvantageous calling conventions were analysed without these extensions.

Delayed widening tries to perform widening immediately before a loop condition (cf. section 3.2.1) in order to enable restriction of the value sets immediately after widening. The improved conditional jump analysis not only to restricts the value sets of the compared operands at conditional jumps, but also the value sets of variables that are guaranteed to have the same value. Both VSA extensions try to eliminate values from the value sets that can not occur at runtime. Hence, disabling the additions only adds values to the value sets. Consequently, disabling the additions does not increase the number of false negatives but only increases the number of false positives. All false positives occur in executables that do not contain a buffer overflow, i.e. no buffer overflow was detected at a wrong position in an executable that does contain a buffer overflow at another position.

Figure 4.9 shows the number of false positives with and without extensions. Using the buffer overflow analysis without both VSA extensions results in a 30.2% false alarm rate. Using delayed widening reduces the false alarm rate to 29.8%. The improvement is small because the widening point selection in many of the test cases already selects the best widening point. Using the improved conditional jump analysis without delayed widening yields a 2.7% false alarm rate. Not a single false positive occurs when both extensions are used. The improved conditional jump analysis reduces the false alarm rate drastically due to the fact that the index variables are in many test cases loaded into a register and then compared, similar to the example that was used to motivate the improved conditional jump analysis (cf. figure 3.15).

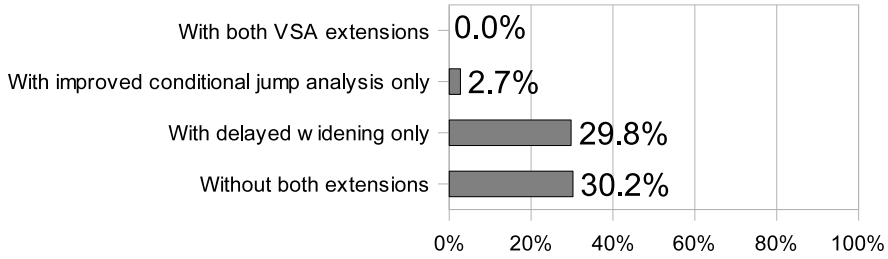



Figure 4.9: The number of test cases in which a false positive occurred with and without the use of the VSA extensions described in section 3.2.

Figure 4.10 shows the time required for the analysis with different settings when executed on a Intel Xeon 5080. The time includes the time required for disassembling the test executables by IDA Pro. The median of the time required is decrease by 6.0% when both extensions are disabled. The additional time required for the improved conditional jump analysis however leaves much room for improvement due to the fact that the implementation used is rather inefficient.

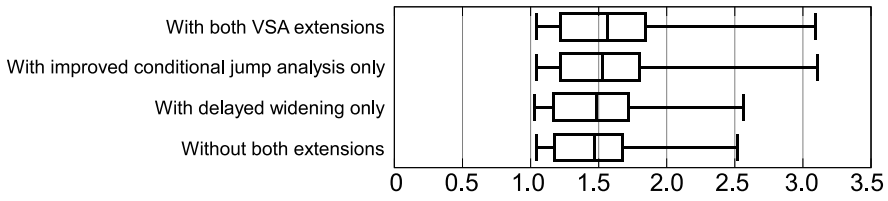



Figure 4.10: Minimum, maximum, 5% and 95% quantile and median of the time required for the analysis of the synthetic test cases with different configurations.

### 4.1.7 Comparison with other approaches

Originally, the synthetic test cases were used to compare different tools that try to detect buffer overflows in C code [Kra05]. In this section, the results of the C code analysis tools are compared with the ones of the VSA-based approach described in this thesis. The results of ARCHER [XCE03], BOON

[WFBA00], PolySpace [Tec], Splint [LE01, EL02] and Uno [Hol02] were obtained by Kratkiewicz [Kra05]. Splint differentiates between possible and likely buffer overflows in the current version 3.1.2. Detection rate and false alarm rate of counting all buffer overflows reported by Splint are the ones reported by Kratkiewicz. In addition to those results, detection rate and false alarm rate resulting from using only the likely buffer overflows were added using a slightly modified analysis script by Kratkiewicz [Kra05].

The metrics used by Kratkiewicz are chosen to show whether the compared tools are able to differentiate between the versions with and without buffer overflow of one basic test case. Thus, only false alarms at the buffer access that in three versions is a buffer overflow and in the fourth is not a buffer overflow are taken into account for the false alarm rate. False positives at other locations are not taken into account [Kra05]. Using this definition, the false alarm rate of the VSA-based approach described in this thesis is zero also with the advantageous calling conventions. This is due to the fact that the false positives described in section 4.1.5 do not occur at that buffer access in question.

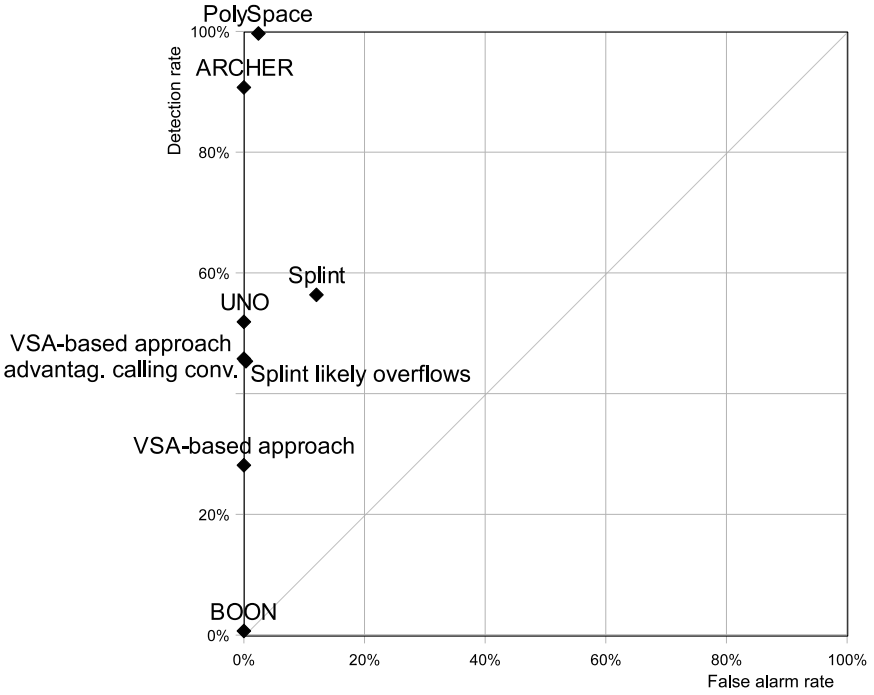
Figure 4.11 compares detection rate and false alarm rate of different approaches. The detection rate of BOON is very low due to the fact that BOON only tries to identify buffer overflows caused by the misuse of string manipulation functions [Kra05]. The approach showing the second lowest detection rate is the VSA-base approach described in this thesis when analysing the executables with disadvantageous calling conventions. Using the advantageous calling conventions instead, the detection rate is 45.8%. Splint has a slightly better detection rate when using only the buffer overflows considered likely by Splint. The false alarm rate however is unlike the one of the VSA-based approach is not zero. Counting all reported buffer overflows instead of only the ones considered likely raises the detection rate of splint by 11%. The downside is that the false alarm rate grows by 11.7%. Like the VSA-based approach and BOON, UNO has a zero false alarm rate. The detection rate is with 51.9% higher than the one of BOON, Splint and the VSA-based approach. ARCHER and PolySpace perform very well. ARCHER detects 90.7% at a zero false alarm rate. PolySpace detects almost all buffer overflows while the false positives rate still is very low.

One reason why most of the tools analysing C code perform better than the VSA-based approach is that the C code contains much more information about variable and especially buffer boundaries than the executable. This eliminates the VSA-based approach's most important sources for false negatives in the synthetic examples: insufficient magnitude and discrete buffer overflows.

The magnitude of the buffer overflows has no effect on whether a buffer overflow is detected or not by the C analysis tools. They either detect a buffer overflow in all magnitudes or not at all [Kra05]. Also, discrete buffer overflows are much more obvious in C code. E.g. the buffer overflow in figure 4.6b consists of the two instructions

```
int b[16];
b[16];
```

and can not be distinguished from a legal memory access in assembler code (cf. section 4.1.3). In C code in contrast, the buffer overflow is rather easy to detect as both determining the index value and the size of the buffer is very easy. Figure 4.12 shows the false alarm rate and detection rate of the compared approaches both on the continuous and the discrete buffer overflows.




(a) Graphical comparison

| Tool | Detection rate | False alarm rate |
|---|---|---|
| VSA based approach | 28.1% | 0.0% |
| VSA based approach with advantageous calling conventions | 45.8% | 0.3% |
| ARCHER | 90.7% | 0.0% |
| BOON | 0.7% | 0.0% |
| PolySpace | 99.7% | 2.4% |
| Splint | 56.4% | 12.0% |
| Splint "likely" overflows only | 45.4% | 0.3% |
| UNO | 51.9% | 0.0% |

(b) In numbers

Figure 4.11: Detection and false alarm rate of different approaches. Detection rate and false alarm rate of ARCHER, BOON, PolySpace, Splint and UNO have been taken from [Kra05]. Detection rate and false alarm rate of Splint with likely buffer overflows only have been obtained using a script by Kratkiewicz [Kra05].

Subfigure (b) shows the results of analysing the discrete buffer overflows. All C analysing tools except for BOON show a high detection rate. The VSA based approach shows a rather low detection rate as described in the previous sections.

Subfigure (a) shows the results of analysing only the test cases with continuous buffer overflows. Here, the detection rate of the VSA-based approach is much higher. PolySpace and ARCHER detect almost all continuous buffer overflows. Splint and UNO however perform much worse than in the case of discrete buffer overflows.

## 4.2 Real world example

While synthetic examples are suited well to systematically determine strengths and weaknesses of a given approach, they have only limited significance for the real world as the tend to be much smaller and less complicated than real examples. Therefore, the media streaming server Icecast [Fou] was analysed as well.

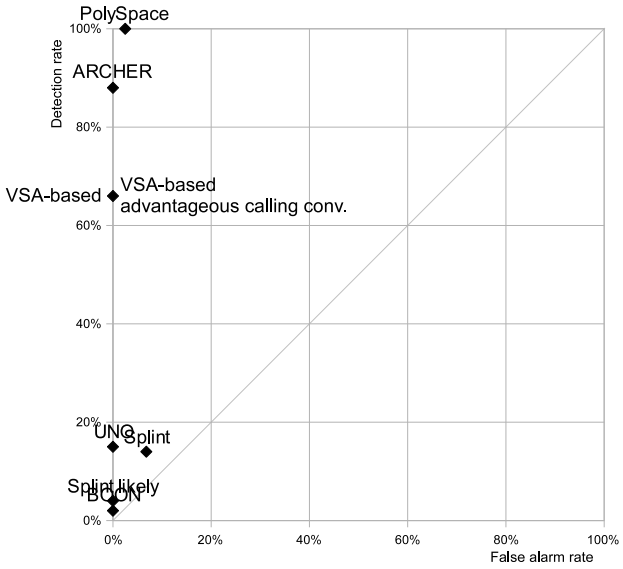
Section 4.2.1 describes why Icecast was chosen and which difficulties were encountered in other executables. The analysed version of Icecast contains a buffer overflow which is described in section 4.2.2. The last two subsections describe the manual analysis required in order to analyse Icecast and the results of the analysis.

### 4.2.1 The choice of a suitable program

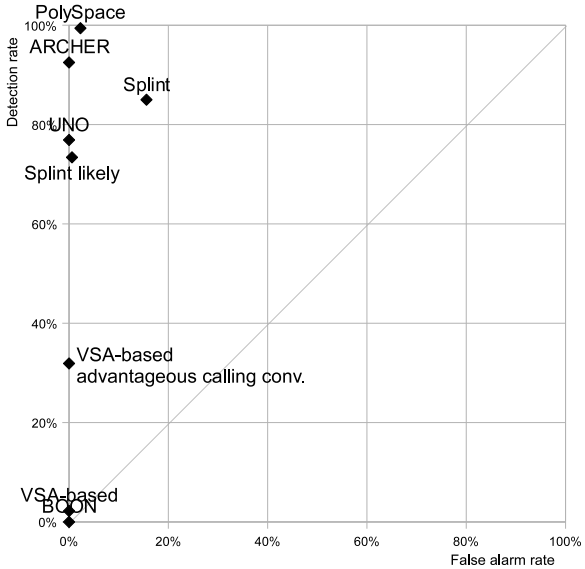
Only executable that contain at least one stack buffer overflow and of which the source code is available were considered potential real world examples. The reason why only programs of which the source code is available were taken into account is that manual reverse engineering is required in order to determine, whether an alleged buffer overflow is real. The time required for reverse engineering can be drastically reduced using the source code of the analysed program. Programs that meet these requirements where identified with the help of Metasploit [LLC], a tool for penetration testing that contains information about numerous vulnerabilities as well as the corresponding exploits.

Three different programs where considered as potential real world examples: Sendmail [Con], a mail server, PuTTY [Tat], a SSH client, and Icecast [Fou], a multimedia streaming server. Sendmail and PuTTY however turned out to be difficult to analyse. Sendmail simply was too large to analyse it on the available hardware. PuTTY uses coroutines which is an alternate concept of subroutines that, unlike the usual concept, omits the caller-callee relation between subroutine and instead puts all subroutines on the same level [Con63]. The abstract memory model used by the VSA (cf. section 2.5.5) however is based on the classical subroutine concept. Thus, analysing PuTTY would result in unrealistically bad results.

Eventually, the Windows version of Icecast 2.0.1 was chosen as real world example. Icecast 2.0.1 contains a stack buffer overflow that allows an attacker to execute arbitrary code on the attacked host by sending a fake HTTP request of more than 32 lines.




(a) Continuous buffer overflows only




(b) Discrete buffer overflows only

Figure 4.12: Detection and false alarm rate of different approaches both in the executables with discrete and in the ones with continuous buffer overflows. Detection rate and false alarm rate of ARCHER, BOON, PolySpace, Splint and UNO have been taken from [Kra05]. Detection rate and false alarm rate of Splint with likely buffer overflows only have been obtained using a script by Kratkiewicz [Kra05].

```
1  int httpp_parse(...){
2    ...
3    char *line[32];
4    ...
5    lines = split_headers(data, len, line);
6    ...
7  }
8
9  static int split_headers(char *data, unsigned long len,
10                                                  char **line){
11   int last_line = 0;
12   ...
13   for (i = 0; i < len && num_lines < 32; i++) {
14     ...
15     if (data[i] == '\n') {
16       last_line++;
17       ...
18       line[last_line] = &data[i + 1];
19     }
20   }
21   ...
22 }
```

Figure 4.13: An excerpt from the Icecast 2.0.1 source code. Line 18 contains a buffer overflow.

### 4.2.2 The buffer overflow

Icecast 2.0.1 contains an off-by-one buffer overflow on the stack. Figure 4.13 shows a slightly simplified excerpt of the Icecast source code that illustrates the buffer overflow. The function `httpp_parse` in lines 1 to 7 parses a HTTP request. In this process, an array of 32 pointer values called `line` is used. The variable `data` contains the HTTP request. The function `split_headers` called in line 5 fills `line` with pointers to the locations in `data` where new lines in the HTTP request start. If `data` contains more than 32 lines, only pointers to the first 32 should be stored in `line`. Due to a bug, the function however may store up to 33 pointers resulting in a buffer overflow.

The variable `last_line` in the function `split_headers` contains the index of the most recently stored pointer to a line beginning and is initialized in line 11. The main part of the function is a loop that searches for newline characters in the HTTP request and stores the pointers to the line beginnings in `line`. The loop condition in line 13 ensures that the value of `last_line` is below 32 at the beginning of the loop body. Line 15 checks, whether the current character in the HTTP request is a newline character. If this is the case, `last_line` is increased by one and a pointer to the next character in the HTTP request is stored in `line[last_line]` in line 18 of the source code. As the value of `last_line` is at most 31 at the beginning of the loop, its value can be at most 32 after the increment in line 16. As however `line` is an array of only 32 elements, only

index values in the range from 0 to 31 are allowed. Hence, an index value of 32 results in an off-by-one buffer overflow. Thus, an attacker can trigger a buffer overflow by sending a HTTP request of more than 32 lines.

As the variable line is located immediately below the return address of the function `httpp_parse` on the stack, the buffer overflow overwrites the return address of that function. Hence, if a HTTP request contains more than 32 lines, the return address of `httpp_parse` is overwritten with a pointer to the 33rd line. Consequently, the control flow will jump to the 33rd line of the HTTP request after the function `httpp_parse` returns. The buffer overflow is very easy to exploit because the return value is not overwritten with a value chosen by the attacker but instead with a *pointer* to a value chosen by the attacker. If an attacker creates a fake HTTP request that contains code in the 33rd line, the return address of `httpp_parse` will be overwritten with a pointer to that code and the code will be executed after `httpp_parse` returns.

### 4.2.3 Manual analysis required

Before the buffer overflow analysis can be executed, some information has to be specified manually, mostly due to the fact that the VSA implementation used does not include all features yet.

One feature that is not supported yet is the analysis of external libraries. Thus, when a library function call is reached by the VSA, the value sets are propagated along the intraprocedural control flow edge to the instruction to which the called library function returns ignoring the effects of the library function. One exception is the value set of the stack pointer which is updated according to the function effects. For this purpose, the function's effect of the stack pointer have to be known. The used implementation of the VSA tries to determine each library function's effects on the stack pointer both using a heuristic and information obtained from IDA Pro. If however both fail for a given library function, the stack pointer effects of that function have to be specified manually. This was the case for 14 functions in Icecast.

Usually, memory is allocated on the stack by simply subtracting the number of bytes to allocate from the stack pointer (cf. section 2.2.4). If the number of bytes to allocate however is above a certain threshold, the compilers use a stack allocation function instead. With the used implementation of the VSA, analysing this stack allocation function results in imprecise value sets for the stack pointer which almost certainly causes serious trouble in the further analysis. Therefore, the stack allocation function is not analysed but instead simply the number of bytes specified by the function's argument is subtracted from the stack pointer. For this purpose, it has to be specified manually which function is a stack allocation function. An alternative to this approach would be an VSA extension called affine-relations analysis [Bal07]. The affine-relations analysis keeps track of affine relations between the registers and thus allows not only to determine possible values for each register but also which combinations of register values are possible. Using this additional information, analysing the stack allocation functions should yield the correct results.

In addition to the stack pointer effects and the stack allocation function, the user has to specify where the analysis has to start. Each thread is analysed separately. Thus, the user has to specify all entry points of threads. Icecast contains eight different threads. However, only seven of them were analysed.

One thread of Icecast contains code related to the graphical user interface. This code mainly consists of one main loop that calls event-handling functions. While the callback functions are located inside the Icecast executable, the main loop is located in an external library function and thus can not be included in the analysis with the used VSA implementation. Analysing the GUI thread without this main loop makes little sense and thus the main thread was not analysed at all. Again, this problem could be solved by including library code into the analysis like the original VSA implementation does [Bal07].

### 4.2.4 Analysis results

Each of the seven threads in Icecast was analysed separately. Figure 4.14 shows for each thread the number of instructions analysed, how often each instruction was analysed on average before a fixpoint was reached, how often an instruction was analysed at maximum, the number of functions analysed and the duration of the analysis in seconds on an Intel Xeon 5080. In total, the analysis required approximately one 67 minutes. The performance evaluation of the original VSA implementation [BRS04] however suggests that a reduction of computing time may be achieved through code changes. This was however not a central topic of this thesis.

| Thread entrypoint | | Instructions | Times analysed | | Functions | Duration |
|---|---|---|---|---|---|---|
| Function name | Address | analysed | average | max | analysed | (seconds) |
| _slave_thread | 40E090 | 80072 | 2.60 | 13 | 238 | 674 |
| fserv_thread_function | 410C40 | 57926 | 2.49 | 16 | 170 | 464 |
| _handle_connection | 412A70 | 123120 | 2.68 | 30 | 321 | 1155 |
| _stats_thread | 4084F0 | 9606 | 1.55 | 7 | 42 | 71 |
| stats_connection | 4089F0 | 18545 | 3.63 | 9 | 42 | 125 |
| source_main | 40B780 | 107210 | 3.55 | 17 | 225 | 993 |
| yp_touch_thread | 40C790 | 66123 | 2.73 | 11 | 170 | 545 |

Figure 4.14: Number of instructions analysed, average and maximum times an instruction has been analysed, number of functions analysed and the analysis duration for each thread.

In total, 42 buffer overflows were reported. One of these buffer overflows is the one described in section 4.2.2 and thus a correct detection. The other 41 reported buffer overflows are false positives or likely false positives. In many cases, there are several false positives due to the use of the same pointer value at multiple positions in one loop. In total, the false positives occur in eight different functions. The table in figure 4.15 shows the number of reported buffer overflows by function. The second column shows whether the function is an API function or a function from the Icecast source code. Columns three to seven show how many buffer overflows were reported, whether the blamed memory accesses were reading or writing ones and whether the reported buffer overflows were real buffer overflows. The last three columns show which properties of the functions led to false positives.

Reverse engineering of the functions in which the buffer overflows were reported revealed three causes for the false positives:

- **Loops iterating over zero-terminated strings ($Z$).** One cause of false positives are loops iterating over zero-terminated strings. Figure 4.16

| Function Name | API? | Reported buffer overflows Total | Read | Write | Correct | False pos. | Reason(s) Z | C | A |
|---|---|---|---|---|---|---|---|---|---|
| split_headers | No | 1 | | 1 | 1 | | | | |
| parse_headers | No | 3 | 3 | | | 3 | | x | |
| strcmp (inline) | Yes | 4 | 4 | | | 4 | x | | |
| strftime | Yes | 2 | | 2 | | 2 | | x | |
| strncat | Yes | 12 | 3 | 9 | | 12 | | x | x |
| memset | Yes | 2 | | 2 | | 2 | | x | |
| strcat | Yes | 10 | 3 | 7 | | 10 | x | | x |
| _write_lk | Yes | 2 | | 2 | | 2 | | x | |
| __crtCompareStringA | Yes | 6 | | 6 | | 6 | x | | |

Figure 4.15: Buffer overflows reported in Icecast

shows such a loop. The loop continues to overwrite the elements of `buffer` until a zero character is found. Similar loops are commonly used in C and C++. If `buffer` contains a zero character, the loop will stop overwriting `buffer` before `buffer`'s upper boundary and no buffer overflow occurs. Hence, if `buffer` always contains at least one zero character, the buffer access in line 3 is safe. Analysing such a loop with the VSA is problematic because the loop condition does not use `i`. Thus, the loop condition can be used to restrict the value set of `buffer[i]` but not to restrict the one of `i`. Consequently, the buffer overflow detection is not able to compute an upper bound for `i` and a buffer overflow is reported in line 3 even if `buffer` always contains a zero character. Similar difficulties occur whenever the loop condition uses the content of the buffer rather than the one of the index variable.

```
1 int i = 0;
2 while(buffer[i] != '\0'){
3   buffer[i] = 'A';
4   i = i + 1;
5 }
```

Figure 4.16: A loop that fills a buffer with As until a zero character is found.

Determining, whether a loop that iterates over a zero-terminated string contains a buffer overflow is difficult for a human as well. In order to determine, whether the loop contains a buffer overflow, it has to be determined whether the buffer contains under all conditions at least one zero character. For a large program like Icecast this is very difficult and time consuming. Thus, only a short check for obvious buffer overflows was done for the alleged buffer overflows in loops iterating over zero-terminated strings. Consequently, these reported buffer overflows might be correct as well. Either way, the buffer overflow detection approach is not able to differentiate between safe loops iterating over zero-terminated strings and ones containing buffer overflows.

The functions where false positives are caused caused by a loop iterating over a zero terminated string are marked in the "Z" column of the table in figure 4.15.

- **Loop condition does not use index variable (*C*).** If the loop condition does not use the index variable but a different expression, then the value of the loop condition can not be restricted using the loop condition and a buffer overflow is likely. This is illustrated by the program in figure 4.17.

```
1 int i = 0;
2 int left = 20;
3 while(left > 0){
4   buffer[i] = 'A';
5   i = i + 1;
6   left = left - 1;
7 }
```

Figure 4.17: A small program that overwrites the first 20 elements of `buffer` with As.

The program contains a loop that overwrites the first 20 elements of `buffer` with As. Assuming that `buffer` has at least 20 elements, the program does not contain a buffer overflow. In the loop, the variable `i` is used as index variable. However, not `i` but `left` is used to keep track of the number of remaining bytes to be written. Consequently, the loop condition can be used to restrict the value set of `left` but not the value set of `i`. Hence, the value set of `i` at the buffer access in line 4 is $\{0, 1, \ldots, \infty\}$ after widening has been performed. Hence, a buffer overflow is reported.

These difficulties might be mitigated using the affine-relations analysis proposed by Balakrishnan [Bal07]. The affine-relations analysis keeps track of and makes use of affine relations between the registers. If `i` and `left` in the example were registers variables, the affine-relations analysis might be able to determine that `i` equals 20−`left`. This relation might be used to restrict the value set of `i` at the loop condition as well. The affine-relations analysis however only keeps track of affine relations among registers. Thus, it does not help if both variables are register variables.

The functions where false positives are caused caused by a loop iterating over a zero terminated string are marked in the "C" column of the table in figure 4.15.

- **Counter variable used after loop (*A*).** The value set analysis is able to compute quite precise value sets for index variables inside loops given that the index variable is used in the loop condition and that widening is performed immediately before the loop condition (cf. section 3.2.1.5). Unfortunately, the value sets after the loop are less precise, even if an advantageous widening point is chosen. This can lead to false positives. The described situation is illustrated in figure 4.18

  The program consists of one loop that overwrites the first 19 elements of the buffer `buffer` with As. Then, the 20th element is overwritten with a zero character in line 6. Assuming that `buffer` has at least 20 elements, the program does not contain a buffer overflow. Unfortunately, the buffer overflow detection would report one in line 6.

```
1  int i = 0;
2  while(i < 19){
3     buffer[i] = 'A';
4     i = i + 1;
5  }
6  buffer[i] = '\0';
```

Figure 4.18: A small program that overwrites the first 19 elements of `buffer` with As and the 20th with a zero character.

During the analysis of the loop, widening is performed at one point. After this process, the value set of `i` would become $\{0, 1, \ldots, \infty\}$ before the loop condition in line 2. According to the loop condition, the control flow remains inside the loop if `i` is smaller than 19. Thus, the value set of $\{0, 1, \ldots, \infty\}$ would be split into the values of `i` that let the control flow remain inside the loop $\{0, 1, \ldots, 18\}$ and the values of `i` that let the control flow leave the loop $\{19, 20, \ldots, \infty\}$ . The latter value set would be propagated along the CFG edge that leaves the loop. Hence, when the analysis reaches line 6, the value set of `i` is $\{19, 20, \ldots, \infty\}$ and a buffer overflow is reported. Similar difficulties always occur whenever a counter variable is used as index outside the loop. The difficulties could be mitigated using the limited widening extension proposed by Balakrishnan [Bal07].

The functions where false positives are caused caused by a loop iterating over a zero terminated string are marked in the "A" column of the table in figure 4.15.

Icecast was also used to determine the cost of delayed widening in practice. Figure 4.19 shows the number of positions at which widening was performed ("effective widening points") and the total duration of the analysis both with and without delayed widening. Delayed widening increases the number of positions at which widening is performed by 16% to 39%. The analysis duration is increased by 10% at most which seems an acceptable sacrifice for the increase precision.

| Thread entrypoint | | # effective widening points | | | Duration (seconds) | | |
|---|---|---|---|---|---|---|---|
| Function name | Address | without | with | increase | without | with | increase |
| _slave_thread | 40E090 | 428 | 550 | 29% | 616 | 674 | 9% |
| fserv_thread_function | 410C40 | 240 | 325 | 35% | 429 | 464 | 8% |
| _handle_connection | 412A70 | 818 | 994 | 22% | 1070 | 1155 | 8% |
| _stats_thread | 4084F0 | 63 | 73 | 16% | 68 | 71 | 4% |
| stats_connection | 4089F0 | 74 | 96 | 30% | 120 | 125 | 5% |
| source_main | 40B780 | 678 | 900 | 33% | 935 | 993 | 6% |
| yp_touch_thread | 40C790 | 299 | 415 | 39% | 494 | 545 | 10% |

Figure 4.19: The number of positions at which widening was performed and the total duration of the analysis both with and without delayed widening.

# Chapter 5

# Summary and conclusions

Buffer overflow vulnerabilities are not only easy to overlook but also in many cases easy to exploit. In this thesis, a technique that detects stack buffer overflows in executables using static analysis techniques has been described. The presented approach is based on the value set analysis, a technique that computes sets of potential values for registers and variables. Synthetic test cases and a real world example have been used to evaluate the performance of the technique.

Chapter 2 explained buffer overflows, basic principles of the analysis of executables as well as two analysis techniques: reaching definitions analysis [ALSU07] and the value set analysis [Bal07, BRS04]. The reaching definitions analysis allows to determine the locations where each register was potentially defined at every position in the program. The value set analysis computes a set of potential values for every register and every variable. These value sets are fundamental for the buffer overflow detection.

Chapter 3 described the buffer overflow detection approach. The basic idea of the approach is to use the results of the value set analysis to compute the set of potentially accessed locations for each memory operand. Then, heuristics are used in order to determine which memory accesses are likely buffer overflows. Four different heuristics were introduced. The return address heuristcs considers memory accesses that potentially overwrite a return address buffer overflows. The jump heuristic tries to determine whether the address of a memory access jumps over a return address during address calculation. The third heuristic is the argument heuristic which identifies buffer overflows that potentially access a location above the upper bound of the current stack frame. The argument heuristic can only be used in functions with advantageous calling conventions. The fourth heuristic tries to identify small buffer overflows that do not reach up to the next return address. For this purpose, "index" variables are identified in a first step. In a second step, the heuristic determines whether an "index" variable is potentially overwritten by a memory access using that "index" variable.

Also, two modifications applied to the value set analysis were described. Delayed widening tries to increase the precision of the value set analysis in loops. The improved conditional jump analysis tries to increase the use of conditional jumps for the value set analysis.

The approach was evaluated in chapter 4. First, a set of synthetic test cases by Kratkiewicz [Kra05] was used. Different versions with different stack layouts

were generated of each of the original programs in order to determine, how the stack layout affects the buffer overflow detection.

No false positives occurred during analysis. The detection rate strongly depended on the magnitude of the buffer overflows and on whether they were continuous or discrete. Continuous buffer overflows overwrite a consecutive area in the memory while discrete ones only access a single location in the memory. As continuous buffer overflows make it possible for an attacker to affect larger parts of the memory, they are usually easier to exploit and can thus be considered more severe. The detection rate ranged from zero for the small discrete buffer overflows to 88.1% for large continuous ones.

It has been shown that the modifications made to the value set analysis are a necessary prerequisite for the reliable detection of buffer overflows as disabling them results in a 30.2% false alarm rate for the synthetic test cases.

Analysing the test cases in different variants with different stack layouts revealed that the detection rate strongly depends on the stack layout used. For small continuous buffer overflows, the stack layout can make the difference between a 22% and a 88.1% detection rate.

The used calling conventions may affect the performance of the analysis due to the fact that the argument heuristic can only be used with some calling conventions. The use of the stdcall calling conventions, which make use of the argument heuristic possible, increases the detection rates for discrete buffer overflows. Most notably, the detection rate for large discrete buffer overflows increases from 6.4% to 92.5%. On the other hand, false positives occurred in 0.3% of the test cases with the advantageous calling conventions. These false positives were however not directly caused by the stdcall calling conventions but occurred as side effect of the slightly different structure of the executables.

The VSA-based approach described in this thesis has been compared to five approaches for the analysis of C code. The VSA-based approach could not achieve the precision of most of the tools, especially in the executables with discrete buffer overflows. An example has been used to illustrate that detecting discrete buffer overflows in executables in general is difficult and in some cases even impossible. Analysing the continuous buffer overflows however, the VSA-based approach performed significantly better than three of the five C code analysis tools.

In order to evaluate the buffer overflow detection in a realistic scenario, the media streaming server Icecast [Fou] was analysed. 42 buffer overflows were reported. One of them was correct and the other 41 reported buffer overflows were false negatives or, in some cases, likely false negatives. On one hand this illustrates that the appraoch is also able to detect more complex buffer overflows in larger executables than synthetic test cases. On the other hand, the false alarm rate leaves room for improvement.

Three different causes of false positives could be identified: loops that iterate over zero-terminated strings, loop conditions that do not use the index variable and loop variables that were used after the loop.

**Future work.** Solving the difficulties with zero-terminated strings is one of the major open challenges for the presented approach. A possible solution might be to add mechanisms that retrieve additional information about strings to the value sets analysis. In practice, it is better not to detect buffer overflows in loops

that iterate over zero terminated string at all than to detect a buffer overflow in every such loop. Thus, a simpler approach might mitigate the difficulty by identifying loops that iterate over zero terminated strings and then suppressing all buffer overflow warnings inside those loops. The other two difficulties that lead to false positives in the real world example could be solved or at least mitigated using additional techniques described in [Bal07].

Another major challenge is the low detection rate for discrete buffer overflows. As has been shown, it is however in some cases impossible to detect discrete buffer overflows. Therefore, concessions will always have to be made when trying to detect discrete buffer overflows in executables.

In addition, it is desirable also to detect buffer overflows that are not located on the stack but in global variables or on the heap. Detecting buffer overflows in global variables is expected to be quite difficult as there is no obvious way to determine the boundaries of global variables. The detection of heap buffer overflows is expected to be much easier due to the fact that heap memory is allocated using special functions that receive the number of bytes to allocate as argument. Thus, a promising approach is to determine the maximum size of each chunk of heap memory and to use this information in order to determine whether the analysed program potentially writes past the boundaries of a chunk of heap memory.

# Bibliography


[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2007.

[Bal07] G. Balakrishnan. *WYSINWYX: WHAT YOU SEE IS NOT WHAT YOU EXECUTE*. PhD thesis, UNIVERSITY OF WISCONSIN, 2007.

[Bou93] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993.

[BR06] G. Balakrishnan and T. Reps. Recency-Abstraction for Heap-Allocated Storage. *LECTURE NOTES IN COMPUTER SCIENCE*, 4134:221, 2006.

[BRS04] G. Balakrishnan, T. Reps, and WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES. Analyzing Memory Accesses in x86 Executables. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 5–23, 2004.

[CFBV06] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static Detection of Vulnerabilities in x86 Executables. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 269–278, 2006.

[Con] Sendmail Consortium. Sendmail. http://www.sendmail.org/.

[Con63] M.E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[Cor99a] Intel Corporation. Intel Architecture Software Developers Manual Volume 1: Basic Architecture. *Order Number 243190*, 1999.

[Cor99b] Intel Corporation. Intel Architecture Software Developers Manual Volume 2: Instruction Set Reference. *Order Number 243191*, 1999.

[Cor99c] Intel Corporation. Intel Architecture Software Developers Manual Volume 3: System Programming. *Order Number 243192*, 1999.

[CPM+98] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium, 1998-Volume 7 table of contents*, pages 5–5. USENIX Association Berkeley, CA, USA, 1998.

[CSF98] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. *Software Maintenance, 1998. Proceedings. International Conference on*, pages 228–237, 1998.

[DRS03] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. *ACM SIGPLAN Notices*, 38(5):155–167, 2003.

[EL02] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE SOFTWARE*, pages 42–51, 2002.

[Eto03] H. Etoh. ProPolice: GCC Extension for Protecting Applications from Stack-Smashing Attacks. *IBM (April 2003), http://www.trl.ibm.com/projects/security/ssp*, 2003.

[Fou] Xiph.Org Foundation. Icecast 2.0.1. http://www.icecast.org/.

[Hol02] G. Holzmann. UNO: Static source code checking for userdefined properties. In *Proceedings of the World Conference on Integrated Design and Process Technology, Pasadena, CA, June*, 2002.

[Ida] PRO Ida. disassembler, Data Rescue.

[Kra05] K.J. Kratkiewicz. Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. Master's thesis, 2005.

[LE01] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10 table of contents*, pages 14–14. USENIX Association Berkeley, CA, USA, 2001.

[LLC] Metasploit LLC. Metasploit. http://www.metasploit.com.

[One96] Aleph One. Smashing The Stack For Fun And Profit. *Phrack magazine*, 49, 1996.

[Sec] Secure Software. RATS - Rough Auditing Tool for Security. http://www.fortify.com/security-resources/rats.jsp.

[Spo88] E.H. Spofford. The Internet Worm Program: An Analysis. 1988.

[Tat] Simon Tatham. PuTTY. http://www.chiark.greenend.org.uk/ sgtatham/putty/.

[Tec] P.S. Technologies. PolySpace C Developer Edition.

[VBKM00] J. Viega, JT Bloch, T. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, page 257. IEEE Computer Society Washington, DC, USA, 2000.

[WFBA00] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, volume 17. San Diego: CA, 2000.

[Whe] D.A. Wheeler. FlawFinder. http://www.dwheeler.com/flawfinder.

[XCE03] Y. Xie, A. Chou, and D. Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM New York, NY, USA, 2003.

[Zit03] M. Zitser. Securing software: an evaluation of static source code analyzers. Master's thesis, Massachusetts Institute of Technology, 2003.

# EIDESSTATTLICHE ERKLÄRUNG

Hiermit versichere ich, dass die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Bonn, 22. Dezember 2008