Master’s Thesis Nr. 39
Systems Group, Department of Computer Science, ETH Zurich

Eliminating Insecure Uses of C Library Functions

by
Raphael Tawil

Supervised by
Dr. Zachary Anderson
Prof. Timothy Roscoe

October 2011–April 2012
Abstract

Many C programs, especially legacy programs, still use insecure functions, for example functions like `gets()`, that are inherently insecure. Other functions lack bounds checking or important security checks. Despite the existence of operating system-level defenses that make use of techniques such as data execution prevention, and address space layout randomization, which reduce the risk of control-flow hijack attacks, greater assurances of security can be obtained by simply not using these functions in the first place.

The goal of this thesis is to improve the security of software by discovering and either automatically eliminating calls to these functions at compile-time, or providing a secure wrapper function for them, which carries out the necessary security checks at runtime. This is done with the means of a compiler extension, and is therefore completely transparent to the programmer.

When evaluating our compiler, we have seen that it successfully prevented against a variety of attacks that exploited vulnerabilities posed by the usage of insecure C library functions. Additionally, we have seen that our compiler was able to replace calls to insecure functions at compile-time with a good success rate. With regards to performance, we have seen that for some micro-benchmarks, when compiled with our compiler, they run significantly slower. However, when benchmarking a real-world application, no significant performance degradation was observed.
Acknowledgements

I would like to thank Dr. Zachary Anderson for his continuous support, valuable feedback, and for all the interesting discussions throughout this thesis. It has been a great experience extending my knowledge in the area of compiler extensions and program transformations, and how they can be applied to solve security-related issues. Additionally, I would like to thank Prof. Timothy Roscoe for his useful advice.
Contents

1 Introduction 1

2 Problem Description 3
   2.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   2.2 Insecure calls and their consequences . . . . . . . . . . . . . . . . . . . . . . . . 4
      2.2.1 Buffer overflow vulnerabilities . . . . . . . . . . . . . . . . . . . . . . . . 4
      2.2.2 Format string vulnerabilities . . . . . . . . . . . . . . . . . . . . . . . . 8
   2.3 Categorization of C library functions . . . . . . . . . . . . . . . . . . . . . . . . 9
      2.3.1 Category 1: Never secure . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      2.3.2 Category 2: Conditionally secure . . . . . . . . . . . . . . . . . . . . . . 10
      2.3.3 Category 3: Always secure . . . . . . . . . . . . . . . . . . . . . . . . . . 11
   2.4 Usage statistics of insecure functions . . . . . . . . . . . . . . . . . . . . . . . . 11

3 Solution: Secure C Library C Compiler (sclcc) 15
   3.1 Tracking of allocated memory regions . . . . . . . . . . . . . . . . . . . . . . . 15
   3.2 Program transformations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
      3.2.1 Tracking of local arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
      3.2.2 Tracking of global arrays . . . . . . . . . . . . . . . . . . . . . . . . . . 21
      3.2.3 Optimizations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
   3.3 Secure C Library (SCL) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
      3.3.1 Tracking of allocated heap memory . . . . . . . . . . . . . . . . . . . . . . 23
      3.3.2 Wrappers for threading functions . . . . . . . . . . . . . . . . . . . . . . . 26
      3.3.3 Secure wrappers for insecure functions . . . . . . . . . . . . . . . . . . . . 26
CONTENTS

3.4 Logging and error handling ........................................... 30

4 Evaluation ................................................................. 31
  4.1 Security ............................................................... 31
    4.1.1 Test cases ..................................................... 31
    4.1.2 Results ....................................................... 32
  4.2 Performance ......................................................... 32
    4.2.1 Micro-benchmarks .............................................. 32
    4.2.2 Macro-benchmark: Apache webserver ......................... 37
  4.3 Compilation time .................................................... 39
  4.4 Results of a large-scale application ............................... 41
    4.4.1 Compilation of Ubuntu 11.10 source packages ............... 41
    4.4.2 Optimization statistics .................................... 41

5 Related Work ............................................................. 43
  5.1 Operating system-level defenses .................................... 43
    5.1.1 Address space layout randomization (ASLR) .................. 43
    5.1.2 Data execution prevention (DEP) ............................. 44
    5.1.3 Heap protection .............................................. 44
  5.2 Compiler- and linker-level defenses ............................... 45
    5.2.1 Canaries ....................................................... 45
    5.2.2 Bounds checking .............................................. 46
    5.2.3 Control-flow integrity ...................................... 47
  5.3 Runtime library defenses .......................................... 48
    5.3.1 Libsafe ....................................................... 48
    5.3.2 Heapshield ................................................... 48
  5.4 Comparison of security features ................................... 49
  5.5 C11 Standard ........................................................ 50

6 Conclusion and future work ........................................... 53
  6.1 Summary ............................................................. 53
  6.2 Future work ........................................................ 54

References ........................................................................... 57
List of Tables

2.1 Functions in category *never secure* ........................................ 10
2.2 Detailed information about functions in category *never secure* ........ 10
2.3 Functions in category *conditionally secure* .............................. 11
2.4 Detailed information about functions in category *conditionally secure* . 13
2.5 Usage statistics of insecure C library calls in the Ubuntu 11.10 distribution. . . 14

3.1 Insecure functions and their secure equivalents considered in sclcc’s optimization phase .................................................. 22
3.2 Insecure functions and the replacements used by their wrappers ............ 29

4.1 Mean CPU-time and standard deviation for different concurrency levels of the synthetic micro-benchmark ................................. 33
4.2 The ptrdist benchmarks used, along with a description, indication of whether dynamic allocation is used, and the input used to run them .................. 35
4.3 Mean CPU-time and standard deviation of the ptrdist micro-benchmarks . . . 36
4.4 Mean response time and standard deviation for different numbers of simultaneous clients .......................................................... 38
4.5 Mean throughput and standard deviation for different numbers of simultaneous clients .......................................................... 38
4.6 Mean CPU-time and standard deviation of the CPU-time used for compilation of the ptrdist micro-benchmarks ................................. 40
4.7 Optimization statistics for the successfully built Ubuntu 11.10 source packages. . 42
4.8 Optimization statistics for Ubuntu 11.10 source packages, including partially built ones .......................................................... 42

5.1 Comparison of security features provided by sclcc and different approaches mentioned .......................................................... 49
5.2 Some of the C11 standard’s bounds-checked functions .......................... 50
5.3 Some of the C11 standard’s functions that dynamically allocate a needed buffer. 50
List of Figures

2.1 Stack layout for `vulnerable_func()` ............................................................ 5

3.1 Compilation and linking process of sclcc ....................................................... 16
3.2 Linked list whose nodes contain pointers to the thread-local red-black trees of
each thread ........................................................................................................... 18

4.1 Mean CPU-time and standard deviation of the synthetic benchmark .......... 34
4.2 Mean CPU-time of the synthetic benchmark relative to its gcc-compiled version .... 34
4.3 Mean CPU-time and standard deviation of the ptrdist benchmarks .......... 36
4.4 Mean CPU-time of the ptrdist benchmarks relative to their gcc-compiled version .... 37
4.5 Mean response time and standard deviation for the Apache benchmark ...... 38
4.6 Mean throughput and standard deviation for the Apache benchmark ...... 39
4.7 Mean CPU-time and standard deviation used for compilation of the ptrdist
benchmarks ........................................................................................................ 40
4.8 Mean CPU-time used by sclcc’s compilation of the ptrdist benchmarks relative
to the CPU-time used by gcc ........................................................................ 41
Listings

2.1 Example of a stack-based buffer overflow vulnerability ............................... 5
2.2 Header of a chunk ......................................................................................... 6
2.3 Unlinking macro ............................................................................................ 7
2.4 Example of a heap-based buffer overflow vulnerability ............................... 8
3.1 Structure type of a red-black tree node used to keep track of allocated memory . 18
3.2 func1(), declaring a buffer on the stack, before transformation .................... 20
3.3 func1(), declaring a buffer on the stack, after transformation ....................... 20
3.4 func2(), allocating a buffer on the stack using alloca(), before transformation 20
3.5 func2(), allocating a buffer on the stack using alloca(), after transformation . 21
3.6 Call to strcpy() before optimization ............................................................ 22
3.7 Call to strcpy() after optimization .............................................................. 22
3.8 Wrapper function for malloc() ................................................................. 23
3.9 Wrapper function for free() ................................................................. 24
3.10 Wrapper function for calloc() .............................................................. 24
3.11 Wrapper function for realloc() ............................................................. 25
3.12 Wrapper function for strcpy() ............................................................ 26
3.13 Function used to retrieve bounds information for a pointer to some memory address 27
4.1 Code executed by each individual thread of the synthetic micro-benchmark . . 33
Chapter 1

Introduction

The C programming language is the most popular language for systems programming and is used for the development of operating systems, embedded systems, web servers and many other projects that require low-level access to hardware. It is a traditional programming language still used nowadays for numerous projects: Out of the approximately 225 thousand projects on the open source portal SourceForge [7], as of 2012, about 31% of them are written in C/C++. C’s standard library has been developed in the early 1970s and has been refined ever since.

Unfortunately, due to the lack of security awareness at that time, multiple functions of the C standard library exhibit serious security issues. Nevertheless, usages of them can still be found in source code of recent projects, operating systems among them. This is the case in spite of the fact that for many of these functions, secure equivalents have been integrated into more recent versions of the C standard library. This could be the result of the lack of security awareness of programmers, or because the ratio of programmer time to security benefit is assumed too high when considering the task of replacing calls to insecure functions with calls to their secure equivalents by hand, an argument that especially applies to legacy code of larger projects.

While modern operating systems have features that help prevent well-known attacks that try to exploit vulnerabilities induced by the usage of insecure C library functions, a higher level of security can be attained by not using insecure functions in the first place. This is because, as we will see later, even though the defense mechanisms incorporated into current operating systems certainly help prevent these attacks, they only mitigate the risk but cannot guarantee the absolute impossibility of them due to various limitations. Additionally, the challenge of integrating current OS-level defenses into embedded systems, such as smartphones, is an area of active research [19].

Furthermore, even if an attack is detected by current OS-level defenses, and in many other defense mechanisms that we explore in chapter 5, it will either result in the termination of execution of the vulnerable program, or in a crash of it. However, a good solution to the problem of insecure library calls would not prevent an attack by just halting execution but by
blocking the attack while guaranteeing continued execution of the expected control-flow of the program.

This thesis introduces a compiler extension for the well-known GNU C Compiler [3] that either automatically eliminates calls to insecure functions, or redirects them to newly introduced secure wrapper functions. Automatic elimination is done if the C library provides a secure equivalent of the insecure function and the information necessary to call this equivalent can simply be inferred from the source code by static analysis. If this is not possible, calls to an insecure function are redirected to a secure wrapper function which will look up the necessary information in runtime data structures. This is all done transparently by the compiler, which results in a significantly higher level of security without any additional work by the programmer.

Programs compiled with our compiler are protected against library-based control-flow hijack attacks, without having their execution impeded in case of an attack. This is a practical side-effect of our solution, which current OS-level defenses are unable to achieve.

The thesis is structured as follows:

- In Chapter 2, we introduce some necessary definitions, describe the problem posed by calls to insecure functions of the C library, followed by a categorization of them according to their security property. We then summarize the results of an analysis that shows that there is still a high number of calls to them in current source code.

- In Chapter 3, we present our compiler extension solution in depth.

- In Chapter 4, we evaluate our solution in terms of multiple performance metrics, namely security, slowdown of compiled programs, and compilation time. Additionally, we present the success rate of compile-time optimizations of a large scale application.

- Chapter 5 presents related work and compares the various approaches to our solution.

- Chapter 6 concludes the thesis and discusses possible directions for future work.
Chapter 2

Problem Description

2.1 Definitions

We first introduce the following general definitions for the safety and security of functions. They are in accordance with Burn’s causal basis for defining the general terms safety and security [20], where the terms are distinguished by analyzing the causal consequence of a failure. A failure is any behavior that has an undesired effect, even if it complies with the program specification.

Safety: We say that a function is safe, if it cannot result in a failure that can do us immediate, direct harm.

Security: We say that a function is secure, if it cannot result in a failure that could enable, or increase the ability of, others to cause us harm.

Harm can include a deviation from the expected control-flow, manipulation of data stored in memory, leakage of confidential information, segmentation faults, but is not limited to these as it is a subjective matter of what is considered harm.

An example of a safety violation would be a function whose execution can result in a failure causing a segmentation fault without the interaction of an external attacker, that is the segmentation fault is the direct causal, and probably, but not necessarily, also temporal, consequence of the failure, whereas for a security violation, the segmentation fault has to be caused by the action of an external attacker, who was enabled to conduct it because of the direct causal consequence of the failure.

Under the above definition of security, any harm that can be caused by an external attacker would violate that definition. This thesis focuses on control-flow hijack attacks, though other sorts of attacks may also be prevented as a side-effect. Therefore, the general term security will be restricted to include only the harm of a deviation of the expected control-flow. The new definition reads as follows:
Security (restricted): We say that a function is secure, if it cannot result in a failure that could enable, or increase the ability of, others to hijack the control-flow of a program.

This new definition is a restricted form of the original one and we will stick to it throughout this thesis. It has to be noted here that this thesis focuses only on C-library based control-flow hijack attacks.

We introduce the following additional definitions describing the trustworthiness of a data source:

**Trusted source:** A trusted source is a data source, integrated into a program by the developer, or a component of the operating system, that is completely isolated from any external entity. That is, a user or attacker cannot manipulate the data emerging from it.

**Untrusted source:** An untrusted source is any other data source.

An example of a trusted source is a string compiled and linked into the data section of a program whereas an example of an untrusted source is a string supplied to the program as user input from the terminal, network, disk, etc..

### 2.2 Insecure calls and their consequences

An insecure call is a call to a function that violates our definition of security. When considering functions of the standard C library, there are two main problems a function might exhibit which causes the definition to be violated. The first one is the lack of bounds-checking, which creates a buffer overflow vulnerability and the second one is accepting specially crafted, harmful format strings which creates a format string vulnerability. The exploitation of both vulnerabilities can cause a deviation of a program’s expected control-flow, and therefore violates our definition of security. Both kinds of vulnerabilities will be explained in more detail next.

#### 2.2.1 Buffer overflow vulnerabilities

**Stack-based buffer overflows**

The most straightforward buffer overflow attack is the traditional stack smashing attack which is well documented [13]. In order to carry out this attack, an attacker overwrites the return address stored on the stack, for example by supplying a long string as input to the program which then passes it as argument to an insecure function, causing a stack-based buffer overflow and overwriting of the return address of the caller function if the buffer is stored on its stack. This enables an attacker to cause a targeted redirection of the expected execution when the function returns. Execution can either be redirected to an attacker-supplied piece of code, or to some other location within the program.
A simple textbook example of a stack-based buffer overflow vulnerability is shown in Listing 2.1. We assume that \texttt{str} originates from an untrusted source. Figure 2.1 shows the stack layout\footnote{Note that this is the general layout of an IA-32 stack frame. On the x86-64 architecture it might look a little different due to certain optimizations\cite{30}.} right before the call to \texttt{strcpy()}. Because \texttt{strcpy()} does not do bounds checking, an attacker can supply a string longer than the size of the buffer and overwrite the return address.

\begin{lstlisting}[language=C]
void vulnerable_func(char *str) {
    char buffer[10];
    strcpy(buffer, str);
}
\end{lstlisting}

Listing 2.1: Example of a stack-based buffer overflow vulnerability

Another kind of stack-based buffer overflow vulnerability is the ability of an attacker to overwrite the frame pointer stored on the stack\cite{28}. When doing so, the modified frame pointer will be popped from the stack and thereby written to the base pointer register instead of the original one in the epilogue of the vulnerable function, which is \texttt{vulnerable_func()} in our example. This way, some other address already stored at a known location on the stack can be used as return address for the function calling the vulnerable function when it writes the modified frame pointer from the base pointer register to the stack pointer register in its own epilogue before returning.

\textbf{Heap/.bss-based buffer overflows}

Another class of buffer overflow vulnerabilities are heap-based buffer overflows or overflows of buffers stored in the .bss segment which contains uninitialized global variables or variables declared as static. When an insecure function writes to the heap or to a buffer located in the .bss segment, the return address cannot directly be overwritten. However, many times function pointers or longjump-buffers are stored there. Therefore, similar to stack-based buffer overflows, an insecure function can be used to overwrite them which leads to a redirection of execution as soon as control-flow is redirected to the location specified by them\cite{22}.
Additionally, when overflowing a buffer on the heap, sophisticated attacks allow an attacker to overwrite `malloc()`-metadata which can lead to a control-flow hijack attack [14, 24, 27]. We present here a vulnerability found in earlier versions (versions prior to v2.3.5) of the dynamic allocator of the GNU C library [4]. While defense mechanisms have been integrated into allocators of modern operating systems against such attacks, as we will explore in Chapter 5, more complex but similar attacks are still carried out today.

The mentioned vulnerable allocator uses chunks to organize blocks of allocated and free memory. A chunk is comprised of header information, followed by the allocated or free memory block described by the header. The header used was of the form shown in Listing 2.2.

```c
struct malloc_chunk {
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    struct malloc_chunk *fd; // only used by free chunks
    struct malloc_chunk *bk; // only used by free chunks
};
```

Listing 2.2: Header of a chunk

The field `prev_size` describes the size of the physically preceding chunk, `size` describes the size of the current chunk. The header is included in the size. The next two fields are only used by free chunks. Each free chunk is a node in free list which is implemented as a cyclic doubly linked list. The fields `fd` and `bk` are pointers to the header of the next and previous chunk in the free list, respectively. Because of size alignment, the low order bit of `size` is called `PREV_INUSE` and is used to indicate whether the physically preceding chunk is allocated or free.

When a chunk is freed, `free()` checks whether the chunk physically following the chunk being freed is free (and is therefore in the free list). If this is the case, it will be unlinked from the free list and the chunks will be coalesced, resulting in a larger free chunk and a new entry in the free list. In order to perform unlinking, the macro in Listing 2.3 was used. It has three parameters, namely `P` which is a pointer to some chunk `C` which is to be unlinked and two temporary pointers `BK` and `FD`. We can see that unlinking is done by first assigning the `bk` pointer of the chunk following chunk `C` in the free list, the address of the one preceding it in the list, followed by assigning the `fd` pointer of the chunk preceding chunk `C`, the address of the one following it.
An attacker could now exploit a heap-based buffer overflow by overwriting header information of the chunk physically following the overflowed one, in the following way [27]:

- Set the size field to -$\text{sizeof}$(INTERNAL\_SIZE\_T)
- Set the low order bit of prev_size to zero
- Replace the fd pointer with a pointer to the return address or to a function pointer, minus $(\text{sizeof}(\text{INTERNAL\_SIZE\_T}) \times 2 + \text{sizeof}(\text{struct malloc\_chunk } *))$
- Replace the bk pointer with a pointer to the code to execute

Following these steps enables an attacker to redirect execution as soon as free() is called on the overflowed chunk. For example, consider the chunks A, B, and C physically following each other and chunk A is vulnerable to an overflow. Assuming no overflow takes place, the following would happen when free() is called on chunk A: it will be checked for coalescing purposes whether chunk B is free by looking whether the PREV\_INUSE bit of chunk C’s size field is set. In order to do that, the address of the header of chunk C is located by adding the value contained in the size field of chunk B’s header to the address of its header. If the bit is not set, chunk B is free and will be unlinked and coalesced with chunk A to form a larger chunk, as explained before. However, when chunk A is overflowed and the header of chunk B is modified in the above way during the overflow then, when free() is called on chunk A, it will be checked like before whether chunk B is free. However, when computing the address of chunk C’s header by adding -$\text{sizeof}(\text{INTERNAL\_SIZE\_T})$ to the address of chunk B’s header, we get a fake address for chunk C’s header which causes the PREV\_INUSE bit to be checked by looking at the prev_size field of chunk B’s header rather than at the size field of chunk C’s header. Because we set that bit to zero during the overflow, free() interprets chunk B to be free. Therefore, it will perform unlinking of it as specified in Listing 2.3. Doing so, it will overwrite the memory referenced by the fd pointer of chunk B’s header plus $(\text{sizeof}(\text{INTERNAL\_SIZE\_T}) \times 2 + \text{sizeof}(\text{struct malloc\_chunk } *))$ (which contains a function pointer or return address) with the address pointed to by pointer bk of chunk B’s header which was set to point to code of an attacker’s choice.
As for stack-based buffer overflows, we also give a simple example of a heap-based buffer overflow in Listing 2.4 [27] which allows a control-flow hijack attack when the described vulnerability is present in the dynamic allocator. Like before, we assume str originates from an untrusted source. Header information of the chunk containing buf2 can be overwritten as described above because data is copied to it by strcpy() which does not do bounds checking. As soon as buf1 is freed, a control-flow redirection can take place when unlinking is done for the chunk containing buf2. The attack works under the assumption that the chunk containing buf2 physically follows the chunk containing buf1.

```c
void vulnerable_func(char *str) {
    char *buf1 = malloc(10);
    char *buf2 = malloc(10);
    strcpy(buf1, str);
    free(buf1);
    free(buf2);
}
```

Listing 2.4: Example of a heap-based buffer overflow vulnerability

### 2.2.2 Format string vulnerabilities

When the format string parameter of the printf()-family, originates from an untrusted source, multiple attacks are possible. However, for this project, only one attack is of interest because it can be used to do a control-flow hijack. The format specifier %n, which causes the number of bytes output so far to be written to the memory location pointed to by its corresponding parameter can be used to achieve this. An attacker can supply a format string that contains carefully chosen format specifiers, in order to walk up the stack until a pointer pointing to an address of interest is located (for example the address of a function pointer, or the address of the saved frame pointer of the function two steps behind in the call stack, which can result in a control-flow hijack attack similar to the one described in Section 2.2.1), followed by the %n specifier which is finally used to overwrite the memory referenced by that pointer with an address of the attacker’s choice. Contrary to intuition, the generation of a targeted address is easy, even without supplying a very long string, for example by using %.500000x in a format string for printf(), which will cause the hexadecimal integer of the corresponding argument to be printed in such a way that the resulting string representation of it will be padded with leading zeros until its total length is 500000, which drastically increases the number of bytes written and therefore also the value written by the %n specifier.

Additionally, for the scanf()-family, if the format string and the input data originates from an untrusted source, an attacker can write any arbitrary values to memory addresses pointed to by the data stored on the stack. Similar to before, there are cases where this can lead to an overwrite of critical data, resulting in a control-flow hijack.
Now that we have seen different kinds of vulnerabilities, our goal is to prevent their exploitation by an attacker. While some defense mechanisms against these attacks have already been incorporated into modern operating systems, as we will further explore in Chapter 5, they have several drawbacks—a major one being the detection of overflows after the fact, followed by halting execution. In this thesis we are developing a tool that prevents attacks exploiting mentioned vulnerabilities induced by wrong usage of insecure C library functions, before the fact.

2.3 Categorization of C library functions

In this section, we categorize C library functions into the following three categories: *Never secure, conditionally secure*, and *always secure*.

In the first category, functions are listed that, no matter how they are called, always violate our definition of security.

In the second category, functions are listed that, depending on some conditions of how they are called, for example whether data originates from an untrusted source, violate our definition of security.

All remaining C library functions that never violate our definition of security belong to the third category, however, there are a few exceptions there, as some functions will be ignored. Exceptions include multiple wide character string processing functions that are equivalents of the ANSI string functions which will be listed in this section. The reason for ignoring them is the fact that, after computing usage statistics for them, as done for the other insecure functions (see Section 2.4), it was apparent that they have literally never been used in all of the C code of the Ubuntu 11.10 Linux distribution [9].

Additionally, it has to be noted here that one can argue that functions that do bounds checking in order to protect against buffer overflows might also belong to the second category because they could be passed an incorrect size parameter. In this project, we do not consider these cases as it is wrong use of such a function and a security hole created by carelessness of the programmer, rather than by the seemingly correct, though insecure use of a library function. Because we assume correct use of these functions, we place them in the third category.

2.3.1 Category 1: Never secure

In Table 2.1, functions belonging to category *never secure* are listed. Actually, we identified only one function that belongs to this category.
Table 2.1: Functions in category *never secure*

Table 2.2 shows detailed information about the function in category *never secure*. The table shows the purpose of the function as well as its problems that causes our definition of security to be violated. Lastly, it shows any condition necessary to induce the problem. Obviously, in this category, no such conditions are necessary.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Problem</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get()</code></td>
<td>Reads a string from the standard input and writes it to the buffer pointed to by <code>str</code>.</td>
<td>Since the function does not perform any bounds checking, the buffer pointed to by <code>str</code> might be overflowed.</td>
<td>No condition. Since the function writes data originating from an untrusted source to a bounded buffer, no call is ever secure.</td>
</tr>
</tbody>
</table>

Table 2.2: Detailed information about functions in category *never secure*

### 2.3.2 Category 2: Conditionally secure

In Table 2.3, functions belonging to category *conditionally secure* are listed.

Table 2.4 shows detailed information about the functions in category *conditionally secure*. Similar to the previous category, the table shows the purpose of the functions as well as its problems that causes our definition of security to be violated, along with any condition necessary to induce the problem.
Function

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>char * strcpy (char * destination, const char * source);</td>
</tr>
<tr>
<td>char * strcat (char * destination, const char * source);</td>
</tr>
<tr>
<td>int scanf (const char * format, );</td>
</tr>
<tr>
<td>int fscanf (FILE * file, const char * format, ...);</td>
</tr>
<tr>
<td>int sscanf (const char * source, const char * format, ...);</td>
</tr>
<tr>
<td>int vscanf (const char * format, va_list args);</td>
</tr>
<tr>
<td>int vfscanf (FILE * file, const char * format, va_list args);</td>
</tr>
<tr>
<td>int vsscanf (const char * source, const char * format, va_list args);</td>
</tr>
<tr>
<td>int sprintf (char * str, const char * format, ...);</td>
</tr>
<tr>
<td>int vsprintf (char * str, const char * format, va_list arg);</td>
</tr>
<tr>
<td>int wcscpy (wchar_t * destination, wchar_t source);</td>
</tr>
<tr>
<td>int wcscat (wchar_t * destination wchar_t source);</td>
</tr>
<tr>
<td>int printf (const char * format, ...);</td>
</tr>
<tr>
<td>int fprintf (FILE * stream, const char * format, ...);</td>
</tr>
<tr>
<td>int vprintf (const char * format, va_list arg);</td>
</tr>
<tr>
<td>int vfprintf (FILE * stream, const char * format, va_list arg);</td>
</tr>
<tr>
<td>int snprintf (char * str, size_t size, const char * format, ...);</td>
</tr>
<tr>
<td>int vsnprintf (char * str, size_t size, const char * format, va_list arg);</td>
</tr>
</tbody>
</table>

Table 2.3: Functions in category conditionally secure

2.3.3 Category 3: Always secure

All remaining C library functions do not violate our definition of security, or have not been considered for this project for reasons mentioned.

2.4 Usage statistics of insecure functions

In this section, we present statistics of the usages of the functions in the first two categories mentioned in Section 2.3. These statistics were obtained by analyzing the source code of the Ubuntu 11.10 Linux distribution [9]. For the analysis, only source packages that contained pure C code have been considered. In total, we counted 2102 packages, 932 of them were pure C packages, which is a proportion of about 44%. The C packages analyzed contained approximately 32 million lines of source code.

The number of calls represented have been obtained by counting the number of matches of the regular expression "[^a-zA-Z0-9function", where function has been replaced with the function name of all functions we were interested in. A regular expression of this kind matches a call to the function almost all of the time, therefore the statistics, albeit not exact, are very good approximations.

Out of all wide character string functions, only wcsncpy(), wcscat(), and swprintf() have been used. The latter already does bounds checking and was therefore ignored, along with the others that have never been used. Wrapper functions for wcsncpy() and wcscat() have
<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Problem</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcpy()</td>
<td>Copies the ANSI string pointed to by <code>source</code> to the buffer pointed to by <code>destination</code>.</td>
<td>Since the function does not perform any bounds checking, the buffer pointed to by <code>destination</code> might be overflowed.</td>
<td><code>source</code> originates from an untrusted source.</td>
</tr>
<tr>
<td>strcat()</td>
<td>Concatenates the ANSI string pointed to by <code>source</code> to the string pointed to by <code>destination</code>.</td>
<td>Since the function does not perform any bounds checking, the buffer pointed to by <code>destination</code> might be overflowed.</td>
<td><code>source</code> originates from an untrusted source.</td>
</tr>
<tr>
<td>[v]scanf()</td>
<td>Reads data from the standard input and stores them according to the format string specified by <code>format</code> into the locations pointed to by the additional arguments.</td>
<td>Since the function does not perform any bounds checking by default, the buffers pointed to by its arguments might be overflowed. Additionally, memory pointed to by data on the stack can be overwritten.</td>
<td>The format string originates from an untrusted source or string-type format specifiers are used. (Input always originates from an untrusted source.)</td>
</tr>
<tr>
<td>[v]fscanf()</td>
<td>Reads data from <code>stream</code> and stores them according to the format string specified by <code>format</code> into the locations pointed to by the additional arguments.</td>
<td>Since the function does not perform any bounds checking by default, the buffers pointed to by its arguments might be overflowed. Additionally, memory pointed to by data on the stack can be overwritten.</td>
<td>The format string originates from an untrusted source or string-type format specifiers are used, both while <code>stream</code> is an untrusted source.</td>
</tr>
<tr>
<td>[v]sscanf()</td>
<td>Reads data from <code>str</code> and stores them according to the format string specified by <code>format</code> into the locations pointed to by the additional arguments.</td>
<td>Since the function does not perform any bounds checking by default, the buffers pointed to by its arguments might be overflowed. Additionally, memory pointed to by data on the stack can be overwritten.</td>
<td>The format string originates from an untrusted source or string-type format specifiers are used, both while <code>str</code> originates from an untrusted source.</td>
</tr>
<tr>
<td>Function</td>
<td>Purpose</td>
<td>Problem</td>
<td>Condition</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>[v]sprintf()</td>
<td>Writes to a buffer pointed to by <code>str</code> an ANSI string consisting of a</td>
<td>Since the function does not perform any bounds checking by default, the buffer pointed to by <code>str</code></td>
<td>A string-type format specifier is used while its corresponding parameter originates from an untrusted source or the format string originates from an untrusted source.</td>
</tr>
<tr>
<td></td>
<td>sequence of data formatted as the <code>format</code> argument specifies.</td>
<td>might be overflowed.</td>
<td></td>
</tr>
<tr>
<td>wcscpy()</td>
<td>Copies the wide character string pointed to by <code>source</code> to the buffer</td>
<td>Since the function does not perform any bounds checking, the buffer pointed to by <code>destination</code></td>
<td><code>source</code> originates from an untrusted source.</td>
</tr>
<tr>
<td></td>
<td>pointed to by <code>destination</code>.</td>
<td>might be overflowed.</td>
<td></td>
</tr>
<tr>
<td>wcscat()</td>
<td>Concatenates the wide character string pointed to by <code>source</code> to the</td>
<td>Since the function does not perform any bounds checking, the buffer pointed to by <code>destination</code></td>
<td><code>source</code> originates from an untrusted source.</td>
</tr>
<tr>
<td></td>
<td>string pointed to by <code>destination</code>.</td>
<td>might be overflowed.</td>
<td></td>
</tr>
<tr>
<td>[v]printf()</td>
<td>Writes a string formatted as specified by the <code>format</code> argument to the</td>
<td>A control-flow hijack attack can be carried out using the `%n specifier.</td>
<td><code>format</code> originates from an untrusted source.</td>
</tr>
<tr>
<td>[v]fprintf()</td>
<td>standard input.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[v]snprintf()</td>
<td>Writes to a buffer pointed to by <code>str</code> an ANSI string consisting of a</td>
<td>A control-flow hijack attack can be carried out using the `%n specifier.</td>
<td><code>format</code> originates from an untrusted source.</td>
</tr>
<tr>
<td></td>
<td>sequence of data formatted as the <code>format</code> argument specifies. The</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>functions writes at most <code>size</code> bytes.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: Detailed information about functions in category *conditionally secure*
been implemented, even though their usage number is really low when compared to their ANSI equivalents. An interesting fact revealed by the analysis about format string functions is that the number of legitimate uses of the \texttt{%n} specifier is vanishingly small. For example, for the \texttt{printf}()-family, out of all calls where the format string was passed as string literal (which was the case for 92.3\% of calls), only a mere 65 calls contained this specifier, which is about 0.03\%. This fact will be considered in Section 3.3.3, when dealing with secure wrapper functions.

The full usage statistics can be seen in Table 2.5.

<table>
<thead>
<tr>
<th>Insecure function</th>
<th>Number of calls</th>
<th>Insecure function</th>
<th>Number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{printf}()</td>
<td>96932</td>
<td>\texttt{vprintf}()</td>
<td>212</td>
</tr>
<tr>
<td>\texttt{fprintf}()</td>
<td>82905</td>
<td>\texttt{vprintf}()</td>
<td>94</td>
</tr>
<tr>
<td>\texttt{sprintf}()</td>
<td>17354</td>
<td>\texttt{scanf}()</td>
<td>60</td>
</tr>
<tr>
<td>\texttt{strncpy}()</td>
<td>14511</td>
<td>\texttt{wcscpy}()</td>
<td>48</td>
</tr>
<tr>
<td>\texttt{snprintf}()</td>
<td>13487</td>
<td>\texttt{gets}()</td>
<td>44</td>
</tr>
<tr>
<td>\texttt{strcat}()</td>
<td>6406</td>
<td>\texttt{vsscanf}()</td>
<td>16</td>
</tr>
<tr>
<td>\texttt{sscanf}()</td>
<td>4096</td>
<td>\texttt{wscanf}()</td>
<td>13</td>
</tr>
<tr>
<td>\texttt{vscanf}()</td>
<td>701</td>
<td>\texttt{printf}()</td>
<td>8</td>
</tr>
<tr>
<td>\texttt{fscanf}()</td>
<td>686</td>
<td>\texttt{scanf}()</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{gets}()</td>
<td>433</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: Usage statistics of insecure C library calls in the Ubuntu 11.10 distribution.

As can be seen, there are a lot of calls to the potentially insecure functions, that is functions belonging to one of the first two categories identified in Section 2.3. This is even the case for functions which have secure equivalents, for example \texttt{strcpy()} which can be easily replaced by its secure equivalent \texttt{strncpy()}.

To summarize, we introduced definitions in order to distinguish between the terms safety and security. We then further restricted the definition of security to only include harm of a control-flow hijack. Additionally, we have seen different kinds of such control-flow hijack attacks, namely methods exploiting different kinds of buffer overflow vulnerabilities as well as a format string vulnerability.

Furthermore, during the categorization of C library functions, we have identified one function that is always insecure and further 18 functions that are potentially insecure. We have seen that a lot of these functions, although the C library provides secure equivalents for many of them, are still used in the wild. This conclusion could be drawn from our analysis of the source code of the Ubuntu 11.10 Linux distribution [9].

Knowing all these facts, it is clear that the level of security of C programs can greatly be improved by providing a solution to the problem of insecure C library functions. In the next chapter, we present our solution which prevents the mentioned control-flow hijack vulnerabilities induced by the careless usage of potentially insecure C library functions.
Chapter 3

Solution: Secure C Library C Compiler (sclcc)

In this chapter, we present our solution to the problem posed by calls to insecure C library functions, as described in Chapter 2. It consists of a compiler extension for gcc which does the following tasks:

- It does multiple program transformations to preprocessed source files before they are passed on to gcc for compilation. These transformations are implemented in the C Intermediate Language (CIL) [33].

- In the linking stage, object files are always linked against the SCL library which is sclcc’s library implementing secure wrappers for insecure C library functions, as well as other necessary runtime features.

Our compiler carrying out these tasks is called Secure C Library C Compiler or in short, sclcc. The compilation process is illustrated in Figure 3.1.

3.1 Tracking of allocated memory regions

The idea used by sclcc is the following: In the program transformation phase, our compiler tries to replace insecure functions with a secure equivalent, if it can infer the necessary information to do so. If not, the call is intercepted at runtime by a wrapper function implemented in the SCL library. For the wrappers, we maintain a mapping of valid memory regions. In practice, valid memory regions are divided among several mappings for efficiency reasons. Therefore, there are multiple instances of a suitable data structure holding memory bounds information for different memory allocations.
Firstly, there is one such data structure per thread which is set up as soon as a new thread is spawned and destroyed when the thread terminates. It is used to keep track of bounds information of memory regions on the thread’s stack.

Analogously, there is one such data structure per thread for heap memory allocated by a thread. Although one global instance of the data structure would be enough, because unlike the stack where each thread has its own private stack memory, the heap is accessible by all threads, we decided to use one per thread in order to reduce contention for the data structure in a multithreaded environment where multiple threads do heap memory allocation concurrently and try to register bounds information for the allocated block in the same shared data structure.

Additionally, there is one such global data structure holding memory bounds information for global data. When a program compiled with sclcc is run, global data is registered in this data structure at the beginning of execution, as later described in Section 3.2.2. Furthermore, this data structure is used in case a thread terminates without having freed all memory it allocated on the heap. In such a case, the bounds information will be transferred from the thread’s local data structure to this global one.

Lastly, there is a linked list whose elements are nodes containing pointers to the thread-local data structures of each thread. This is necessary to retrieve bounds information in case an address registered on some thread’s local data structure escapes to another thread. This can either be an escape through the heap, or when the thread creates another thread, passing
a memory address registered on one of its local data structures, to the newly created thread as an argument.

When an insecure function is called, the call is intercepted by sclcc’s wrapper function. The wrapper function looks for bounds information within these data structures. If bounds information is found, a bounds-checked, secure equivalent of the insecure function is called. If not, the action is determined by the error handling mode. The error handling modes are described in more detail in Section 3.4.

We chose a red-black tree to store the set of valid memory regions. A red-black tree is a binary tree with certain restrictions that cause it to always be balanced in such a way that the search, insert, and remove operations all have a worst-case time complexity of \( O(\log n) \), where \( n \) is the number of elements in the tree. To guarantee thread-safety, reader-writer locks have been used on a per-tree basis which allow multiple readers but only one writer at a time. Since each thread has its own local red-black trees, a reader is only blocked when it looks up bounds information for an address that is stored in some other thread’s red-black trees (while this thread owns a writer lock). This will only happen if the thread did not find bounds information in its own red-black trees, as will be seen in Section 3.3.3. This is assumed to be a rare case, as threads are expected to operate on their own stack memory and heap memory allocated by themselves. When accessing the globals red-black tree, a reader is only blocked when a thread terminates without having freed all memory it allocated on the heap, as then the terminating thread owns a writer lock in order to transfer the nodes of its heap red-black tree to the globals red-black tree.

Before the final decision of using red-black trees, we rejected to use splay trees. Although a splay tree has the practical property of having frequently accessed nodes move towards the top of the tree, this benefit comes at the cost of every read operation also being a write operation, which is impractical in a multithreaded environment. Additionally, we tried using a non-blocking and lock-free skip list. However, it performed much worse than our current solution, which might be the result of the skip list’s frequent usages of atomic operations.

The linked list mentioned before which is used by a thread to access the red-black trees of the other threads therefore consists of nodes containing pointers to both red-black trees of each thread. This is illustrated in Figure 3.2.

We used the implementation provided by the BSD library [2] for the linked list and red-black tree data structures. Listing 3.1 shows the structure type that has been used to keep track of allocated memory regions. The red-black trees store nodes of this type. The type \texttt{mem_field_t} is either a \texttt{uint32_t} or a \texttt{uint64_t}, depending on the architecture of the system. The base and size fields specify the base address and size of a registered memory region, respectively. Additionally, the \texttt{rbt_node_info} field was declared by the \texttt{RB_ENTRY} macro which expands to the definition of a structure containing node information (parent node, left child, right child, and color of the node) relevant to the red-black tree. Similarly, the \texttt{rbt_node_info} field declared by the \texttt{LIST_ENTRY} macro expands to the definition of a structure containing node information
Figure 3.2: Linked list whose nodes contain pointers to the thread-local red-black trees of each thread.

(previous and next element) of a linked list. The linked list is a free list for scecc’s own custom node allocator. The allocator was implemented for efficiency reasons in order to avoid later calls to dynamic memory allocation functions each time a memory address is registered in one of the red-black trees by the compiled program. When a thread is created, memory for a specified number of such nodes is preallocated and freed when the thread exits. A node within the memory of the custom allocator is either in a red-black tree, or in the free list of the allocator. When the free list of the custom allocator is empty, calls are redirected to `malloc()`. A node is allocated before the insert operation and freed after the remove operation.

```c
struct rbt_mem_node {
    // node information for the red-black tree
    RB_ENTRY(rbt_mem_node) rbt_node_info;

    // node information for the linked list used as the free list by the
    // custom node allocator
    LIST_ENTRY(rbt_mem_node) ll_node_info;

    // base address of the allocated region
    mem_field_t base;

    // size of the allocated region
    mem_field_t size;
};
```

Listing 3.1: Structure type of a red-black tree node used to keep track of allocated memory.
There are two different node compare functions. The first one just does a simple compare of the base addresses of the nodes and ignores the size field. The second compare function performs an inclusion test of the intervals defined by the (base, size)-pairs of the nodes. Equality holds if the inclusion test succeeds. In a case of inequality, the compare operation is redirected to the first compare function.

The insert and remove operation of the red-black tree use the first compare function when traversing the tree. Functions performing these operations are wrappers of heap memory management functions as well as functions that register or deregister stack memory or memory of global data. Therefore, the order of the nodes in a red-black tree is determined by the base address field of the nodes.

The search operation of the red-black tree uses the second compare function when traversing the tree. It is only used during the retrieval of bounds information for a pointer. The specialized compare function is necessary because the pointer might point somewhere in the middle of an allocated memory region. In order to perform bounds information lookup for a pointer, the search function is passed a node which is constructed by setting the base to the address pointed to by the pointer and the size to zero.

3.2 Program transformations

3.2.1 Tracking of local arrays

Sclcc keeps track of all local character and wide character arrays allocated on the stack. Since the critical parameters of all insecure functions listed in Section 2.3 is a pointer of type char or wchar_t, we only keep track of arrays of these types and ignore cases where a programmer might cast some address on the stack not declared as char or wchar_t array, to one of these types. If for some reason necessary, arrays of other types could be added to sclcc’s program transformation code easily.

In order to do the tracking of an array’s base address and size, our compiler iterates through the local variables of each function. If it finds an array of the mentioned types among these variables, it inserts a call to scl_register_stack_var() at the beginning of the function. This function inserts the memory range for the array into the red-black tree the thread maintains for its stack memory. An call to scl_deregister_stack_var() is inserted before each return statement. This function removes the item from the red-black tree. For example, Listing 3.2 shows a function that declares a local buffer and zeros it out before transformation, and Listing 3.3 shows the same function after transformation.
void func1(void) {
    char buf[10];
    memset(buf, '\0', sizeof(buf));
}

Listing 3.2: func1(), declaring a buffer on the stack, before transformation

void func1(void) {
    char buf[10];
    scl_register_stack_var((void*) buf, 10 * sizeof(char));
    memset(buf, '\0', sizeof(buf));
    scl_deregister_stack_var((void*) buf);
}

Listing 3.3: func1(), declaring a buffer on the stack, after transformation

In addition to tracking the arrays mentioned above, sclcc also keeps track of variables on the stack that have been allocated using alloca(). This function is passed a size parameter and allocates size bytes on the stack. Since using this technique larger regions on the stack can be allocated which might be passed as parameter to an insecure function, it makes sense to keep track of such memory allocations. In order to achieve this, the following transformations are necessary: First, a dummy variable is introduced to the function’s locals. The result of a call to alloca() has to be assigned to some variable, otherwise the base address of the allocated region is lost. The value of the variable to which this base address is assigned, is assigned to the dummy variable after the base address of the allocated region is inserted into the thread’s local red-black tree tracking stack memory. A call to remove the item, where the base address is referenced by the dummy variable, is inserted before each return statement of the function. In contrast to local arrays, for alloca(), we need to introduce a dummy variable because the original variable initially holding the base address might be assigned another value. For arrays, this is not necessary since attempts to assign the variable holding the base address will be flagged as a type error. Like above, Listing 3.4 shows a function which declares a local buffer using alloca() and zeros it out before transformation, and Listing 3.5 shows the same function after transformation.

void func2(void) {
    char *buf;
    buf = alloca(10);
    memset(buf, '\0', 10);
}

Listing 3.4: func2(), allocating a buffer on the stack using alloca(), before transformation
void func2(void) {
    char *buf;
    void *scl_tmp_var_0;

    scl_tmp_var_0 = (void *) 0;
    buf = (char *) alloca(10);
    scl_register_stack_var((void *) buf, 10);
    scl_tmp_var_0 = buf;
    memset(buf, '\0', 10);
    scl_deregister_stack_var((void *) scl_tmp_var_0);
}

Listing 3.5: func2(), allocating a buffer on the stack using alloca(), after transformation

3.2.2 Tracking of global arrays

Sclcc also tracks global char and wchar_t arrays. In order to do this, first a constructor function is added to the source file. A function which carries the attribute constructor is executed right at the beginning, even before the main function takes over control. After this, the program transformation code iterates through the globals of the source file and when it finds an array to track, analogously to local variables, it inserts an instruction into the body of the constructor function which causes an entry for the array to be inserted into the global red-black tree.

Of course, generally when talking about arrays, not only arrays that are defined as such are meant. Sclcc also supports arrays that are a field of a composite type, that is a structure or a union. Additionally, multidimensional arrays are also supported.

3.2.3 Optimizations

Many times, when bounds information can be inferred statically from the source code, calls to insecure functions can be replaced directly during the program transformation phase. Such optimizations can be made when the C library provides a secure equivalent of the insecure function. Calls that have been optimized like this will be more efficient during runtime since they skip the bounds information lookup procedure as they directly call a secure version of the original function. Sclcc has the ability to infer bounds information if the critical parameter of an insecure function is of the form array + expression, where expression causes some positive displacement within array.

In addition to replacing calls to insecure functions, sclcc also guarantees equal semantics of all functions with regards to null termination, namely that result strings are always null terminated. For example, strncpy() does not implicitly add a terminating null character where strcpy() would. Therefore, during the optimization phase, a call to strncpy() is replaced with
a call to `strncpy()`, followed by an instruction that guarantees null termination. The same applies to `wcsncpy()`. The wrapper of an insecure function has the same behavior as the resulting instructions of the optimizations of that function, with regards to null termination.

As example, Listing 3.6 shows a call to `strcpy()` that is insecure if `str` originates from an untrusted source. If `buf` is a `char` array of size `SIZE`, then the call would be optimized at compile-time and be replaced by sclcc with the two lines of code shown in Listing 3.7. The subtraction of zero in the first line means that there was no displacement within the array, otherwise the displacement would be subtracted. The second line writes a null to the end of the buffer. This is because `strncpy()` always pads the buffer with zeros except of the case where exactly `SIZE` bytes are written, in which sclcc takes over the task of guaranteeing null termination.

```c
strcpy(buf, str); // potentially insecure call
```

Listing 3.6: Call to `strcpy()` before optimization

```c
strncpy(buf, str, (SIZE * sizeof(char) - 1) - 0); // secure call
*((char *)(buf + (SIZE * sizeof(char) - 1))) = (char) 0; // terminating '0'
```

Listing 3.7: Call to `strcpy()` after optimization

Table 3.1 shows a list of insecure functions and their secure equivalents. In sclcc’s optimization phase, calls to these insecure functions can be replaced with their secure equivalent, if bounds information can simply be inferred statically. The statistics of successful optimizations for these functions will be presented in Section 4.4.2.

<table>
<thead>
<tr>
<th>Insecure function</th>
<th>Replacement function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gets()</code></td>
<td><code>fgets()</code></td>
</tr>
<tr>
<td><code>strcpy()</code></td>
<td><code>strncpy()</code></td>
</tr>
<tr>
<td><code>strcat()</code></td>
<td><code>strncat()</code></td>
</tr>
<tr>
<td><code>sprintf()</code></td>
<td><code>snprintf()</code></td>
</tr>
<tr>
<td><code>vsprintf()</code></td>
<td><code>vsnprintf()</code></td>
</tr>
<tr>
<td><code>wcscpy()</code></td>
<td><code>wcsncpy()</code></td>
</tr>
<tr>
<td><code>wcsat()</code></td>
<td><code>wcsncat()</code></td>
</tr>
</tbody>
</table>

Table 3.1: Insecure functions and their secure equivalents considered in sclcc’s optimization phase
3.3 Secure C Library (SCL)

3.3.1 Tracking of allocated heap memory

As mentioned in Section 3.1, sclcc keeps track of allocated heap memory regions in a thread-local red-black tree. In order to achieve this, sclcc overrides the C library’s original heap memory management functions, namely malloc(), free(), realloc(), and calloc(). All of these functions do the requested operation and update the red-black tree accordingly. In order for them to be able to call the original function, the SCL library has a constructor function that uses the dynamic linker to retrieve the address of the original function, which is then stored as a function pointer. Simplified versions of the wrapper functions are shown next. Listings 3.8, 3.9, 3.10, and 3.11 show pseudocode for the malloc(), free(), calloc(), and realloc() wrappers, respectively.

```c
void * malloc ( size_t size ) {
    struct rbt_mem_node node;
    void *addr = NULL;

    // call original malloc()
    addr = malloc_orig(size);

    if (addr) {
        node.base = (mem_field_t) addr;
        node.size = (mem_field_t) size;

        // insert node into the thread-local heap red-black tree
        rbt_mem_insert(rbt_heap, &node, &rwlock_rbt_heap);
    }

    return addr;
}
```

Listing 3.8: Wrapper function for malloc()
void free (void *ptr) {
    struct rbt_mem_node node;
    node.base = (mem_field_t) ptr;
    node.size = (mem_field_t) 0;

    // first try to remove it from the thread-local heap red-black tree
    // and in case of failure from the globals red-black tree
    if (!rbt_mem_remove(rbt_heap, &node, &rwlock_rbt_heap))
        rbt_mem_remove(rbtGlobals, &node, &rwlock_rbtGlobals);

    // call original free()
    return free_orig(ptr);
}

Listing 3.9: Wrapper function for free()

void *calloc (size_t num, size_t size) {
    struct rbt_mem_node node;

    void *addr = calloc_orig(num, size);

    if (addr) {
        node.base = (mem_field_t) addr;
        node.size = (mem_field_t) num * size;

        // insert node into the thread-local heap red-black tree
        rbt_mem_insert(rbt_heap, &node, &rwlock_rbt_heap);
    }

    return addr;
}

Listing 3.10: Wrapper function for calloc()
void * realloc (void *ptr, size_t size) {
    struct rbt_mem_node node;
    void *addr = NULL;

    // in case ptr == NULL, realloc() behaves like malloc()
    if (ptr == NULL)
        return malloc(size);

    // in case size is 0, realloc() behaves like free()
    if (size == 0) {
        free(ptr);
        return NULL;
    }

    // since the base address might be changed, we remove
    // the old red-black tree node and create a new one
    node.base = (mem_field_t) ptr;
    node.size = (mem_field_t) 0;

    // first try to remove it from the thread-local heap red-black tree
    // and in case of failure from the globals red-black tree
    if (!rbt_mem_remove(rbt_heap, &node, &rwlock_rbt_heap))
        rbt_mem_remove(rbt_globals, &node, &rwlock_rbt_globals);

    // call original realloc()
    addr = realloc_orig(ptr, size);

    if (addr) {
        node.base = (mem_field_t) addr;
        node.size = (mem_field_t) size;

        // insert the new node into the thread-local heap red-black tree
        rbt_mem_insert(rbt_heap, &node, &rwlock_rbt_heap);
    }

    return addr;
}

Listing 3.11: Wrapper function for realloc()
3.3.2 Wrappers for threading functions

Analogously to wrapping memory management functions, sclcc also implements wrappers for threading functions like `pthread_create()` and `pthread_exit()`. The purpose of these wrappers is to set up the thread’s local heap and stack red-black trees as well as inserting a node containing pointers to them into the linked list mentioned in Section 3.1 at thread creation. At thread termination, the red-black trees are freed and the node in the linked list corresponding to the terminating thread is removed. As mentioned in 3.1, if a terminating thread still has bounds information in its heap red-black tree, the nodes of the tree will be transferred to the globals red-black tree, before the thread’s local red-black tree is freed.

3.3.3 Secure wrappers for insecure functions

As mentioned before, sclcc implements a wrapper function for each insecure function. A wrapper function is called when no optimization was done. The wrapper function looks up bounds information for all critical parameters and then calls a secure, bounds-checked equivalent of the original insecure function. The function that retrieves the bounds information first checks the current thread’s local stack red-black tree. If it does not find the bounds information there, it checks the thread’s local heap red-black tree, followed by the globals red-black tree in case of failure. If the bounds-information is not found there, it iterates through the linked list containing the thread-local red-black trees of the other threads and tries to find bounds information in each one of them; first doing so for the red-black tree tracking heap memory and later for the one tracking stack memory.

Next, we give some simplified example code for the wrapper function of `strcpy()` in Listing 3.12, as well as for the bounds information retrieval procedure in Listing 3.13.

On line two, the `strcpy()` wrapper function first retrieves the number of remaining bytes of the buffer into which destination points to. A case where no bounds information is found is handled as specified by the error handling mode described in more detail in Section 3.4.

```c
char * strcpy (char *destination, const char *source) {
    mem_field_t num = retrieve_remaining_bytes(destination);
    if (num == 0) {
        // no bounds information found -> do error handling
    }
    // call safe equivalent
    strncpy(destination, source, num-1);
    // enforce null termination
    destination[num-1] = '\0';
    return destination;
}
```

Listing 3.12: Wrapper function for `strcpy()`
```c
mem_field_t retrieve_remaining_bytes(void *addr) {
    struct rbt_mem_node node, *pnode = NULL;
    mem_field_t ret = 0;

    node.base = (mem_field_t) addr;
    node.size = (mem_field_t) 0;

    // check if destination is on the current thread's stack red-black-tree
    if (! (pnode = rbt_mem_lookup(rbt_stack, &node, &rwlock_rbt_stack))) {
        // destination not there -> check the current thread's heap red-black-tree
        if (! (pnode = rbt_mem_lookup(rbt_heap, &node, &rwlock_rbt_heap))) {
            // destination not there -> check the globals
            if (! (pnode = rbt_mem_lookup(rbtGlobals, &node, &rwlock_rbtGlobals))) {
                // destination not on there -> check heap
                if (! (pnode = try_rbt_ll(LL_HEAP, &node))) {
                    // destination not on there -> check stack
                    if (pnode) {
                        ret = pnode->base + pnode->size - (mem_field_t) addr;
                    }
                } else {
                    pnode = try_rbt_ll(LL_STACK, &node);
                }
            } else {
                // red-black trees of other threads
                if (! (pnode = try_rbt_ll(LL_STACK, &node))) {
                    // red-black trees of other threads
                    pnode = try_rbt_ll(LL_STACK, &node);
                } else {
                    // destination not on there -> check stack
                    if (pnode) {
                        ret = pnode->base + pnode->size - (mem_field_t) addr;
                    }
                }
            }
        } else {
            // red-black trees of other threads
            if (! (pnode = try_rbt_ll(LL_HEAP, &node))) {
                // red-black trees of other threads
                pnode = try_rbt_ll(LL_STACK, &node);
            } else {
                // destination not on there -> check stack
                if (pnode) {
                    ret = pnode->base + pnode->size - (mem_field_t) addr;
                }
            }
        }
    }

    return ret;
}
```

Listing 3.13: Function used to retrieve bounds information for a pointer to some memory address.
On line 9, `strncpy()` , the secure bounds-checked equivalent of `strcpy()` is called, followed by an instruction on line 12 that guarantees null termination.

The bounds retrieval function, `retrieve_remaining_bytes()` shown in Listing 3.13, first declares a red-black tree node whose base is set to the address for which we want to carry out the bounds lookup and the size is set to zero. The fact that the lookup operation of the red-black tree uses the inclusion compare function, as mentioned in Section 3.1, allows us to do this and also guarantees us to find bounds information registered in the tree even when the address passed to the retrieval function points in the middle of some allocated region. Then, on line 10, first the thread’s local red-black tree tracking memory of the thread’s stack is checked, followed by the thread’s local red-black tree which registered memory on the heap allocated by the thread. On line 16, the function continues to look for bounds information in the global red-black tree that registered global arrays and also contains bounds information for heap memory allocated by threads that terminated already without freeing the heap memory allocated by them. Lastly, on line 20 and 23, first the heap red-black tree of the other threads is checked followed by their stack red-black tree. The function stops its search as soon as the bounds information has been found and returns the number of remaining bytes that can be written to the buffer passed for lookup, or 0 if no bounds information has been found.

Almost all wrappers of the insecure functions listed in Section 2.3 have the same design, except of the `scanf()`- and `printf()`-family.

For the former ones, the wrapper function first verifies that the number of arguments passed to the function by the programmer is equal to the number of format specifiers, in order to guarantee the impossibility of overwriting memory pointed to by data on the stack other than memory pointed to by the supplied arguments, even when the format string is untrusted. After this, it iterates through all parameters corresponding to a format specifier and checks whether the format specifier is a string-type specifier, that is, `%[1]s`, or `%[1][`, since these specifiers could result in an overflow of the buffer pointed to by their corresponding parameter. Then, the bounds information is retrieved for these parameters and the format string is sanitized by adding a width specifier for each of these format specifiers. After that, the original function is called but with the newly sanitized format string, preventing an overflow.

For the latter ones, our wrapper function performs an early return if it finds a `%n` specifier, followed by a call to a bounds checked version for `[v]sprintf()` only.

It has to be noted here that handling the `%n` with the means of an early return can be justified by the fact that the number of legitimate uses of this specifier is negligible, as mentioned in Section 2.4. Additionally, in Chapter 5 we will see that the new C11 standard handles this specifier similarly.

An additional note is that for `[v]sprintf()`, sclcc might not protect against all instances of `%n` attacks, because in order to do so, every call of `[v]sprintf()` would need to be redirected to its wrapper function, eliminating its optimization stated in Section 3.2.3. Since we handle the `%n` problem out of completeness but not out of necessity (as can be seen from the statistics.
in Section 2.4), we decided here that this is one of the rare cases where an optimization is more important than completeness.

In any case, it would probably be a good idea to totally eliminate the \%n specifier from the standard for the whole `printf()`-family of functions because as it is so rarely used, removing it has a vanishingly small cost in programmer time, but a very large security benefit. Actually, as we will see in Chapter 5, this has been done in the C11 standard.

Table 3.2 summarizes the insecure functions listed in Section 2.3, along with the replacement function used by their wrapper.

<table>
<thead>
<tr>
<th>Insecure function</th>
<th>Replacement function used by wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>gets()</td>
<td>fgets()</td>
</tr>
<tr>
<td>strcpy()</td>
<td>strncpy()</td>
</tr>
<tr>
<td>strcat()</td>
<td>strncat()</td>
</tr>
<tr>
<td>[v]f</td>
<td>s)scanf()</td>
</tr>
<tr>
<td>sprintf()</td>
<td>snprintf(), with an early return if %n is found in the format string</td>
</tr>
<tr>
<td>vsprintf()</td>
<td>vsnprintf(), with an early return if %n is found in the format string</td>
</tr>
<tr>
<td>wcscpy()</td>
<td>wcsncpy()</td>
</tr>
<tr>
<td>wcscat()</td>
<td>wcsncat()</td>
</tr>
<tr>
<td>[v]f)printf()</td>
<td>original function, with an early return if %n is found in the format string</td>
</tr>
<tr>
<td>[v]snprintf()</td>
<td>original function, with an early return if %n is found in the format string</td>
</tr>
</tbody>
</table>

Table 3.2: Insecure functions and the replacements used by their wrappers
3.4 Logging and error handling

Logging and error handling features have been incorporated into sclcc.

The former one helps when debugging since it outputs warnings and errors related to sclcc during runtime, either to the standard output or to a log file. For logging, verbose mode can be switched on which causes other information like thread creation and termination, heap and stack memory bounds information that is registered and deregistered in a red-black tree to also be logged. Sclcc’s logging feature can be switched on by setting the environment variable `SCL_LOG` to either `STDOUT` for standard output logging or to `FILE:logfile.log` for file logging. Similarly sclcc’s verbose mode can be switched on by setting the environment variable `SCL_VERBOSE=ON`.

Using sclcc’s error handling environment variable `SCL_ERR`, it can be configured what is done in case of an error that occurs when a wrapper function of an insecure function is called but no bounds information is found. Sclcc provides two options to which this environment variable can be set, namely either `CALL_UNSAFE` which causes the original insecure function to be called, or `FORCE.Quit` which causes the program to terminate immediately.

By default, error handling is set to `CALL_UNSAFE`, logging does not take place and verbose mode is switched off.
Chapter 4

Evaluation

4.1 Security

In order to evaluate the security-layer introduced to programs compiled with sclcc, we came up with a variety of simple test-cases that probe the different kinds of security features provided by sclcc.

4.1.1 Test cases

Included in the tests were cases of library-based buffer overflows, where the buffer was

- declared as either single- or multi-dimensional array,
- declared as either global or local,
- declared as a field in a (potentially nested) structure type,
- a named type,
- or any possible combination of the above.

Additionally, buffers allocated by the following different memory allocation methods have been tested, namely buffers allocated

- on the stack using `alloca()`,
- on the heap using the `malloc()`-family of functions,
- by one of the above methods but for a variable size buffer, that is its size cannot be inferred until allocation time.
The buffer was then passed to functions in category *never secure* or *conditionally secure*. The tests have been carried out in a multi-threaded environment which was set up by spawning multiple threads trying to exploit a certain vulnerability in parallel.

Additionally, we tested inter-thread address transfers, that is a buffer was allocated by a thread on its stack or on the heap and was then passed to another thread which in turn passed it to an insecure library function.

Furthermore, in addition to buffer overflow test cases, the `%n` format specifier was passed with and without a conversion specifier to a variety of relevant format string functions in an attempt to overwrite the return address on the stack.

### 4.1.2 Results

When evaluating the mentioned test cases, sclcc performed as expected. It successfully prevented all buffer overflow- and format string-based control-flow hijack attacks from the above test cases, while allowing the expected program control-flow to continue, just as it was designed to do.

### 4.2 Performance

As described in Chapter 3, sclcc introduces multiple overheads to compiled programs, namely the overhead of memory allocation instrumentation, as well as the overhead of performing security checks. Therefore, it would be interesting to measure the performance penalty introduced to sclcc-compiled programs. We first carried out some micro-benchmarks measuring the slowdown in terms of CPU-time and then a macro-benchmark measuring the throughput and response time of the well-known Apache webserver [1].

For all our benchmarks, our testbed was a 4-socket, 64-core AMD Opteron 6276 running at 2.3 GHz having 256 GB of main memory in four banks running the Linux 3.2.2 kernel. The storage system was a 2-disk RAID array.

#### 4.2.1 Micro-benchmarks

We first performed a very synthetic micro-benchmark which consisted of a simple program that creates multiple threads, each one looping through `malloc()`, `strcpy()`, and `free()` procedure, as shown in Listing 4.1. This benchmark shows the worst-case performance penalty caused by sclcc, as all the tasks it does involve an overhead introduced by sclcc, namely dynamic memory allocation and runtime bounds checking. We measured CPU-time for different concurrency levels. For sclcc, we used two different versions. For one version, security checks are skipped. That is, the wrapper functions have been disabled. Therefore, the resulting overhead is caused mainly by memory allocation instrumentation which is an overhead interesting to quantify by itself. For the other, security checks are enabled. In order to allow for a fair
comparison, all of gcc’s security features, which are explained in more detail in Chapter 5, have been turned off. It has to be noted here that the CIL-framework itself did not introduce any significant performance degradation. The raw data of the results is shown in Table 4.1 and Figure 4.1 (note the log-scale y-axis). Additionally, Figure 4.2 shows the CPU-time of the micro-benchmark compiled with both versions of sclcc, relative to the CPU-time of a gcc-compiled version.

We can see that our synthetic micro-benchmark runs significantly slower when compiled with sclcc. The slowdown factors are similar for different concurrency levels in spite of sclcc’s thread creation and termination overhead described in Section 3.3.2. The mean slowdown factor for the version of sclcc not performing security checks ranges from 3.2 to 7.1 with a mean of 5.4, while the mean slowdown factor for the version performing them ranges from 5.3 to 12.4 with a mean of 8.8. Therefore, the mean slowdown factor was about 1.6 times higher for the version of sclcc performing security checks.

```c
void * thread_code(void *args)
{
    int i;
    char *c;
    for (i = 0; i < 1000000; i++) {
        c = malloc(1024);
        strcpy(c, "test string");
        free(c);
    }
    pthread_exit(NULL);
}
```

Listing 4.1: Code executed by each individual thread of the synthetic micro-benchmark

<table>
<thead>
<tr>
<th># Threads / Compiler</th>
<th>gcc</th>
<th>sclcc without checks</th>
<th>sclcc with checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>132.5 ± 4.4</td>
<td>418.0 ± 16.4</td>
<td>698.0 ± 8.9</td>
</tr>
<tr>
<td>2</td>
<td>254.0 ± 7.5</td>
<td>1340.5 ± 415.5</td>
<td>1978.5 ± 456.1</td>
</tr>
<tr>
<td>4</td>
<td>533.0 ± 47.8</td>
<td>2975.5 ± 923.0</td>
<td>4432.5 ± 619.3</td>
</tr>
<tr>
<td>8</td>
<td>1040.0 ± 39.7</td>
<td>5713.0 ± 801.3</td>
<td>9284.0 ± 833.3</td>
</tr>
<tr>
<td>16</td>
<td>2096.5 ± 40.0</td>
<td>14953.5 ± 1069.4</td>
<td>26098.5 ± 1876.3</td>
</tr>
<tr>
<td>32</td>
<td>8815.5 ± 533.7</td>
<td>51431.0 ± 5482.8</td>
<td>88124.5 ± 5482.8</td>
</tr>
<tr>
<td>64</td>
<td>59190.0 ± 7970.2</td>
<td>302887.5 ± 30504.4</td>
<td>514462.0 ± 58281.1</td>
</tr>
</tbody>
</table>

Table 4.1: Mean CPU-time and standard deviation for different concurrency levels of the synthetic micro-benchmark
Figure 4.1: Mean CPU-time and standard deviation of the synthetic benchmark

Figure 4.2: Mean CPU-time of the synthetic benchmark relative to its gcc-compiled version
In addition, we performed some non-synthetic micro-benchmarks. We used the ptrdist benchmarking suite [15] which contains micro-benchmarks that perform numerous complex memory operations such as pointer arithmetic and dynamic memory allocation and deallocation. Table 4.2 shows the five benchmarks we used, along with a short description, indication of whether they use dynamic memory allocation, and the input we used to run them.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Computation</th>
<th>Dynamic allocation</th>
<th>Input used</th>
</tr>
</thead>
<tbody>
<tr>
<td>anagram</td>
<td>An anagram generator</td>
<td>Yes</td>
<td>anagram words &lt; input.in$^1$</td>
</tr>
<tr>
<td>backprop</td>
<td>A neural net simulator</td>
<td>No</td>
<td>backprop</td>
</tr>
<tr>
<td>bc</td>
<td>GNU bc calculator</td>
<td>Yes</td>
<td>bc &lt; primes.b$^2$</td>
</tr>
<tr>
<td>ft</td>
<td>Minimum spanning tree computation</td>
<td>Yes</td>
<td>ft 10000 20000</td>
</tr>
<tr>
<td>ks</td>
<td>Kernighan-Schweikert graph partitioning</td>
<td>Yes</td>
<td>ks KL-4.in</td>
</tr>
</tbody>
</table>

1 input.in was fed to the program 20 times for one benchmarking run.

2 A limit of 8000 was used for primes.b.

Table 4.2: The ptrdist benchmarks used, along with a description, indication of whether dynamic allocation is used, and the input used to run them.

We measured CPU-time of the five benchmarks. Like before, we used a version of sclcc performing security checks and one version skipping them. Again, all of gcc’s security features have been turned off. The raw data of the results is shown in Table 4.3 and Figure 4.3. Additionally, Figure 4.4 shows the CPU-time of the benchmarks compiled with both versions of sclcc, relative to the CPU-time of a gcc-compiled version.

We can see that some of these micro-benchmarks run significantly slower when compiled with sclcc. For these benchmarks, there was no significant difference between the two versions of sclcc. For both versions, the mean slowdown factor for these benchmarks ranges from about 1 to almost 5 with a mean of approximately 2.5. The very high slowdown factor of some of these benchmarks are probably caused by frequent calls to dynamic memory allocation functions.
**Table 4.3: Mean CPU-time and standard deviation of the ptrdist micro-benchmarks**

<table>
<thead>
<tr>
<th>Benchmark / Compiler</th>
<th>gcc</th>
<th>sclcc without checks</th>
<th>sclcc with checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>anagram</td>
<td>$81.4 \pm 5.0$</td>
<td>$344.2 \pm 5.0$</td>
<td>$356.2 \pm 18.3$</td>
</tr>
<tr>
<td>backprop</td>
<td>$760.2 \pm 1.4$</td>
<td>$855.8 \pm 5.0$</td>
<td>$799.4 \pm 2.4$</td>
</tr>
<tr>
<td>bc</td>
<td>$126.4 \pm 6.3$</td>
<td>$621.6 \pm 6.8$</td>
<td>$624.4 \pm 6.7$</td>
</tr>
<tr>
<td>ft</td>
<td>$755.2 \pm 5.0$</td>
<td>$951.4 \pm 19.3$</td>
<td>$939.2 \pm 18.9$</td>
</tr>
<tr>
<td>ks</td>
<td>$2067.2 \pm 14.6$</td>
<td>$2007.6 \pm 9.2$</td>
<td>$2010.0 \pm 4.5$</td>
</tr>
</tbody>
</table>

**Figure 4.3: Mean CPU-time and standard deviation of the ptrdist benchmarks**
4.2.2 Macro-benchmark: Apache webserver

For our macro-benchmark, we compiled the multithreaded Apache v2.2.16 webserver [1] and measured the throughput and response time for both, a gcc-compiled version and an sclcc-compiled version. For each version, we measured both metrics for different numbers of simultaneous clients that repeatedly sent GET requests to the server. Again, all of gcc’s security features have been turned off. We did not distinguish between sclcc’s different overheads in this experiment. The raw data of the results are shown in Tables 4.4 and 4.5. Plots of the data are shown in Figures 4.5 and 4.6. Note the logarithmic scale on the x-axis.

Since the graphs more or less overlap for both metrics, we see that there was no significant performance degradation in terms of throughput and response time. This was in spite of the fact that Apache calls dynamic memory allocation functions when serving client requests – probably because next to the task of processing the requests, the calls to dynamic memory allocation functions become negligible. This suggests, that for some real-world applications, compiling them with sclcc is useful as the application may benefit from an increased level of security without significant performance degradation.
Table 4.4: Mean response time and standard deviation for different numbers of simultaneous clients

<table>
<thead>
<tr>
<th># Clients / Compiler</th>
<th>gcc</th>
<th>sclcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2933 ± 0.0000</td>
<td>0.2968 ± 0.0000</td>
</tr>
<tr>
<td>2</td>
<td>0.3296 ± 0.0005</td>
<td>0.3282 ± 0.0005</td>
</tr>
<tr>
<td>4</td>
<td>0.5024 ± 0.0018</td>
<td>0.4955 ± 0.0020</td>
</tr>
<tr>
<td>8</td>
<td>0.6246 ± 0.0015</td>
<td>0.6255 ± 0.0017</td>
</tr>
<tr>
<td>16</td>
<td>1.2330 ± 0.0023</td>
<td>1.2346 ± 0.0028</td>
</tr>
<tr>
<td>32</td>
<td>2.4590 ± 0.0154</td>
<td>2.4577 ± 0.0151</td>
</tr>
<tr>
<td>64</td>
<td>4.8817 ± 0.0761</td>
<td>4.8928 ± 0.0644</td>
</tr>
</tbody>
</table>

Table 4.5: Mean throughput and standard deviation for different numbers of simultaneous clients

<table>
<thead>
<tr>
<th># Clients / Compiler</th>
<th>gcc</th>
<th>sclcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3366.00 ± 9.66</td>
<td>3312.00 ± 16.87</td>
</tr>
<tr>
<td>2</td>
<td>6100.00 ± 42.16</td>
<td>5996.00 ± 33.73</td>
</tr>
<tr>
<td>4</td>
<td>7910.00 ± 71.34</td>
<td>7956.00 ± 27.97</td>
</tr>
<tr>
<td>8</td>
<td>12654.00 ± 16.47</td>
<td>12532.00 ± 310.58</td>
</tr>
<tr>
<td>16</td>
<td>12798.00 ± 6.32</td>
<td>12792.00 ± 19.32</td>
</tr>
<tr>
<td>32</td>
<td>12800.00 ± 0.00</td>
<td>12748.00 ± 74.36</td>
</tr>
<tr>
<td>64</td>
<td>12800.00 ± 0.00</td>
<td>12786.00 ± 44.27</td>
</tr>
</tbody>
</table>

Figure 4.5: Mean response time and standard deviation for the Apache benchmark
4.3 Compilation time

As described in Chapter 3, sclcc does multiple transformations to the source code during compilation. Therefore compilation time is an interesting metric to measure. We measured CPU-time used by sclcc and gcc for compilation of the five benchmarks of the ptrdist benchmarking suite [15] we used before. All of gcc’s compile-time security features have been turned off. The raw data of the results is shown in Table 4.6 and Figure 4.7. Additionally, Figure 4.8 shows the CPU-time used by sclcc for compilation of the individual benchmarks relative to the CPU-time used by gcc.

We can see that the slowdown factor of the CPU-time used for compilation by sclcc is similar for the individual benchmarks. On average, sclcc’s compilation takes 1.8 times longer in terms of CPU-time.
Table 4.6: Mean CPU-time and standard deviation of the CPU-time used for compilation of the ptrdist micro-benchmarks

<table>
<thead>
<tr>
<th>Benchmark / Compiler</th>
<th>gcc</th>
<th>sclcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>anagram</td>
<td>290.0 ± 0.0</td>
<td>590.0 ± 9.3</td>
</tr>
<tr>
<td>backprop</td>
<td>330.0 ± 0.0</td>
<td>537.5 ± 4.6</td>
</tr>
<tr>
<td>bc</td>
<td>2661.3 ± 3.5</td>
<td>3812.5 ± 23.1</td>
</tr>
<tr>
<td>ft</td>
<td>532.5 ± 4.6</td>
<td>1052.5 ± 20.5</td>
</tr>
<tr>
<td>ks</td>
<td>360 ± 0.0</td>
<td>617.5 ± 7.1</td>
</tr>
</tbody>
</table>

Figure 4.7: Mean CPU-time and standard deviation used for compilation of the ptrdist benchmarks
4.4 Results of a large-scale application

4.4.1 Compilation of Ubuntu 11.10 source packages

We carried out a comprehensive experiment by applying sclcc to all of the pure C code of the Ubuntu 11.10 Linux distribution [9]. Out of the 932 source packages that contain purely C code, we were able to build 284 without any major adjustments. Therefore, the resulting build success rate was approximately 30.47%. This is because the build scripts do not respond well to a compiler other than plain gcc, or linking with libraries they do not expect.

4.4.2 Optimization statistics

We were able to analyze the optimization statistics from all successfully built source packages. They are shown in Table 4.8. The low number of total calls when compared to Table 2.5 suggests that the successfully built packages did contain few calls to insecure functions. When looking at the total number of optimized calls in proportion to the total number of insecure calls, the resulting optimizations success rate is 43.73%, which is quite good, especially given our unsophisticated static analysis.

Table 4.7 shows the optimization statistics when including packages that have not been built successfully. That is, packages that have been compiled only partially, or in which linking failed.
We can see that although we used a really simple form of static analysis, the success rate for sclcc’s compile-time optimizations is quite good. In cases of successful optimization, there will not be any significant performance impact due to the low cost of bounds checking, as the size of the buffer is already known and therefore the secure function is called directly.
Chapter 5

Related Work

5.1 Operating system-level defenses

As mentioned in Chapter 1, modern operating systems help preventing the exploitation of mentioned vulnerabilities. However, they do not completely eliminate them. The two main defenses incorporated into modern operating systems are address space layout randomization (ASLR) and data execution prevention (DEP). Each of these will be explained in more detail next.

5.1.1 Address space layout randomization (ASLR)

The basic idea of ASLR is to randomize the location of objects in memory [31]. This is done by randomizing parts of the virtual address space of a process, for example the location of the heap and the stack. Therefore, memory addresses of objects of the process are unpredictable. Since the address of the stack is different with each execution of the program, an attacker does not know the address of code injected by him when exploiting a stack-based buffer overflow vulnerability. Therefore, even when overwriting the return address, the vulnerable program crashes instead of executing the injected code. The same applies to heap-based buffer-overflow attacks.

Limitations of ASLR

Since ASLR causes a vulnerable program to crash when exploited, it basically transforms control-flow hijack attacks into denial-of-service attacks. Therefore, it makes sense to use a crash detection and reactivation system along with ASLR [31]. This increases fault-tolerance of the vulnerable program which is critical for programs that need to guarantee high availability, for example web servers. However, the benefit of fault-tolerance comes at a cost of a lower security level. As pointed out by Shacham [39], it is possible to carry out a brute-force attack.
by trying to exploit a vulnerability multiple times until the attack succeeds. On 32-bit systems, where only 16 bits are randomized, such an attack is definitely feasible and is expected to succeed after \(2^{16} = 65536\) attempts, which is doable in a matter of minutes. A case study of compromising a self-restarting Apache webserver was successful. Shacham points out that the most promising defense against such an attack would be to upgrade to a 64-bit architecture, which fortunately is the standard now.

An additional weakness of ASLR is the fact that in reality not all code is randomized [38]. A reason for this is that on some systems, some programs run significantly slower when their location is randomized. Another reason is that some code is intentionally left unrandomized because of backwards compatibility as some old programs might expect a certain piece of code to be loaded at the same address every time. Furthermore, on some systems, programs have to be compiled in a way that they become location independent in order for their virtual address space to be randomized upon execution. Unfortunately, not all programs are compiled this way.

Small amounts of unrandomized code can pose a serious security risk [38]. Even though an attacker cannot execute code injected by him when exploiting a vulnerability, he can reuse chunks of code that are unrandomized. Many times, a small amount of unrandomized code allows for serious attacks. Schwartz [38] mentions, that it is possible to call any library function, with any arguments, even when a program did not link to it, with a probability of 80%, using only 100 KB of unrandomized code.

5.1.2 Data execution prevention (DEP)

DEP enforces memory to be either executable or writable, but not both [38]. This way, any code injected to a program by an attacker must be in a writable memory region and is therefore not executable. When the program counter is set to the address of this code, a segmentation fault occurs and the program crashes, rather than executing the supplied code.

Limitations of DEP

A limitation of DEP is that an attacker can still carry out a return-to-libc attack [34], where the attacker causes a redirection of control flow to a function within libc. Furthermore, using this technique, it is possible for an attacker to call `mprotect()` to make the stack executable and then jump to his injected code [38]. This has actually been done in real-world exploits. Although ASLR helps protecting against this, it is still possible to use small parts of unrandomized code to do this task, as mentioned.

5.1.3 Heap protection

In order to protect against heap-based overflows, techniques such as heap canaries [37] and safe unlinking [29] have been incorporated into modern operating systems [24].
An implementation of heap canaries causes each chunk to start with a random canary value. An overflow is detected by checking the canary value before a redirection of execution using methods similar to the one presented in Section 2.2.1 can take place.

Safe unlinking is the deployment of pointer integrity checks. For example, among other checks, a check of \( p \to fd \to bk == p \to bk \to fd == p \), where \( p \) is a pointer to a chunk to be unlinked, is performed since version 2.3.5 of glibc [4] right before unlinking a chunk. Safe unlinking can be bypassed in some cases [35, 36].

Because of the limitations of these OS-level defenses and because of their ability to only detect attacks followed by the action of halting execution, sclcc provides functionality complementary to them that guarantees the continuation of the expected program control flow in case of a control-flow hijack attack that results from a call to an insecure C library function.

### 5.2 Compiler- and linker-level defenses

#### 5.2.1 Canaries

The basic idea of canaries is to place a random canary word on the stack before the return address. This is done in the prologue of a function. In the epilogue, right before the return statement is processed, a check is performed to see whether the canary word has been altered which would be the case if a stack-based buffer overflow is exploited. If it is, the program halts. This approach has first been implemented in StackGuard [23], and was integrated into gcc v2.7.2.2 in 1998.

In the current version of gcc (v4.6.1), the Pro Police stack smashing protector [6] is used. It is based on the idea of StackGuard but also does a variety of other security optimizations, for example regorganizing local variables, so that pointers are placed before buffers. In order to compile a program using ProPolice, the `fstack-protector` argument can be supplied to gcc, but this is done by default in the current version of the compiler (tested on v4.6.1).

A similar approach to StackGuard is StackShield [8]. It is not really a canary-based approach but closely related. StackShield creates an additional stack, called the shadow stack, and uses it to make a safe copy of the return address of the function. In the prologue of a function, the return address is copied to the shadow stack, and in the epilogue, the actual return address is compared with the stored one. If they are not equal, a stack-based buffer overflow is detected.

While canaries on the stack are able to detect stack-based buffer overflows, they are not able to detect modifications to local variables stored on the stack before the return address that might be critical variables, such as function pointers. Additionally, they halt program execution at detection. Similar to the OS-level defenses, we think of sclcc as a complementary tool.
5.2.2 Bounds checking

The C programming language provides the programmer with the full power of pointers. Arbitrary pointer arithmetic can be performed which can lead to read and write operations to uninitialized or freed memory, which is mainly a safety issue. However, pointer arithmetic leading to illegal memory writes might also be a security issue since this is what essentially happens when buffer overflows are exploited. Therefore, the approaches listed here handle safety issues as well as security issues.

Purify [25] enables runtime checking of illegal memory accesses by inserting checking logic for each memory access directly into the object file. This is done by marking each byte of the stack, heap, data and .bss sections as either unallocated, uninitialized, or initialized. The main weakness of Purify is that it does not catch abusive pointer arithmetic which yields a pointer into a valid memory region but which is not the intended referent [26].

In order to overcome this drawback, Jones and Kelly [26] suggest an alternative approach: For each allocated object, descriptors such as the base address and limit are maintained. This way, a pointer can be mapped to a descriptor of an allocated object. When performing pointer arithmetic, a pointer is not allowed to escape the object it pointed to initially. Splay-trees are used to map pointers to object descriptors and perform these checks at runtime. Stack and heap objects are registered and deregistered when their memory is allocated or freed, respectively. Since multiple insecure C library functions do not come with bounds checking, they have been replaced with bounds-checked versions.

Further recent research in this area was the development of CCured [32] which adds metadata to certain pointers, resulting in a fat pointer holding the value of the pointer along with its bounds. An alternative approach which does not change pointer representation is Deputy [21] which allows programmers to use annotated pointer types that help the compiler check the safety of most pointer operations. Both approaches involve static and runtime checking of pointer arithmetic.

Another bounds checking approach is Baggy Bounds Checking [12]. The problem with dynamic approaches is that in order to perform a memory bounds lookup at each pointer reference they have to store the boundaries used for lookup in complex data structures, for example splay-trees. This makes the approach highly inefficient. Baggy bounds checking is a recently developed approach which performs dynamic bounds checking very efficiently by using a special memory allocator enforcing the allocation sizes to be a power of two. Because of this, a single byte is sufficient to store the binary logarithm of the allocation size and can be stored in a simple table. Additionally, the base address of an allocated block can be retrieved by clearing the \( \log_2(s) \) least significant bits of any pointer to the allocated block, where \( s \) is the size of the allocated block. Because of padding and alignment, it can efficiently be checked by a table lookup and using bitwise operations whether newly derived pointers exceed the bounds of the block they pointed to initially. A drawback of this approach is that because the size of the buffers allocated is probably most of the time far away from the next power of two, it might...
result in a significant space overhead. Additionally, because it checks the bounds of padded allocated blocks and not of objects, it relaxes the bounds checking precision which might lead to a high rate of false negatives.

In recent versions of gcc (for this project, v4.6.1 was used) the new environment variable, \textsc{DFORTIFY\_SOURCE} has been introduced. If set, it causes the introduction of different kinds of compile-time and run-time protection mechanisms for insecure C library functions [10]. When an insecure function is called, it does bounds checking if the buffer size is known and aborts the program if it overflows the buffer. Additionally, it stops \%n attacks in format string functions. \textsc{DFORTIFY\_SOURCE} can be set to a level between zero and two; zero switching it completely off, one to add protection mechanisms that do not affect the behavior of the compiled program and two to add more checks that might affect the compiled program [5]. In contrast to sclcc, this gcc extension does not register buffers that are allocated at runtime when their size is not specified until then. This is true for both, stack and heap buffers. Additionally, no bounds checking takes place for buffers that are exchanged between threads. It seems that the \textsc{DFORTIFY\_SOURCE} feature of gcc does a static analysis including a data flow analysis to infer buffer sizes in order to do bounds checking at compile-time and runtime, which would explain this drawback.

Out of these approaches, sclcc best compares to gcc’s \textsc{FORTIFY\_SOURCE} extension. However, as mentioned, there are cases where this feature does not adequately protect against control-flow hijack attacks. This is because buffer sizes are assumed to be computed statically, at compile-time only. Additionally, this feature, although it does provide wrapper functions for insecure C library functions, when a bounds-check fails, the program aborts, while sclcc would look up bounds information and call a secure equivalent. The other approaches mentioned would also protect against buffer overflows but primarily handle cases of safety-issues which was not the focus of this thesis.

5.2.3 Control-flow integrity

Another interesting concepts used to protect against buffer-overflow attacks is the idea of enforcing a very basic safety property, Control-Flow Integrity (CFI) [11]. This property dictates that software execution must follow a path of a predefined Control-Flow Graph (CFG), that is the execution path must not deviate from this CFG at any time. The CFG can be defined either by static analysis, for example by source-code or binary analysis, or by dynamic analysis, for example by execution profiling. The authors use static binary analysis and implement CFI by machine-code rewriting that instruments software with runtime checks which ensure that the control flow remains within a predefined CFG. They conclude that CFI enforcement is practical to implement on modern processors, is compatible with most existing software, and has little performance overhead. From a security viewpoint, CFI is effective against very powerful adversaries that have full control over the entire memory of the executing program, which is certainly possible when exploiting buffer-overflow vulnerabilities.
5.3 Runtime library defenses

5.3.1 Libsafe

Libsafe [17] is a runtime library that can be loaded into any process. It protects against C-library based stack smashing attacks [13] by intercepting calls to insecure C library functions and redirecting them to its own wrappers. Once such a function is called, Libsafe first checks whether the insecure function would write data to an address past the one pointed to by the frame pointer. If this is the case, the program is terminated.

An extension to Libsafe is LibsafePlus [16]. Its main extension is that it does not only protect against stack-based buffer overflow attacks but also against heap-based buffer overflow attacks. In order to do this, LibsafePlus intercepts calls to heap memory management functions in order to keep track of the locations and sizes within a runtime data structure, namely a red-black tree. Additionally, size information of local variables and other objects can be extracted by TIED (Type Information Extractor and Depositor) if the program is compiled in debug mode. This allows for more fine-grained bounds checking than Libsafe. Similar to Libsafe, wrapper functions are implemented for insecure functions and checks are performed before the insecure function is called. If a check fails, the program is terminated.

The extended version of Libsafe, LibsafePlus, incorporates a similar approach as sclcc when it comes to tracking dynamically allocated memory. However they are only runtime libraries and do not do compile-time transformations. Some drawbacks of this is that they cannot track variable size stack buffers, for example allocations using alloca(), or do compile-time optimizations like sclcc. Additionally, thread-safety of the approach is not clear. Programs compiled by sclcc are not only thread-safe – sclcc was actually designed for use in a multithreaded environment.

5.3.2 Heapshield

In order to prevent C-library based heap buffer overflows, HeapShield [18] uses a BiBOP memory management mechanism to keep track of allocated block sizes. The memory allocator divides memory into individual chunks that are a multiple of the system page size (hence the name BiBOP, which means Big Bag of Pages). In addition to dividing the heap into pages, it is required that all blocks in the same chunk have the same size. Therefore, objects of different sizes are segregated from each other as they will be stored in different chunks. The size of the object and other metadata of a chunk is either stored at the beginning of the chunk or in a separate page directory table. By aligning the chunks to page boundaries, the allocator can locate the metadata in constant time. A remainingSize() function is defined which takes a pointer as an argument and returns the number of bytes from the pointer up to the end of the allocated block. This is done by a page directory table lookup in order to read the size of the allocated block. The pointer offset within this block is then subtracted from its size and re-
turned. In order to prevent heap-based buffer overflows, for each C-library function vulnerable to buffer overflows, a wrapper function is defined which, after it verifies that the pointer given as argument is on the heap, first calls `remainingSize()`, and then calls a secure equivalent of the requested function. The computed remaining size is passed as an argument to that function.

Using a custom allocator for heap memory that does not involve tracking of the size of allocated blocks in a secondary data structure and allows bounds lookup in constant time is certainly an interesting idea. It would be interesting to measure the memory overhead introduced by Heapshield’s BiBOP allocator approach, which might result in fragmentation, to the memory overhead introduced by sclcc’s red-black trees.

### 5.4 Comparison of security features

In this section, we give a quick overview of the security features provided by the most relevant approaches mentioned in this chapter, along with the ones provided by sclcc. Table 5.1 indicates for the different approaches whether they detect an attack posed by a certain vulnerability triggered by an insecure C-library function and how they handle it.

<table>
<thead>
<tr>
<th>Protection / Vulnerability</th>
<th>Heap-based buffer overflow</th>
<th>Stack-based buffer overflow</th>
<th>%n attack</th>
<th>Overflow of global buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fixed size buffer</td>
<td>Fixed size buffer</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Stack canaries</td>
<td>X</td>
<td>✓1</td>
<td>✓1</td>
<td>✓</td>
</tr>
<tr>
<td>Heap canaries</td>
<td>✓1</td>
<td>✓1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Libsafe</td>
<td>✓1</td>
<td>✓1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LibsafePlus</td>
<td>✓1</td>
<td>✓1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Heapshield</td>
<td>✓1</td>
<td>✓1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gcc’s FORTIFY_SOURCE</td>
<td>✓1,4</td>
<td>✓1,4</td>
<td>✓1,4</td>
<td>✓1,4</td>
</tr>
<tr>
<td>sclcc</td>
<td>✓2</td>
<td>✓2</td>
<td>✓2</td>
<td>✓2</td>
</tr>
</tbody>
</table>

1 Detection after the fact, usually followed by aborting execution.
2 Detection and prevention before the fact, not impeding program control-flow.
3 Detection before the fact, without handling it. Results in abortion of execution.
4 Does not detect all overflows, assumingly the ones where data-flow analysis fails, as mentioned in Section 5.2.2.
5 No fine-grained size tracking. Always considers the maximum size of the buffer as the difference between the starting address of the buffer and the frame pointer of the corresponding stack frame.

Table 5.1: Comparison of security features provided by sclcc and different approaches mentioned
We can see that sclcc is the only approach where in all cases, an attack is detected and prevented before the fact and program control-flow is not impeded. This is an advantage, in addition to the one that sclcc is the only approach doing compile-time optimizations. It has to be mentioned here though that this table is only relevant when the vulnerability is triggered by an insecure C library function. If not, sclcc would not provide any protection, while some of the other approaches will.

5.5 C11 Standard

One of the main features of the new C11 standard is more language support to prevent against buffer overflows. These features include the presence of secure equivalents of insecure functions that help prevent buffer overflows using multiple mechanisms, namely the enforcement of bounds checking and other runtime constraints [40], as well as dynamic memory allocation by the function itself to guarantee a buffer of a big enough size [41]. To illustrate C11’s replacement functions, examples of the prototypes of the functions replacing gets(), strcpy(), and printf() are shown in table 5.2.

```
char *gets_s(char *s, rsize_t n);
errno_t strcpy_s(char * restrict s1, rsize_t s1max, const char * restrict s2);
int printf_s(const char * restrict format, ...);
```

Table 5.2: Some of the C11 standard’s bounds-checked functions.

As can be seen, the first two function prototypes have been augmented to include a size parameter which is used for bounds checking. For printf_s() (and all new functions of the printf()-family), the standard states that one of the runtime constraints is that the format string may not include the %n specifier. If a runtime constraint is violated, the function stops producing output. Therefore, it seems that our early return approach described in Section 3.3.3 in justifiable since the new standard incorporates a similar mechanism. Similar secure equivalents of originally insecure functions can be found in the C11 specification [40].

As mentioned, in addition to bounds checked functions, the new standard also defines some functions that dynamically allocate a big enough buffer [41]. Again, some function prototypes using this mechanism are shown in Table 5.3.

```
ssize_t getline(char **lineptr, size_t *n, FILE * stream);
int vasprintf(char **restrict ptr, const char *restrict format, va_list arg);
int fscanf(FILE *restrict stream, const char *restrict format, ...);
```

Table 5.3: Some of the C11 standard’s functions that dynamically allocate a needed buffer.
The getline() function can be used instead of gets() and allocates the buffer dynamically, preventing buffer overflows. The address of the dynamically allocated memory region is then written to the location pointed to by lineptr. Similarly, vasprintf() can be used instead of vsprintf(). For fscanf(), the prototype did not change but an additional feature was introduced: If, after the % sign, the assignment-allocation character m follows, and we are dealing with a string conversion specifier, the buffer for the resulting string is allocated dynamically by the function. The address of the resulting buffer is then written to the corresponding argument which is either of type char ** (or wchar_t **, if the length modifier l has been used).

While both approaches definitely help preventing buffer overflows, one has to hope that these new functions will actually be used and also integrated into legacy code. Unfortunately, as mentioned in Section 2.4, even though secure equivalents of some insecure functions have been available for some time in previous versions of the C standard, their insecure version has still been used in the wild.

A drawback of the C11 functions that dynamically allocate their buffers should be noted: Since they return the address of the allocated memory region and no longer care about it, it is the programmer’s task to free this space. Therefore, usages of these functions are great for security but might increase the number of memory leaks if they are used carelessly. It can also be expected that identifying memory leaks caused by such functions is harder than identifying regular memory leaks caused by the usage of memory management functions.
Chapter 6

Conclusion and future work

6.1 Summary

In the course of this thesis, we have developed a solution for the problem posed by insecure C library functions.

Before this, we first distinguished between the terms safety and security, followed by a restriction of the term security to only include vulnerabilities posed by insecure C library function that allow a control-flow hijack attack. We have seen different kinds of examples for such attacks, namely the exploitation of stack-based or heap-based buffer overflow vulnerabilities, or the exploitation of a format-string vulnerability. We then performed a categorization of C library functions, putting each in one of the categories never secure, conditionally secure, or always secure and have seen that functions in the former two categories are still used in the wild, in spite of the fact that for many of them, newer versions of the C library provide secure equivalents for them.

Our solution was the development of a compiler extension for gcc which consisted of two parts.

The first part was the development of program transformation code that carries out transformations to the source code at compile-time. These transformations include direct replacement of insecure functions with secure ones where possible, and the insertion of calls to register and deregister bounds information of buffers for runtime checking. The CIL [33] framework has been used for this.

In the second part, the Secure C Library was developed, which provided multiple runtime features required by sclcc-compiled programs. These include wrappers for dynamic memory management functions in order to register and deregister bounds information at runtime, as well as wrapper functions for insecure C library functions that are used in cases where a call could not be optimized during the program transformation phase at compile-time.

During evaluation, we have seen that sclcc performs well in terms of security as it was able to prevent all control-flow hijack attacks it was designed to prevent. Unlike many approaches
listed in Chapter 5 that only detect attacks and act by halting program execution in case of an attack, normal program control-flow was able to continue for attacks performed on scelcc-compiled programs which is due to the secure wrapper functions provided by scelcc.

In regards to performance, we have seen that scelcc causes slowdowns to compiled programs due to multiple performance overheads, such as memory allocation instrumentation and security checks. By carrying out benchmarking experiments, we have seen that for some micro-benchmarks, the slowdown factor was significant. However, when benchmarking a real-world application, namely the multithreaded Apache webserver, we have not noticed any significant performance degradation in terms of throughput and response time.

During a large-scale application of scelcc carried out by attempting to compile all pure C source packages of the Ubuntu 11.10 Linux distribution [9], we have seen that scelcc was able to compile about 30% of source packages and was able to optimize about 43.73% of calls to insecure C library functions. When also including partially compiled packages, the number was similar.

When comparing scelcc to other approaches mentioned in Chapter 5, we have seen that it acts complementary to current OS-level defenses such as address space layout randomization (ASLR) and data execution prevention (DEP), since both of them exhibit significant drawbacks. We have seen that the use of canaries is widely deployed by modern compilers. While they certainly detect buffer overflows, the only thing they do at detection is halting the vulnerable program’s execution. The same is true for heap canaries implemented in memory allocators of modern operating systems. Scelc on the other hand, tries to keep program execution intact and is able to do so, if a vulnerability is caused by a call to an insecure C library function. Scelcc can best be compared to other compile-time or runtime bounds-checking approaches, such as the new FORTIFY_SOURCE source feature of gcc, or the LibsafePlus runtime library. We have also seen how scelcc compares in terms of security to various approaches mentioned in Chapter 5.

Lastly, we have given an overview of the different features introduced by the new C11 standard which were designed to prevent library-based buffer-overflow attacks.

6.2 Future work

Possible future work can include compile-time data-flow analysis which is expected to improve the success rate of compile-time optimizations. Additionally, a method to exclude tracking of buffers that are never passed to an insecure function would certainly help in terms of efficiency, especially for buffers dynamically allocated on the heap.

Also, with regards to compile-time optimizations, it would be interesting to count the number of lines of code that are represented by the 284 successfully compiled source packages mentioned in Section 4.4.2. Additionally, improving scelcc’s successful package building rate would definitely be a goal to pursue.

The automatic generation of patch files for source files during compilation would also be a
useful feature. Using this feature, one can have sclcc generate patch files for the compile-time optimizations that can be sent upstream to the developers of software, making their code more secure.

Furthermore, an approach can be integrated into the program transformation code that registers individual structure fields for structures dynamically allocated on the heap. Sclcc tracks heap memory on a per-block basis only, which protects against attacks similar to the one presented in Section 2.2.1 but not against an attack where a structure containing an array followed by a function pointer is allocated dynamically and the function pointer is overwritten by an insecure function to which the base address of the array was passed as an argument.

Additionally, an interesting idea for future work is to use a custom allocator similar to Heapshield [18] to track allocated heap memory regions, which allows for more efficient bounds lookup.

Lastly, trying to extend sclcc to not only handle security issues but also safety issues would be an interesting task. Since sclcc has the ability to infer bounds information statically, or dynamically during runtime, compile-time optimizations and wrapper functions for C library functions could be implemented that are prone to memory access errors when used wrongly. This would introduce further slowdown which will be traded for an additional layer of fault-tolerance.
References

[14] Anonymous. Once upon a free(). Phrack, 9[57], August 2001. 6
REFERENCES


[22] M. Conover. w00w00 on heap overflows. January 1999. 5


REFERENCES


[34] Nergal. The advanced return-into-lib(c) exploits. Phrack, 11[58], December 2001. 44


[38] E. J. Schwartz. The danger of unrandomized code. Usenix, 36[6]:7–12, December 2011. 44

