EAI Endorsed Transactions
on Security and Safety

Research Article **ICST.ORG**

Separated Control and Data Stacks to Mitigate Buffer Overflow Exploits

Christopher Kugler[1] and Tilo Müller[1,*]

[1]Department of Computer Science, Friedrich-Alexander-University of Erlangen-Nuremberg, Germany

Abstract

Despite the fact that protection mechanisms like *StackGuard*, *ASLR* and *NX* are widespread, the development on new defense strategies against stack-based buffer overflows has not yet come to an end. In this article, we present a novel compiler-level protection called *SCADS: Separated Control and Data Stacks* that protects return addresses and saved frame pointers on a separate stack, called the *control stack*. In common computer programs, a single user mode stack is used to store control information next to data buffers. By separating control information from the *data stack*, we can protect sensitive pointers of a program's control flow from being overwritten by buffer overflows. To substantiate the practicability of our approach, we provide SCADS as an open source patch for the LLVM compiler infrastructure. Focusing on Linux and FreeBSD running on the AMD64 architecture, we show compatibility, security and performance results. As we make control flow information simply unreachable for buffer overflows, many exploits are stopped at an early stage of progression with only negligible performance overhead.

Keywords: Stack-based Buffer Overflows, LLVM, Separate Control Stack

Received on 02 September 2014; accepted on 05 January 2015; published on 05 October 2015

doi:10.4108/eai.5-10-2015.150477

1. Introduction

Security in software has a long history. After one of the most prominent examples of malware, the *Morris Worm* [16, chapter: "The Internet Worm"], gained a lot of media attention, steadily more and more exploits have been developed to abuse several vulnerabilities, present in frequently used programs. As a counter measure, scientists began to develop patches and protection mechanisms, to be integrated into programs, to fix vulnerabilities or at least make it as unfeasible as possible for attackers, to abuse them any further. This triggered a long-lasting race, with the task for both of the two sides (attackers and defenders) to be the first to explore a new or old vulnerability and attack or fix it before the opponent gains knowledge about it. This race is still ongoing today, now that security in the IT sector is even more important than ever before. And until today, one of the most frequently abused vulnerabilities, already part of the attack strategy of the Morris Worm, is still not fixed satisfactorily. The talk is of *buffer overflows*. Buffer overflows are a vulnerability in programs that is caused by missing range checks on buffers, which makes them an inherent problem of the used programming language.

C is one of the languages that does not perform any range checks on buffers by specification, thus it is not supporting the prevention of buffer overflows all by itself. According to the *TIOBE-index* [28], C is still the most commonly used programming language as of 2014, which may be the main reason why buffer overflows are so dangerous and well-known, even in modern software. Of course, in the history of IT security, many attempts have been made to eliminate the vulnerability of buffer overflows. Though the developed defensive schemes either were not able to protect against all consequences of a buffer overrun, or they had other disadvantages, like a non-negligible performance overhead.

Protection schemes that focus on incorporating a posteriori boundary check for buffers (against the language's specification), lower the performance of programs written in C severely, and are therefore not the favored protection mechanism of choice, because C as a programming language is mostly used because of its well-known high performance features. Some supporting protection concepts like *Data Execution Prevention (DEP)* [18] and *Address Space Layout Randomization (ASLR)* [26] share the idea of not protecting the overrunning buffers at all, but instead target to enfeeble any exploits that abuse buffer overflows, by reducing the amount of options an exploit has, as soon as the redirection of the program's control flow has occurred. But history once again has proven, that this strategy also is not the answer to everything. The invention of DEP, although it was a very important progress to be made in the field of IT security, has spawned attack schemes such as *Return Oriented Programming (ROP)* [23, 25], which is able to bypass the protection offered by ASLR and DEP to successfully exploit a program.

[*]Corrsesponding author. Email: timuller@cs.fau.de

The main reason why ROP can be successful is the fact that neither DEP nor ASLR protect the *Return Instruction Pointer (RIP)* from modification, which is the most popular target of exploits for control flow manipulation. A rather obvious conclusion to this situation is that a better strategy may be to protect the RIP and equals, hence to use a security mechanism which sets in at a temporal stage of protection between DEP/ASLR and boundary checking of buffers. The approach we present within this article is based on just this idea. The protection of *control flow data* like the RIP can be maintained in many different ways, and our concept is not the first to take up this strategy. Nevertheless, our approach is defined by the way it addresses this task, by introducing a second stack to the generally single stack of a runtime environment. The two stacks, namely the *control stack* and the *data stack*, are supposed to manage different types of data without mixing them, to fulfill the protection of control flow data, especially the RIP. For this reason the defensive concept we introduce is given the name *Separated Control and Data Stacks (SCADS)*.

2. Design and Implementation

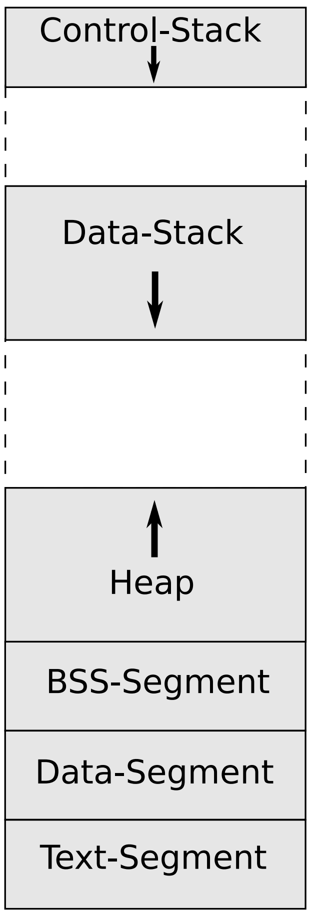


Figure 1. Userland memory layout of a program compiled with SCADS.

SCADS is based on the idea of *separating* the data which is stored on the stack of a process. It shall not be possible for an attacker to manipulate and redirect a program's control flow after a buffer overflow occurred. To achieve this separation, we introduce a second stack, the so called *data stack)*. While the remaining, native stack of the unified model now shrinks in size and becomes the so called *control stack*. For the purpose of preventing misunderstandings, we denote the original stack of the non-SCADS runtime environments as *unified stack*.

In consequence to the twofold stack usage, the process memory layout experiences a few changes. Figure 1 roughly indicates the new memory layout that is accompanied with the usage of SCADS. The memory segment of the unified stack is now occupied by the control stack, and in contrast to the original memory layout there now exists an additional memory segment between the heap and the control stack: the data stack. We further describe the exact location of the data stack in the forthcoming sections, but for now we want to mention that the control stack cannot collide with the data stack when it grows downwards, as both the control and the data stack are fully able to grow automatically.

Both of the newly introduced stacks manage mutually exclusive types of data. To clarify what kind of data is stored on each of the stacks, we classify two types of data that can be identified on the unified stack:

Control Flow Data This type of data is responsible or at least capable of managing the control flow of the program. The most important example of control flow data is the return instruction pointer, which is stored on the stack every time a function is called. Common exploits often modify the RIP, to redirect the control flow to pre-delivered shellcode or other source of malicious code.

Regular Data This data type is everything but control flow data. It includes all data which is necessary for computations but does not influence the control flow, e.g. buffers. Moreover local variables and saved data registers are considered to be of the type of regular data. Regular data, especially buffers, are often used by common exploits to gain the ability to modify nearby control flow data.

The idea of separation originates in the aspiration to construct a runtime environment, which is immune to overwriting control flow data such as the RIP, consecutive to a previously enforced abuse of regular data, for example through a buffer overflow. A detailed analysis regarding this immunity is realized in section 4. It is crucial that no data stored on the data stack gives any hints on the location of the control stack, which is already inherently achieved by the separation of the two before mentioned data types. Moreover, it is necessary to prevent the detection of the control stack location by brute force attacks. Therefore the location of the data stack is partially randomized and implicitly with it also the gap between the control and data stack. The purpose of a non-static offset between the control and the data stack requires both of the two stacks to be referenced by a register, which allows access to the stack pointers (and frame pointers if used) as known from the unified stack.

Listing 1: Exemplary assembly code that indicates which of the two stacks is influenced by a corresponding machine instruction (green: control stack; yellow: data stack).

```
push %r14
sub 8, %rsp
mov %rbp, %r14

sub 8, %rbp
mov %rbx, (%rbp)
sub 8, %rbp
mov %rcx, (%rbp)

call function

mov (%rbp), %rcx
add 8, %rbp
mov (%rbp), %rbx
add 8, %rbp

add 8, %rsp
pop %r14
ret
```

In consequence to the data separation, the machine instructions of the x86 ISA can now also be classified into different categories. Some of them influence the state of the control stack, others the state of the data stack, and still others do not have an influence on any of the two stacks. Listing 1 illustrates the responsibility of the two stacks for different machine instructions, by the help of an exemplary assembly program. The code highlighted in green influences the control stack, the code in yellow influences the data stack and the code in gray does not have an influence on any of the stacks, mainly because it does not perform a memory access.

2.1. Initialization of Data Structures

The initialization phase is an essential process of the SCADS concept. All necessary data structures, that are needed to guarantee the protection of the control flow data, are created and initialized. The summarized task of the initialization phase is to set up a runtime environment that is conform with the SCADS concept. A program is supposed to be compatible with the SCADS environment as soon as the `main`-function is entered, such that initialization operations must be completed before.

2.1.1. Linking the Initialization Module. As we do intend to separate the initialization phase from the actual program code, we perform the initialization before the body of `main` is executed. One way to achieve this would be to hard-code the initialization mechanism into the compiler, and let it place the machine instructions at the very beginning of `main`. But this is not be the method of preference because anything that is to be executed before the main function should not be the task of the compiler, but the task of the linker. So the initialization phase is encapsulated within a separate module, which is linked to the binary and builds a wrapper around the `main` function.

Listing 2: Command-Line to link a SCADS-binary with the init module.

```
    clang -Xlinker --wrap=main -o scads_bin
        scads_module.o init_module.o
```

An exemplary command-line, which is used to link a binary with the initialization module, is illustrated in listing 2. The here defined object file `init_module.o` contains a function called `__wrap_main`. By passing the `-Xlinker -wrap=main` flags to the linker, it is advised to replace every call to the function `main` with an invocation of the function `__wrap_main`. To ensure the invocation of the original `main` function, the `__wrap_main` function must make a call to it after the initialization is finished. Everything that is contained in, and performed by the initialization module is now be described in section 2.1.2.

2.1.2. The Initialization Process. The initialization itself consists of a couple of steps to be performed. Figure 2 shows the single steps of the initialization phase by the help of a control flow diagram. As mentioned in section 2.1.1, the initialization is implemented within the `__wrap_main` function, which is called by the `_start` function. When the `__wrap_main` function is entered, the runtime environment still consists only of one single stack, the unified stack. This stack is used during the whole initialization to allocate and set up the second stack, the data stack. By the end of the initialization, we no longer reference the unified stack by its original name but call it control stack. Because after the initialization steps, the behavior of the unified stack equals the behavior of the control stack, as explained in section 2.3.

For the whole initialization process, no lib-functions are used but only raw system calls without wrapper functions around them. The reason for this approach is a compatibility issue which occurs when functions of a custom library compiled with SCADS are used before the initialization of the data stack has finished. In such case, the program aborts due to a segmentation fault. This topic is further explained in section 7.4.

Allocation of the Data Stack Before any computations for the allocation of the data stack can be made, the arguments of the main function `argc` and `argv` (passed via `%edi` and `%rsi`) must be saved, because the registers containing them are used by various function and system calls. The two registers are therefore copied onto the unified stack until the data stack is created. Although the `argv` pointer is overwritten later on anyway, it yet stores the information about the location of the

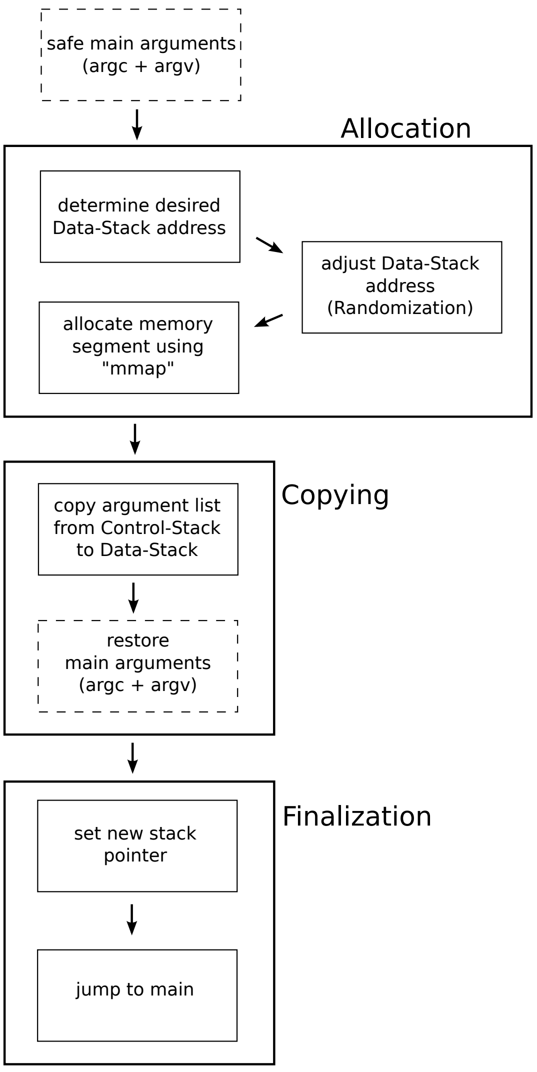

safe main arguments
(argc + argv)
Allocation
determine desired Data-Stack address
adjust Data-Stack address (Randomization)
allocate memory segment using "mmap"
Copying
copy argument list from Control-Stack to Data-Stack
restore main arguments (argc + argv)
Finalization
set new stack pointer
jump to main

Figure 2. Control-Flow diagram of the initialization phase step-by-step.

command-line arguments, which needs to be known for later copying of the arguments, as explained in section 2.1.2.

Afterwards a desired, raw starting address for the data stack segment is computed. This is achieved by retrieval of the unified stack address (i.e. the current value of `%rsp`), which is the control stack address later on. As shown in figure 1, the data stack is determined to lie below the control stack, within the unallocated memory gap between the unified stack and the heap. To prevent the data stack from being placed inside the control stack, we first obtain the maximal extension of the control stack, which can be retrieved via the *getrlimit* syscall by passing the *RMLIMIT* constant as resource parameter. By default the maximal extension size of the stack is defined to be 8 MiB.

$$address_{Data,static} = address_{Ctrl} + 8 * maxExtension_{Ctrl} \tag{1}$$

$$address_{Data,final} = address_{Data,static} + randomEntropy$$
$$with\ \ randomEntropy \in [-2^{23}, 2^{23} - 1] \tag{2}$$

We use the value returned by this syscall to identify the maximal starting address (control and data stack grow both from high to low addresses) for the data stack, satisfying the condition that the data stack does lie below and not inside the extensible area of the control stack. Within a 64-bit address space, there is no lack of leeway, so we can define the raw static starting address of the data stack, as shown by formula 1, without introducing any complications. With a default value of 8 MiB for the $maxExtension_{Control}$ variable, this leads to a starting address of the data stack, 64 MiB below the control stack extension during the execution of the data stack allocation.

Nevertheless, as already indicated, the so computed address is just a static one. Meaning that apart from the $maxExtension_{Control}$ variable, there is no other variable factor to influence the resulting address. For processes with equal maximal stack extension, the gap between data and control stack would remain always the same. If an attacker could obtain this information, pointers could be misused similar to the attack scheme by Bulba [17]. Knowing that the maximal stack extension is a parameter to rarely change, this static sized gap would be the regular case and therefore not sufficient to guarantee an anonymous location of the control stack relative to the data stack.

Listing 3: Allocation of the data stack via mmap.

```
// this variable is placed on the control stack
int control_stack_address = 0;

void *segment_addr_final = (void*)(((long
     long)&control_stack_address) - ((long
     long)DATA_STACK_OFFSET));
const int MMAP_SYSCALL_PROT = PROT_READ | PROT_WRITE;
const int MMAP_SYSCALL_FLAGS = MAP_PRIVATE |
     MAP_ANONYMOUS | MAP_GROWSDOWN;
const int INITIAL_STACK_SIZE = PAGE_SIZE;
const int FD = -1;
const int OFFSET = 0;

// invoke mmap
void *data_stack_ptr =
     invoke_mmap_syscall(segment_addr_raw,
     INITIAL_STACK_SIZE, MMAP_SYSCALL_PROT,
     MMAP_SYSCALL_FLAGS, fd, offset);
```

We want to prevent this to not cause any vulnerabilities for exploits, which may abuse the known relative position of the control stack. Therefore as shown by formula 2, the final starting address of the data stack's memory segment is further enhanced by adding a random component to it. The variable $randomEntropy$ in formula 2 is a pseudo-random value which is computed using *klibc's* [4] `rand` and `srand` functions. We randomize the 24 least-significant bits of the static address by adding the signed pseudo-random value

$randomEntropy$ which lies in the interval $[-2^{23}, 2^{23} - 1]$. This means the data stack can be adjusted in both directions, to high and low addresses, depending on the sign of the pseudo-random value.

With the final data stack address, we can now allocate the memory segment using the system call `mmap`. The entire code that is executed for the allocation is illustrated in Listing 3. As result of the allocation, we receive an anonymous, non-executable memory segment of initial size of one page, including read- and write-privileges. The `MAP_GROWSDOWN` flag, which is passed to mmap, allows the data stack to grow in size, just like the control stack does. The growth of the stack is an essential property, because it allows the allocation of the stack with the minimal size of one memory page, so that no unused memory is wasted. However the growth of the data stack, with the `MAP_GROWSDOWN` flag activated, is a bit more complicated than one would expect. A short explanation of the addressed complexity is given in section 2.2.3. Also note that `MAP_GROWSDOWN` is a Linux specific flag. The equivalent on FreeBSD is the `MAP_STACK` flag, and the call to mmap is accordingly adjusted, depending on the operating system, the binary is compiled and linked on.

Copying of Command-Line Arguments In general, no data migration needs to be performed that transfers data from the control stack on to the data stack, with one exception. We implemented SCADS in such a way that all data gets automatically separated and stored at its destination, from the moment of the invocation of the `main` function. But there is one important block of data, that gets stored on the control stack even before `main` is executed. The talk is of the *command-line arguments*, which shall not remain on the control stack, and therefore do get migrated onto the data stack within the initialization process.

According to the control flow chart of figure 2, the next step after the allocation of the data stack is the copying of the command-line arguments from the control stack onto the bottom of the data stack, we just mentioned. It is vital for the SCADS concept that the arguments are no longer addressed on the control stack. By copying them onto the data stack we once again enforce the important condition of separating control flow data from regular data, as all command-line arguments are classified as regular data and so shall not remain on the control stack.

The goal is to prevent exploits from abusing the `argv` pointer to reach control flow data and modify it. Thus, to successfully prohibit such actions, the `argv` pointer must not contain an address, that points to the control stack. Hence we copy all arguments onto the data stack as mentioned before. To perform this task, the argument list is parsed from the control stack, to identify the overall space that is needed to store the arguments on the data stack. After the whole block of command-line arguments has been copied, the addresses of the `argv` array pointing at the arguments, are no longer consistent. Therefore, the whole `argv` array is translated to the new locations of the arguments, and placed on the data stack, where the `main` function can reach it. Last but not least, the `argc` variable, indicating the number of arguments, and the `argv` pointer, pointing at the start of the arguments array, are restored into the registers `%edi` and `%rdi` to fulfill the parameter interface of the `main` function.

Finalization Two more small steps need to be accomplished to finalize the initialization phase. We already computed the starting address of the data stack and know the offset to the end of the argument list, which is now also located on the data stack. But we must place this address where the program expects it. The SCADS environment is configured to have `%rbp` manage the stack pointer of the data stack (see section 2.2.1). Hence, the starting address of the data stack, below the argument section, is placed into `%rbp` using inline assembler. Finally, a call to the `main` function is made to initiate the program's true control flow, which from then on uses the SCADS runtime environment, including the control and the data stack.

2.2. The Data Stack

As explained in the previous section 2.1, the data stack is a completely new memory segment. This segment was and is not present in a runtime environment with the unified stack. Its allocation is done during the initialization phase (see 2.1.2). Nevertheless, the newly created data stack stores most of the data that is stored on the unified stack, and therefore its size does only differ very slightly from the size of the unified stack. Its location is designed to be below the control stack, within the unallocated memory space between the heap and the control stack. However its position is not completely static, as it is partially randomized to improve the protection against several exploits.

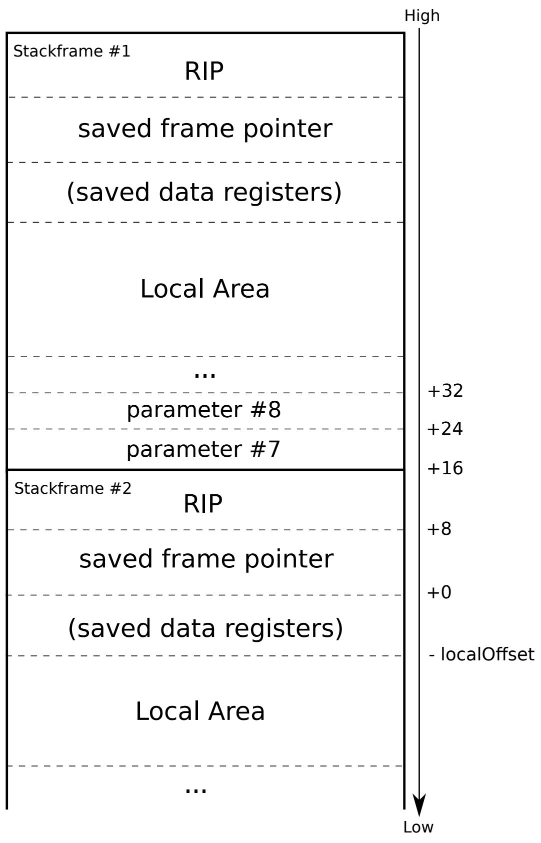
2.2.1. Register Occupation.

The data stack occupies at most two registers for which we have chosen *%rbp* as stack pointer and *%r14* as optional frame pointer. The reason we not chose *%rsp* to be the data stack's stack pointer is based on an architectural matter, which forces the use of *%rsp* for the control stack. A more detailed explanation of this can be found in the forthcoming section 2.3, which focuses on the control stack.

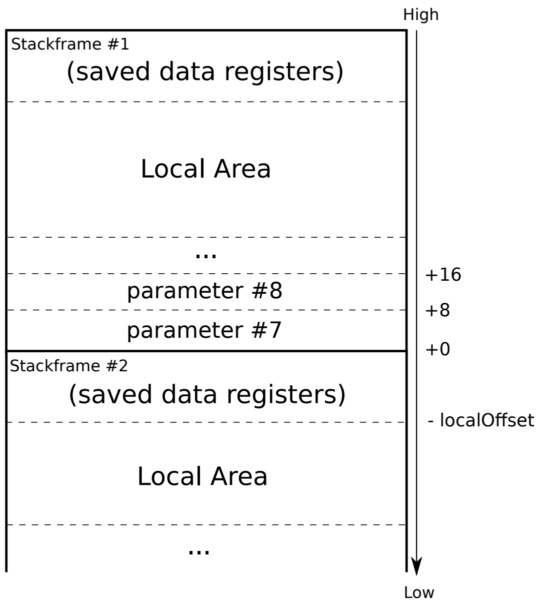
Knowing that *%rsp* is reserved by the control stack, it is obvious to use *%rbp* as stack pointer of the data stack, because it is not occupied by the control stack which does not need a frame pointer for its memory management. The *%rbp* register is most of the time used as a specific purpose register anyway, but we reserve it permanently for the usage as stack pointer. In contrast, the reservation of register *%r14* as frame pointer is not permanent.

2.2.2. Managed Data Types.

On the data stack, we store all sorts of data which are of the regular data type. This includes first and foremost static sized buffers allocated within a program, and in general all sorts of local variables (excluding data that is placed on the heap). There is no machine instruction defined by the x86-64 ISA that implicitly uses *%rbp* without



(a) Several stack frames of the **unified stack** with frame pointer offsets for data structures



(b) Several stack frames of the **data stack** with frame pointer offsets for data structures

Figure 3. Detailed representation of the unified stack in comparison to the data stack (using the AMD64 calling convention).

the register defined as operand. Because of this fact, by using *%rbp* as stack pointer of the data stack we do not end up with problems of storing control flow data like the RIP on the data stack by mistake.

Figure 3(b) shows a compilation of stack frames on the data stack. It is clearly shown that due to the missing frame pointer and RIP on the stack, the parameters stored on the stack are starting at frame pointer offset +0 bytes on a 64-bit architecture. The figure 3 fulfills the concept of the AMD64 calling convention, thus parameter 1 to 6 are not stored on the stack but passed via registers. In contrast to the data stack, the parameters on the unified stack (figure 3(a)) are starting at frame pointer offset +16 bytes. Hence, for the implementation of the parameter access on the data stack, all offsets must be subtracted by 16.

The data stack is also responsible for the storage and retrieval of any callee-saved data registers (not including the frame pointer register, which is classified as control flow data) at the beginning and the end of a function. In this context, a second difference towards the unified model exists, which concerns the topmost access to the data stack. We cannot use *push* and *pop* instructions on the data stack, because these instructions implicitly use the register *%rsp* for storage of the corresponding data, but we define *%rbp* as stack pointer of the data stack. The solution is to simulate the *push* and *pop* instructions on the data stack with a chain of usable, alternative machine instructions, which do not create any side effects on memory regions we want to keep clean from regular data.

Listing 4: Usage of push/pop instructions on the **unified stack**.

```
# push instruction
push %rbx

# pop instruction
pop %rbx
```

Listing 5: Simulation of push/pop instructions on the **data stack**.

```
# push simulation
sub 8, %rbp
mov %rbx, (%rbp)

# pop simulation
mov (%rbp), %rbx
add 8, %rbp
```

Listing 5 demonstrates how the regular *push* and *pop* instructions from listing 4 are translated to operate on the data stack. Every *push* and *pop* is simulated by two instructions, consisting of either *sub*/*mov* or *mov*/*sub*. As we now use two instructions, instead of one, this simulation scheme may have a negative effect on the performance of the program.

2.2.3. Automatic Growth. The data stack is able to grow automatically, just like the control stack, which is managed by the kernel. The flag MAP_GROWSDOWN, which is passed to the mmap system call, enables the feature of the newly allocated memory segment, to grow downwards. But there may be some complications with large stack frames (by large we mean frames with a size larger than one page). It may be necessary to treat such stack frames in a special way to allow the stack to grow correctly. Though, this is a complication dependent on the operating system. On FreeBSD the stack extension can be performed the same way on small as well as large stack frames (as shown by listing 6), without introducing any performance overhead. But on Linux, a large stack frame must be extended page-wise (as shown by listing 7), as otherwise the program's execution is likely to abort.

Listing 6: Stack extension for a large stack frame on the **unified stack**.

```
# stack extension
sub 10000, %rsp
```

Listing 7: Translated stack extension for a large stack frame on the **data stack**.

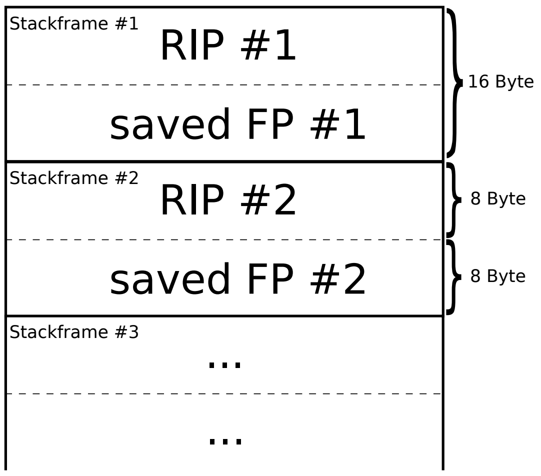
```
# stack extension
sub 4096, %rbp
mov %rax, (%rbp)
sub 4096, %rbp
mov %rax, (%rbp)
sub 1808, %rbp
```

During our analysis on Linux, we have made the observation that it is impossible to extend the size of the data stack in one step for large stack frames, by modifying the stack pointer %rbp only once. Instead it is necessary to extend the stack in steps of single pages (generally 4 KiB large), to ensure that the operating system can allocate guard pages at the top of the growing stack correctly. This circumstance makes need of a chain of instructions to be generated for the extension of the data stack for large stack frames. The chain consists of instructions to modify the stack pointer in direction of growth and *touch* (make a write access) the stack at its current extension. This ensures that the exception handler of the guard page, at the top of the stack, gets invoked, to let the operating system extend the stack by one page. Listing 7 shows the translation of a *normal* stack extension for a large stack frame (shown by listing 6) to the extension mechanism needed for the data stack.

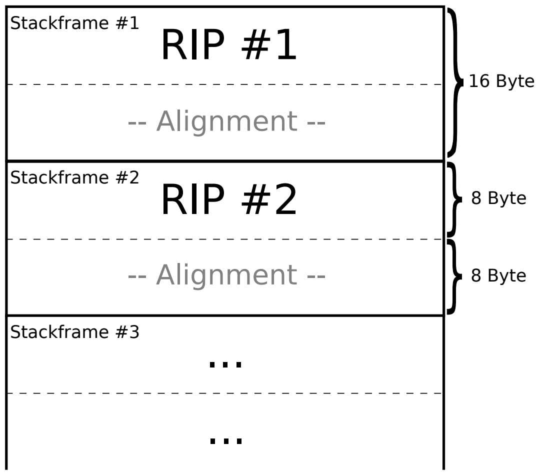
If the data stack's extension is not specially treated, as shown in listing 7, and instead extended normally, as illustrated in listing 6, the program aborts with a segmentation fault due to a memory access to not allocated memory. By extending the stack pointer by more than about one page's size (depending on the position of the stack pointer in the current stack frame), the guard page at the top of the stack is skipped and the next memory access takes place at the unallocated memory region between the data stack and the heap.

2.3. The Control Stack

The control stack stays at the place of the unified stack. This means that it is allocated by the kernel of the operating system at load time of each program. Due to the fact that it now stores much less data compared to the unified stack, the control stack is of rather small size and generally not exceeds the size of one page with the exception of heavy recursion. It keeps the direction of growth from high addresses to low ones and expands towards the data stack.



(a) Several stack frames of the control stack with frame pointer in use



(b) Several stack frames of the control stack without usage of frame pointer (including optional 16 byte alignment)

Figure 4. Detailed representation of the control stack.

2.3.1. Register Occupation.

By the nature of the x86-64 architecture, the stack pointer of the control stack must be stored in the register *%rsp*. This is justified by the fact that the *%rsp* register is implicitly used by the processor for storing the RIP in memory. So every RIP that results from a function call is stored at the memory location *%rsp* points to. Using a different register as the stack pointer would lead to the consequence that generated return addresses would not be stored on the control stack in the first place, thus becoming necessary to copy or move them to the control stack, right after a function has been called. Not only would this lead to a significant performance loss, because a couple of instructions would be needed for copying the addresses. Moreover it would construct a severe vulnerability that would eliminate the necessary condition of separation for a secure environment, defined by the concept of SCADS. The only other reasonable choice for the use of *%rsp* would be as stack pointer for the data stack. Therefore we would store the most vulnerable control flow data, namely the RIP, on the data stack. Hence, it is actually mandatory to use *%rsp* as stack pointer for the control stack.

2.3.2. Managed Data Types.

The control stack is responsible for the maintenance of control flow data. To be exact, the control stack stores *return instruction pointers* and *saved frame pointers* only. This means, that it handles existing control flow data except of function pointers. Our SCADS implementation is yet only prototypical and the usage of the control stack is intended to be extended in the future. But the handling of function pointers is a rather complex task, as the C specification on the usage of pointers of any type, as function pointers, is far from strict.

“A pointer to an object or to void may be cast to a pointer to a function, allowing data to be invoked as a function (6.5.4). A pointer to a function may be cast to a pointer to an object or to void, allowing a function to be inspected or modified (for example, by a debugger) (6.5.4).”[5, J.5.7 Function pointer casts]

The loose specification of function pointers, produces the need to treat any arbitrary type of pointers as function pointers, because such transferring casts are not prohibited. Nevertheless, the storage of regular pointers (classified as regular data) on the control stack in general enforces the mixing of regular data and control flow data on the control stack, which threatens the security of the primary control flow data such as the return addresses. Accordingly, as long as function pointers remain ambiguous, as defined by the C specification, there is no proper solution to find the right place for their storage. So far, function pointers could be cast into pointers of any other type and vice versa, without offending the terms of their specification. This is the main reason why we do not configure the control stack to handle function pointers in SCADS.

Because *%rsp* is used to store the stack pointer of the control stack, it is further possible to use *push* and *pop* instructions for storage and retrieval of data on the control stack. This makes the implementation of the control stack extremely simple, as we do not need to change any machine code generation in the compiler, which concerns the storage and removal of RIPs and the frame pointer of the data stack. They are implicitly stored and retrieved from the control stack by usage of the *call*, *ret*, *push* and *pop* instructions.

2.3.3. Alignment.

Alignment is an optional feature in SCADS which enables compatibility to legacy code, e.g. legacy libraries; it can be enabled by a flag at compile time. When code compiled with SCADS is not confronted with legacy code, this feature is not necessary for a program. The purpose of stack alignment has various reasons. In case of a 64-bit architecture like x86-64, every memory address is represented by an 8 byte value. Since we only store at most two addresses per function call on the control stack, it is always implicitly 8 byte aligned. The planned usage of the control stack would not require any coarser alignment. But to be compatible to legacy code, which uses the control stack for storage of control flow data, as well as regular data, an 8 byte alignment would not be sufficient in some cases. To fulfill a maximum of 16 byte alignment, in case no frame pointer is in use, we extend the control stack by another 8 byte using the *sub* and *add* instruction. On the other hand, if a frame pointer is enabled for a stack frame, the stack is already 16 byte aligned and thus no further operations are performed. Figure 4 shows the two cases which do either not need a forced stack alignment, because the frame pointer is in use, or a stack alignment needs to be enforced by adding gaps of 8 byte of memory to each stack frame.

2.4. Implementation in the LLVM Back-End

We briefly want to describe, what was done to implement SCADS in the LLVM back-end in practice. As our implementation’s target architecture is the x86-64 architecture, most of the modifications have been made to the files belonging to the x86 back-end. A complete list of files (with full project path), which have been modified is:

- include/llvm/CodeGen/CommandFlags.h
- include/llvm/Target/TargetOptions.h
- include/llvm/Target/TargetFrameLowering.h
- lib/CodeGen/PrologEpilogInserter.cpp
- lib/CodeGen/PrologEpilogInserter.h
- lib/CodeGen/TargetFrameLoweringImpl.cpp
- lib/Target/X86/X86RegisterInfo.h
- lib/Target/X86/X86FrameLowering.h
- lib/Target/X86/X86RegisterInfo.cpp
- lib/Target/X86/X86FastISel.cpp
- lib/Target/X86/X86ISelLowering.cpp
- lib/Target/X86/X86FrameLowering.cpp
- lib/Target/X86/X86ISelLowering.h
- tools/llc/llc.cpp

Note that the code for the initialization module is completely independent from the Clang/LLVM compiler and has therefore been excluded into a separate file we named *main_wrap.c*. The name arises from the fact that the initialization module is linked to the program in such a way, that it wraps around the *main* function and calls it as soon as the initialization is done. While the files *llc.cpp, CommandFlags.h* and *TargetOptions.h* have only been adjusted to implement the various flags we integrated to let the user decide about the features to use, the rest of the listed files contribute to the actual functionality of SCADS. The set of flags now contains the following four:

- **`-num-stacks [number of stacks]`**
- **`-enable-xor-encryption`**
- **`-enable-legacy-callback-compat`**
- **`-enable-legacy-stack-alignment`**

As most of the features of SCADS do not interfere with any function's body, but just need to make changes to the prolog and epilog of functions, the file that was most important for SCADS' implementation is the file *X86FrameLowering.cpp*. In this file, the whole prolog and epilog mechanism for every source code function is implemented within the two functions *emitPrologue(. . .)* and *emitEpilogue(. . .)*. It is implemented how the stack extension is managed (regarding normal and large stack frames), which way registers are to be pushed and retrieved from the data stack (as `push/pop` are not usable on the data stack), and the functionality of the compatibility flags, as well as the XOR encryption (see section 2.5).

The LLVM back-end is structured in such a way, that every hardware architecture has its own back-end, separated from any other back-end, to implement the generation of code for just one specific architecture. This also applies for the x86-architecture. For that reason, one of our goals during the implementation of SCADS was to interfere as few as possible with other hardware architectures. Largely, we have accomplished this task, but there is one upper class every single back-end inherits from, which we had to modify, hence also taking influence on the implementation of other back-ends. In the class *TargetFrameLowering* of the file *TargetFrameLowering.h*, we included a couple of data structures and added a second constructor to handle the management of more than one stack. In fact, the class' structure has been modified in such a way that it would be easy to let its inheriting classes handle an arbitrary number of stacks in future. The newly added constructor is only used by the x86 back-end. Some data structures like the *StackDirection*, that were originally single values, have been transformed into arrays to manage multiple values of the same type (i.e. multiple stacks instead of just one). Nevertheless, the access on these arrays is fully managed through read-only functions of that class, which we adjusted in such a way that they hide the information from the caller of the read data structure being an array. So the behavior of the non-modified back-ends remains the same.

Although we had to make a few major changes to the architectural design of the compiler's class structure, we have implemented SCADS in such a way that the compiler is fully downward compatible. If the `-num-stacks` flag is set to "1", the compiler generates code the way it did before we modified it. The rest of the yet not mentioned files, listed above, have experienced minor modifications to maintain the new register occupation for the data and the control stack, and to adjust the offsets to the data structures on the data stack.

2.5. Extension: XOR-Encryption

The SCADS approach alone is designed to protect control flow data, especially the return address, from abuse by malicious code injection. Nevertheless, in this section we want to introduce an extension to the SCADS approach, which also targets the protection of the RIP. This approach is similar to the one presented in [14, section 3.1(c)]: By encrypting the RIP with XOR, we can enforce an enhanced security level regarding the integrity of any stored RIP on the control stack.

Every time a function is entered, at the very beginning of the function, the RIP is encrypted. The decryption of the previously encrypted function takes place at the very end of each function. Thus the return address remains encrypted for the whole runtime of the function (including its sub-calls), until the decrypted state of the return address is needed again, to jump back to its caller. So the implementation of the scheme only requires us to extend the prolog and epilog of every function, generated by the compiler.

To preserve the performance as much as possible, the encryption algorithm is reduced to a simple XOR with the current value of the stack pointer. As performing the XOR operation twice on a single value obviously results in its identity, it can be used for encryption as well as for decryption of the return address equally. Listing 8 shows the implementation of the XOR-encryption within a function's prolog and epilog.

Listing 8: Implementation of the XOR en-/decryption.

```
# prolog: encryption of RIP
xorq %rsp, (%rsp)
...
# epilog: decryption of RIP
xorq %rsp, (%rsp)
```

Here we use the stack pointer for the encryption of the return address. Nevertheless, it would also be possible to use the current frame pointer (if present) to fulfill this task, as shown in [14, section 3.1(c)]. Yet, because the RIP is stored on the control stack, which does not own a frame pointer, the only possibility would be the use of the data stack's frame pointer. But the location of the data stack can be easier obtained by an attacker than the one of the control stack. So an attacker might have a better chance to create a malicious, encrypted return address, whose decrypted result could be a valid address, if the data stack's frame pointer would be used for encryption. Therefore it is wise to choose the `%rsp` register (i.e. the stack pointer of the control stack) for the encryption process.

We want to mention that we define the approach of XOR-encryption of the RIP within our SCADS environment as more effective, because of the fully dynamic randomized positioning of the data stack towards the control stack. In contrast the randomization (or "float" as they call it) of Fu's and Wang's approach [14, section 3.1(b)], takes place during compile time, which lowers the effect of the XOR-Encryption, due to the fact, that the stack address may be known more likely.

3. Related Work

In history, many different approaches have been invented to pit themselves against the menace of exploits of all kind, from which some are similar to our approach. We want to have a discussion on these related schemes and point out the differences between them and SCADS.

3.1. StackGuard

StackGuard [10] is a compiler-based security measure that focuses on the protection of return addresses and saved frame pointers, just like SCADS does. To achieve this task, binaries must be recompiled into code which places so called *canaries* (or *canary word[s]* [10, chap. 3]) on the stack, right below a return address or saved frame pointer. These canaries are in general 8 byte sized values, which can be of different, common types: *random canary*, *terminal canary*, and *XOR canary* [22, sec. 2.2]. Random canaries consist of randomly generated values, terminal canaries include characters that cannot be used within an attack vector of an exploit, and XOR canaries are random canaries which are additionally encrypted with the return address of the corresponding stack frame. While these three types do possess their own specific properties, the security of the whole concept strongly depends on the anonymity of any created canaries. On every removal of a stack frame, with a subsequent jump to the previous control flow, the previously stored canary is checked for integrity and this way allows to detect buffer overflows that may have modified the protected data. When a buffer overflow occurs, the attack vector is placed in the buffer and further, potentially overwriting the return address. But to reach the return, the attacker must overwrite the canary linearly. Thus it is vital for the delivered security level that an attacker must not be able to reproduce the correct value of the canary.

There exists an extension of StackGuard, called *MemGuard* [10, sec. 3.2], which offers a higher security level than StackGuard itself. But with the higher security level, this concept "suffers substantial performance penalties compared to"[10, sec. 4.2.2] StackGuard. For that reason, we are not discussing MemGuard in this article, because performance preservation is one of the main goals of SCADS. A detailed analysis of SCADS' and StackGuard's security quality in comparison, is held in section 4.

With StackGuard, a slight performance overhead may be attached, because the placement of the canaries on the stack requires the execution of "7 [more machine] instructions"[10, sec. 4.2.1] per function call, than without the usage of StackGuard. This performance overhead is generally very small, but may sum up to a noticeable amount when the number of performed function calls is rather high. In contrast, SCADS does not introduce additional machine instructions for each stack frame, hence there is no significant performance overhead in general. The memory overhead of StackGuard is negligible, as the only additional memory that is used, arises from the canaries placed on the stack. With canaries of 8 byte size on a 64-bit system, the average call-depth of the program must be high to create a significant memory overhead. SCADS produces a small initial memory overhead at program start, that is likely to be compensated at runtime.

3.2. StackShield

StackShield [2] is a defensive concept similar to SCADS in specific ways. In 2008, the idea of StackShield was revisited for the binary rewriting tool *TRUSS (Transparent Runtime Shadow Stack)* [27], with the difference that a return address is compared to its shadow copy rather than enforcing its integrity by restoring a backup value.

By managing a redundant storage area, that contains return addresses of the regular control flow, StackShield reminds of the control stack of SCADS which being separated from the main stack. The storage area of StackShield though, is located in the data segment of a process' memory space. Within this memory space, StackShield manages redundant copies of any return address, created during runtime. The original return addresses from the stack, however, remain at their original memory location and are not modified. The objective of StackShield is to prevent the permanent modification of a return address by comparing its value with its copy in the redundant storage area, before the `ret` machine instruction is executed. In case a buffer overflow occurred and modified the return address maliciously, StackShield does notice the modification and perform a counter action. It is up to the programmer to define the behavior in such a case. StackShield may either abort, due to a detected exploitation attempt, or overwrite the modified return address with its original value from its copy and keep the program running, even though some local data may have been altered and causes undefined behavior. Just like StackGuard and SCADS, StackShield is a security mechanism that is compiler-based. This means that it the activation of StackShield within a binary does require to recompile the program, but not to modify the source code.

StackShield does not protect data like saved frame pointers but only the return address. This introduces some vulnerabilities to exploits not focussing on the modification of return addresses solely. In a later version, support for the protection of function pointers with an address range check has been introduced, although this mechanism brings in a couple of additional machine instructions to be executed and therefore decreases the performance noticeably. The overall performance overhead of a binary with StackShield activated may not be underestimated, because the overall management of the redundant storage area, incorporated into every prolog and epilog of a compiled function, is again quite expensive. Like StackGuard, StackShield's performance overhead is thus bound to the number of calls that are made within a program, while SCADS' performance overhead is only *static*.

The memory overhead of StackShield is also similar to the one of StackGuard. Every new stack frame requires a few additional bytes to store the redundant return address as well as minor collateral data used to maintain the data structures of the redundant area in the data segment. Unless a program does not have a very high average call-depth, the memory overhead is not large at all.

3.3. Multi-Stack Approaches

During our research, we found two existing concepts which are especially similar to our approach as they make use of multiple stacks, just like SCADS does. These two concepts are *MSC* ("Multiple Stacks Countermeasure") [29, p. 63ff.] and *SCISM* ("Stack Control Information Separating Mechanism") [14]. While these approaches are closely related to ours, there exist some essential differences having an influence on the security and other properties. We explain these differences and point out what sets SCADS apart from MSC and SCISM in the following.

3.3.1. MSC. MSC is a security measure based on the idea of separating data on the stack to different memory locations. However, MSC follows a much more excessive strategy of data separation by using not only two different stacks but up to five stacks. The five stacks and data they manage are classified upon risk levels that indicate how likely data is to be used for exploitation, as indicated by [29, p. 64, table 3.1]. Thus MSC separates all data, stored on the stack, in a fine-grained way, to prevent correlation between data types, that shall be strictly isolated from each other. As all other previously presented concepts, MSC is a compiler-based approach as well. For the maintenance of the five newly introduced stacks, the compiler that implements MSC uses the frame pointer of a stack frame and modifies the offsets to any local data to redirect the access onto the stack of desire. So in contrast to SCADS, MSC does not reserve a register (or pair of registers) for every single stack that is added to the original one. This method forces the location of every single stack to be static, thus defined at compile time. In our view the static location of every stack and also the static offset between all stacks can introduce a security vulnerability for exploits making use of random write access. Such exploits can easier find the data on corresponding stacks that they need to modify, because the relative position is known before execution.

MSC tries to address this problem by placing all character buffers on a stack where nothing but buffers are stored. However, as explained in [29, p. 63, sec. 3.3.1], it is also possible to store character arrays within a struct, and the data of structs does not get separated across stacks (see [29, p. 68, fig. 3.2]). This proves the approach is kind of "over-engineered" because it is possible to gain random write access during an attack, when a struct containing a buffer inhabits an arbitrary pointer. An example for such an exploit is shown in section 4. We classify the security level of SCADS in such a scenario as more safe, because it is capable of protecting against these exploits.

According to the performance analysis that has been performed in [29, p. 70, sec. 3.4.1], MSC does introduce only minor performance impacts. SCADS is also designed to cause as few performance overheads as possible. Nevertheless, the fact that MSC does not occupy an additional register for the maintenance of the stacks may result in more optimal code. The memory overhead that comes with MSC is rather high: "The drawback of this countermeasure is that it results in gaps on the remaining stacks, resulting in wasted memory" [29, p. 69]. All five stacks are allocated with the same size, which is the maximum size of the unified stack, that needs to be known at compile time. For the storage of the specific data types, that are placed on each of the five stacks, no reordering is fulfilled, to prevent an internal fragmentation as shown by [29, p. 69, fig. 3.3]. This leads to the problem that MSC causes a memory overhead of $5 * sizeof\, stack$ when all five memory segments are mapped to physical memory, which happens as soon as all of the stacks have been used once with an arbitrary operation. So in contrast to a binary that is not been protected with MSC, the memory overhead is increased fivefold and the overall overhead is dependent on the size of the stack, leading to an even worse decline in memory usage, the higher the original stack memory consumption of a program is.

With SCADS we do have an initial memory overhead that cannot be weed out, but in comparison to MSC, SCADS tends to compensate the initial memory overhead the more memory is used by an application at runtime, instead of even increasing the overhead. Furthermore, the initial memory overhead with SCADS is of static size and is explicitly one page (needed to be allocated for the initialization of the data stack) while the memory overhead of MSC is dynamically increasing with the maximum size of the stack. Regarding memory analysis, Younan states [29, p. 72, sec. 3.4.2] that it may be possible to reduce the memory consumption for future work, but it is not an available feature yet.

With SCADS we have the full ability to eliminate the frame pointer in stack frames, which do not need a frame pointer for their data management, whereas MSC is designed to be dependent on the use of a frame pointer in every stack frame. "[The] frame pointer is used as a fixed location access [...] because the value of the register containing the stack pointer changes whenever a variable is pushed or popped from the stack" [29, p. 68, sec. 3.3.2]. Hereby MSC unfortunately loses a simple performance optimization technique.

3.3.2. SCISM. SCISM [14] again is a similar concept to SCADS, though it differs in some essential details. SCISM manages two stacks for data separation, very similar to

SCADS. It allocates the second stack, that is responsible for the storage of the control flow data, within the data segment. One difference to SCADS is the size of the stack, managing the control flow information. In SCISM the stack is allocated with fixed size, defined at compile time. In case the size of the stack is not sufficient, the additionally created control-flow data is migrated to an isolated area within the heap. This forces a permanent, performance-decreasing check for data to be stored on the "local stack" [14, sec. 3.2], whether it already exceeds the maximum size. SCADS allows an automatic growth of the control stack, therefore making any boundary checks unnecessary and allowing a more dynamic way of using the stack. SCISM's behavior in this case is not only decreasing the performance, especially when the size if the stack gets exceeded, but it also causes an initial memory overhead due to the local stack initialization with a static size.

Another performance-limiting difference between SCADS and SCISM is the task specification for the two stacks. SCISM uses the original unified stack for the storage of regular data, while the stack in the data segment is responsible for the storage of control-flow information like return addresses and saved frame pointers. SCADS twists the management vice versa, by using the unified stack (managed by `%rsp`) as control stack, and the data stack is the memory segment to be newly allocated. This may sound like an unimportant difference, but in fact it has an influence on the performance of the security measure. The `call` and `ret` instructions of the x86-architecture are defined to use the the register `%rsp` for their memory access (either storage or retrieval of return addresses) and there is no option to alter this behavior. The consequence for SCISM is, it has to perform copy operations to move the return address between the stacks, because it is not implicitly placed on the right stack. This results in a performance overhead for SCISM which is not present in SCADS, because the control stack is managed via `%rsp`, causing all return addresses to be immediately stored in the right place, thus making any copy operations unnecessary. However, in consequence to the different usage of the two stacks, SCISM does not incorporate incompatibilities to legacy code, as the calling convention is not altered and parameters are accessed via `%rsp`.

Just like MSC, SCISM does not make use of reserved registers for the maintenance of its stacks. This again causes the problem, that the location of the *local stack* is not randomized at runtime. "The offset of the first address for SFM data area from the begin of the static data block is float and decided randomly within a certain constant c [. . .] The offset is fixed in once compile[d] and varied in different compilers." [14, sec. 3.1] Furthermore the fact that the data segment's start address is also not randomized by ASLR on Linux, makes it especially easy for an attacker to reach the local stack and its managed data when random access write is gained. Once more we want to emphasize that we consider a known location of the second stack as a severe vulnerability towards various exploits (as shown in section 4)

3.4. Hardware-Assisted Approaches

In 2009, Francillon et al. [13] presented a multi-stack approach similar to SCADS in the sense that data and control information are stored separately on two distinct stacks. In their work, however, the authors propose a hardware-assisted modification for embedded systems like the AVR micro controller and prove the effectiveness of their approach based on FPGA simulations. A hardware-assisted solution has the advantage that insructions like `call` and `ret` can be modified in a way that return addresses are placed directly on a secure stack without being accessible for user instructions like `mov`. On the downside, hardware-assisted solutions cannot be deployed to end-users easily and cannot be elevated to the x86 architecture easily.

Only established companies like Intel and AMD are in the position to deploy security-related hardware changes with a practical impact for x86. In a U.S. patent from July 2014 [11], AMD proposes the encryption of return instruction pointers. In this approach, data and control information are not separated on two distinct stacks – and hence, control information can still be overwritten. But as instruction pointers are encrypted, they cannot be modified in a predictable manner. This solution is not available on the market during the time of this writing, but the patent states a minimized performance overhead similar to SCADS because the encryption takes place in hardware. As an advantage over SCADS, recompilation of existing code will not be required.

With the *Memory Protection Extensions (MPX)* [7, 21], first proposed in June 2013 as an extension for Intel x86 architectures, Intel presented a concept built upon the idea of integrating boundary checks for buffers. This way, buffer overflows shall be prevented and with it any exploits that make use of buffer overflows. The mechanism of MPX addresses exploitation prevention at an even earlier stage than SCADS. A compiler, which implements the mechanism of MPX is used to integrate the boundary checking into any program code to be compiled. Boundary checking, however, is not present in the specification of the programming language C.

For the boundary checks, "Intel MPX introduces new *bounds registers* and new instructions that operate on bounds registers" [7, sec. 9.3]. These bounds registers are split up into a high and low part of 64 bit size each. When configured for a buffer, such a bounds register represents the lower as well as the upper bound of the corresponding buffer, according to its length. Before an element within the buffer is accessed, a check using its assigned bounds register is made to verify that the access does not exceed the limits of the buffer: "An out-of-bounds memory reference then causes a BR exception." [7, sec. 9.3]. As long as boundary checking can be performed for any single buffer, MPX should be capable of preventing all buffer overflows. However, MPX does not have unlimited bounds registers, but only four of them. A topic of interest might be what happens when more than four buffers are active within a single stack frame. In case MPX saves any of the bound registers to the stack to switch between the buffers to be operated, this introduces another vulnerability as soons as an attacker finds a way to modify the value of the bounds register on the stack. Unfortunately, we were not able to solve these questions at the time of this writing, because MPX was not yet available yet.

4. Security Evaluation

We now inspect the security that can be achieved by using SCADS as defensive approach against known stack smashing techniques. The task to visualize and explain the security level of a defensive method can be highly complex. The achievement of a formal verification is impossible when it comes to a randomized memory layout, which is an essential requirement of SCADS to guarantee the protection against advanced types of exploits. Therefore we do not seek to verify the security of SCADS formally, but instead focus on giving as many relevant examples of common stack smashing exploits as possible. We apply these exploits to programs compiled with SCADS to analyze their effectiveness.

At the same time, we want to discuss the effectiveness of SCADS in contrast to other security measures to point out the uniqueness of our defensive technique. To perform this task, we stick with the most commonly used and widespread alternative approaches known, which we decided to be *StackGuard* [10] and *StackShield* [2]. Both of these methods share the high priority target with SCADS to protect the return instruction pointer from illegal misuse and modifications. Thus they are suitable for an appropriate comparison with SCADS, regarding the level of protection that can be achieved in each case.

In case the return address can be modified permanently without notification, any of the three concepts can be seen as vulnerable against an attack. Thus we simplify our security analysis to exemplary exploits which try to modify the return address permanently. Any tasks beyond the modification of the return address, which are normally performed by real world exploits to invoke the execution of malicious code are ignored in this analysis. Any attack vector we construct for an exploit does therefore only include the data to overflow a supplied buffer and to reach a return address by whatever means. For the details that follow on the successful modification of return instruction pointers, the reader may have a look at the known literature [17, 19, 20], where several stack smashing techniques are explained and referenced entirely.

As shown in listing 9, each of our test binaries is compiled to incorporate the StackGuard, StackShield or SCADS defensive mechanism. The compilation of the SCADS binary is slightly more complex, because the flag which enables SCADS (`-num-stacks`) has been implemented in the compiler's back-end (LLVM). Hence, the LLVM build process needs to be split up into three parts (*intermediate code generation*, *machine code generation* and *linking*), as long as SCADS is not enabled by default. We used *StackShield v0.7* and *GCC v4.6.3* for generating the binaries with StackShield and StackGuard activated.

Listing 9: Build commands for binaries compiled with SCADS, StackGuard and StackShield.

```
# StackGuard build process
gcc -O0 -fstack-protector -std=c99 -o binary_name
     source_name

# StackShield build process
shieldgcc -O0 -std=c99 -o binary_name source_name

# SCADS build process
clang -O0 -emit-llvm -S -o intermediate_name source_name
llc -O0 -march=x86-64 -num-stacks=2 -o asm_name
     intermediate_name
clang -Xlinker --wrap=main -o binary_name asm_name
     main_wrap.o
```

4.1. Simple Buffer Overflow

As first example we want to present a simple exploit abusing buffer overflows. It is only successful on older systems and binaries without further security measures in use, like ASLR and DEP. Although the presented exploit may be rather harmless today, we want to prove that SCADS alone protects against such an attack, and compare the effectiveness with StackGuard and StackShield. For this purpose we use an abbreviation of an example presented in Aleph One's summary of various stack smashing techniques from 1996 [20].

The source code of the frame work, that is used to establish a vulnerable environment for the exploit can be seen in listing 11. Within this example, the vulnerability is caused by the function `strcpy`. The function does not check the size of the output buffer according to the size of the input data. Thus, it is easy to cause a buffer overflow when a string as parameter is used whose size is larger than the output buffer.

Listing 10: Attack vector to exploit framework I.

```
char attack_vector[120];

// fill attack_vector with 120 bytes of
    value 0xff
void set_attack_vector(){
  for(int i = 0; i < 120; i++){
    attack_vector[i] = '\xff';
  }
}
```

Our three test candidates differ in their implementation details, so we cannot use the same attack vector for all three concepts, with the same expectations in general. To justify the effectiveness of an exploit, the attack vectors must be individually defined for every single of the three concepts we investigate. Nevertheless, this example is basic enough that it allows the same attack vector to be used for SCADS, StackGuard and StackShield equally. The exploit we want to introduce with this example, constructs the attack vector in such a manner that it writes data to the buffer, to try to linearly

Listing 11: Exploit framework I (inspired by [20]).

```
#include <string.h>

void overflow_function(char *str) {
  char overflow_buffer[8];

  // buffer overflow occurs here
  strcpy(overflow_buffer,str);
}

int main(int argc, char **argv) {
  set_attack_vector();
  overflow_function(attack_vector);

  return 0;
}
```

overwrite the return address which is expected to be above the overflowed buffer.

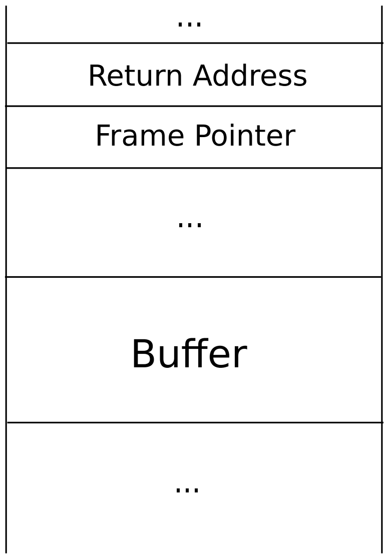
The resulting attack vector, to perform the stack smashing attack, is described by listing 10. We fill the attack vector simply with 120 bytes of the value `0xff`. The size of the buffer is not necessary to be 120 bytes large, but we just chose a size that is large enough to generate an appropriate reaction on all three compiled binaries.

The attack strategy for this exploit is rather simple and is outlined by figure 5. It uses the `strcpy` function to cause a buffer overflow on the entitled `overflow_buffer` (see listing 11). To reach the control flow information (i.e. the return instruction pointer) on the stack, the exploit just fills the buffer with the attack vector declared in listing 10 to linearly overwrite all the space between the start of the buffer and the return instruction pointer, until it eventually reaches the return address itself.

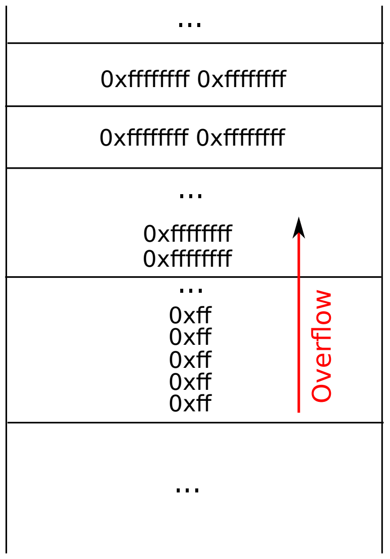
4.1.1. StackGuard.

We first take a look at the behavior of StackGuard towards the exploit. We already know from section 3 how the memory layout is theoretically organized for binaries compiled with StackGuard activated. In fact the memory layout does not differ much from the one of an arbitrary ELF binary at runtime, without any further security mechanisms. The essential difference is the canary that is placed right under the saved frame pointer (or right below the return address if no frame pointer is in use). This canary is also the hurdle the exploit needs to surpass to be effective and perform the action it is intended for.

In figure 6 we see what happens when the exploit overwrites the memory of the process invoked with the StackGuard binary. As the memory layout of StackGuard does not differ immensely from any general layout, figure 6 does only differ slightly from the fictive memory layout shown on figure 5. The only difference is the section that is highlighted in red, which is obviously the stack slot that is occupied by the canary, generated by the StackGuard compiler. By looking at figure 6(b) to the right, we can see the change



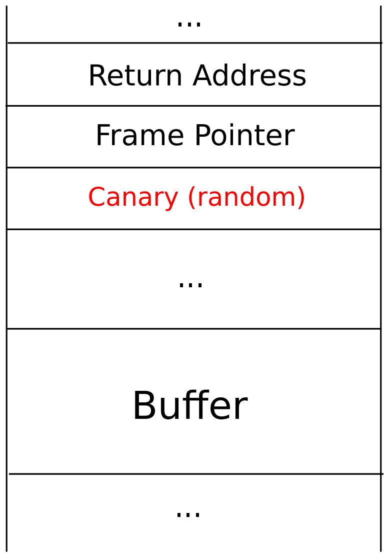
(a) Memory layout **before** overflow



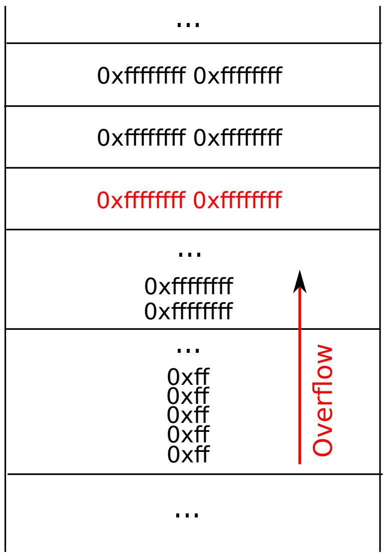
(b) Memory layout **after** overflow

Figure 5. Memory layout of the exploit framework described by listing 11 (without the usage of any security mechanism) before and after initiation of a simple exploit.

that has been made, after the buffer overflow occurred. The presented exploit does not perform any actions to bypass the stored canary, but just overwrites it completely with a value, different from the original. As reaction, the StackGuard mechanism detects the buffer overflow before returning from the `overflow_function` (ref. listing 11), because a check is made to verify the integrity of the canary value. Finally this leads to a signification to the user, about the buffer overflow that took place, and to the abortion of the program in the end. Undoubtedly, the StackGuard mechanism has the ability to prevent a successful exploitation by a stack smashing technique of the presented type.

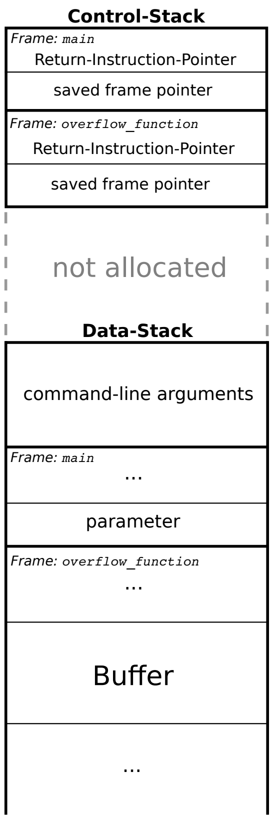


(a) Memory layout of a StackGuard binary **before** the buffer overflow

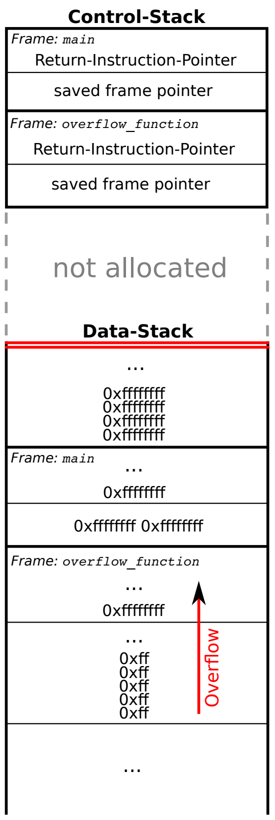


(b) Memory layout of a StackGuard binary **after** the buffer overflow

Figure 6. Memory layout of the exploit framework described by listing 11 compiled with StackGuard.



(a) Memory layout of a SCADS binary **before** the buffer overflow



(b) Memory layout of a SCADS binary **after** the buffer overflow

Figure 7. Memory layout of the exploit framework described by listing 11 compiled with SCADS.

4.1.2. SCADS.

Next up, we find out whether the SCADS approach is also able to deal with the exploit, to not allow further malicious actions to be performed. In contrast to StackGuard, with SCADS we do have greater change in a process' memory layout. We want to remind the reader that SCADS uses two stacks instead of one.

Figure 7 represents the memory layout of a process executing the SCADS binary. Neither return addresses nor saved frame pointers are stored on the data stack where the `overflow_buffer` from our example resides. The control flow data is here located on the control stack, which is separated from the data stack by a large block of unallocated memory. To reach the control flow data, subsequent to a previously enforced buffer overflow on the data stack, the attacker's exploit needs to somehow bypass the unallocated memory space and continue overwriting the necessary data on the control stack.

Though, the exploit we presented is yet only simple and does not perform any complicated actions to skip the unallocated memory block. As explained earlier, it just overwrites the memory above the buffer linearly. Hereby it reaches the border of the data stack (highlighted in red within figure 7(b)) and tries to write to non-allocated memory. This causes the process to terminate with a segmentation fault, leaving the control flow data on the control stack completely untouched. Consequently, the exploit is not able to harm the control flow data within the SCADS environment, leaving SCADS immune to such an exploit.

4.1.3. StackShield.

Last but not least, we take a look at the result of the StackShield's trial. It is not necessary to visualize the memory layout of a StackShield process for this example individually. In fact, the layout is nearly identical to the one presented by figure 5. The only difference is the redundant storage area located within the data segment, below the stack. This area inhabits the copies of all redundantly duplicated return addresses. But the fact that the data segment lies below the stack, while the buffer overflow occurs towards the top of the stack makes it impossible in this case to reach any return address within the data segment, and to modify it permanently. Instead the exploit just overwrites the temporary return address on the stack, which is afterwards detected by StackShield, due to a comparison of the temporary return instruction pointer on the stack, with the redundant copy from the data segment. Further on, this leads to the same reaction as we have seen from StackGuard. The buffer overflow is indicated to the user and the StackShield mechanism causes the process to abort its execution (if not configured differently).

Therefore, also StackShield is capable to defend against the presented exploit successfully. Summarized, we can confirm, that all of the three defense concepts were either able to detect and prevent the exploit from taking control over the program's control flow, or were completely immune to the attack scheme by leaving no chance to reach the control flow information for a malicious modification. So SCADS is fully capable to prevent a successful exploitation when it comes to a simple stack smashing technique. While this alone may not be fascinating, the effectiveness of SCADS can only be stated step by step and we now continue to investigate more complex attack schemes.

4.2. Frame Pointer Overwrite

In [22], Richarte Gerardo lists a few examples how to exploit StackGuard and StackShield successfully. One of the examples describes the modification of the saved frame pointer to alter the control flow consequently. Though this scheme is not usable anymore to attack StackGuard, because since the paper [22] has been released, modifications have been made to the implementation of StackGuard. Most notably the saved frame pointer is now also protected by canaries placed on the stack.

The idea of altering the frame pointer to be indirectly able to alter a process' control flow originates from the way, how a stack frame is cleared. Listing 12 shows the essential machine instruction (highlighted in red) that is needed to make such an attack possible. This instruction moves the value of the frame pointer register into the stack pointer register. By overwriting the saved frame pointer of a stack frame, one can get control over the stack pointer in the next upper frame and abuse this circumstance by setting it to a location within the memory where a self-prepared return address lies. On return, the instruction pointer is then set to this address, thus the control flow is redirected to wherever the attacker wants it.

Listing 12: Machine code showing stack frame initialization.

```
# prolog
push %rbp
sub 0xff, %rsp
...
# epilog
mov %rbp, %rsp
pop %rbp
```

Note that the necessary instruction, which causes the vulnerability, may not be present for every single function. The swapping of the frame and the stack pointer can also be replaced by increasing the stack pointer with an appropriate value, using the `add` instruction (due to the stack's direction of growth). This leads to the same result, of the stack pointer being set to the address right below the saved frame pointer of the last stack frame. The *Clang/LLVM*-compiler uses the `sub/add` instruction for most of its stack frame management. But it may be possible that it switches to the register swapping in some cases, when optimizations are in use.

We do not introduce an additional exploit for this attack strategy, because the one from the previous example (see section 4.1) can be used the same way. Nothing changes except of the data structure, which is now the target of the exploit and is intended to be modified. SCADS is invulnerable against this attack mechanism, because the frame pointer is defined as control flow data. Therefore the saved frame pointers lie on the control stack, separated from any buffers, which may overflow. If no more complex actions are performed by an exploit to reach the control stack, without causing the program to abort, then it is not possible to modify the frame pointer by overflowing a buffer on the data stack. The scenario can be well compared to the first example we presented within the security analysis, where we already explained why the exploit fails to reach the control flow information on the control stack.

4.3. Indirect Pointer Overwrite

In 2000, Bulba and Kil3r presented an article [17] that addresses the exploitation of StackGuard and StackShield. As we want to compare SCADS with both of them, the content of this article is useful regarding the evaluation of SCADS' security. Especially one attack mechanism that was introduced by this article is of interest. It uses regular pointers within a function, under certain circumstances, to gain the ability of random write access to any address of the process' memory. This makes it possible, to bypass canaries without overwriting them. There was no precise name given for this kind of attack scheme, but we label it "*indirect pointer overwrite*".

Listing 13: Attack vector to exploit framework II.

```
char attack_vector_1[64];

// text segment address of "secret()"
char attack_vector_2[9] =
    "\x70\x06\x40\x00\x00\x00\x00\x00";

void set_attack_vector(){
  for(int i = 0; i < 32; i++)
      attack_vector_1[i] = 0x42;

  // overwrites the least-significant
      byte of the pointer "v_pointer"
  attack_vector_1[32] = 0x48;
}
```

Even though, canaries have been later on improved to protect against such exploits as well, in order to let SCADS compete with StackGuard and StackShield, we analyze the behavior of SCADS against this type of exploit and discuss the differences to concepts. Note that the exploit we present in this section, shall act as a wildcard for any type of stack smashing technique, which makes use of random access write ability, to perform its attack strategy. This time we set the precondition that ASLR is activated when the exploit is executed. As ASLR is not existent on FreeBSD at the time of this writing, the following test cases were compiled and executed on Linux only.

Listing 14: Exploit framework II (inspired by vul.c from [17]).

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

struct pack {
  char buffer[32];
  char *v_pointer;
};

void secret(){
  printf("S3CR3T HAS BEEN BREACHED!");
  exit(0);
}

void vuln_function (){
  struct pack p;

  p.v_pointer = p.buffer;

  // buffer overflow occurs here
  strcpy(p.v_pointer, attack_vector_1);

  // writing to previously set memory
      location
  strncpy(p.v_pointer, attack_vector_2,
      16);
}

int main (int argc, char **argv) {
  set_attack_vector();
  vuln_function();

  return 0;
}
```

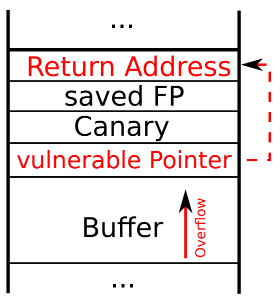
A sample framework allowing the execution of an indirect pointer overwrite exploit is shown with listing 14. The source code is taken from [17, vul.c] and displays a vulnerable use of the regular pointer `v_pointer` in function `vuln_function`. For a working exploit, this example needs at least two user inputs, handled the way as shown in listing 14. On the stack, a modifiable pointer is located, which is set to an address within the current stack frame. This pointer is then overwritten by the misuse of `strcpy`, which causes a buffer overflow in case the output buffer is not large enough, as we already know. Note that the key idea of this exploit is not to overwrite the return address itself by the caused buffer overflow (mostly because it is not possible successfully), but only to overwrite the regular pointer, which is stored on the stack, below the return address. The actual modification of control flow data happens afterwards, when the modified pointer is dereferenced, and the memory location is then overwritten, due to a write access using the second user input. In our example the use of `strncpy` on the pointer fulfills this necessary task.

Modern compilers nowadays may resort the order of local variables stored within a stack frame, leaving any buffers to be the topmost variables with all the rest below them. This would affect the exploit, because it would no longer be possible to overwrite the vulnerable pointer with a buffer overflow. Instead a buffer underflow (similar to the examples shown in [1]) would be needed to accomplish the modification. But buffer underflows are rare to meet, and we do not want to diverge too far from real-world relevant exploits. That is why we included the vulnerable pointer `v_pointer` as well as the `buffer` into a struct, which is then placed on the stack. Reordering is not performed on the internals of structs, therefore listing 14 fulfills all necessary conditions, explained in [17] to get the exploit working.

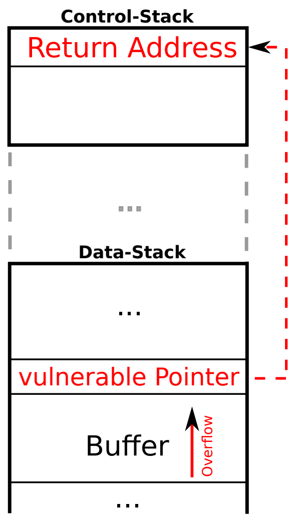
With the presented attack vectors (see listing 13) we want to overwrite the pointer with an address to the control flow data (the return address) on the stack, and then modify it using the second portion of the attack vector, to let it point into a code section, different from the original control flow. We have inserted the function `secret` which is actually never called by the regular control flow of the program. With this function we want to prove whether the exploit was successful or not. So we use the address of `secret` within the text segment as new value for the return address to be modified. It is easy to retrieve the address of the `secret` function, because the text segment is not randomized by ASLR[26]. By analyzing the compiled binary we can obtain the address we need and use it for the second attack vector. The exploit is then found to be successful when the `printf` message of `secret` appears on standard output. Note that the value of the `attack_vector_2`, shown in listing 13, is bound to our test case. Any recompilation of the binary would much likely change the address of the `secret` function and therefore the value we need to set for the `attack_vector_2`.

The complexity of the exploit does not longer allow us to use the same attack vector for all of our three test programs. It is rather necessary to use individual attack vectors to not lower the effectiveness of the exploit on each of the defensive concepts. In fact only the first attack vector (`attack_vector_1`), which causes the overflow and defines the address of the return instruction pointer, needs to be altered to fit the given concept. The second attack vector, containing the address to the code to be executed, shall not change, as long as the source code does not change.

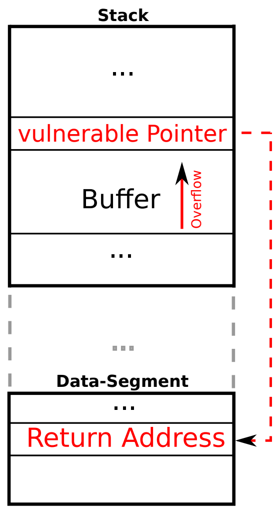
Nevertheless, in case it changes, the new value can be obtained by having a look at the binary's dump, which is of no relevance.



(a) Indirect pointer overwrite on **StackGuard**



(b) Indirect pointer overwrite on **SCADS**



(c) Indirect pointer overwrite on **StackShield**

Figure 8. Exploit using an indirect pointer overwrite on StackGuard, StackShield and SCADS.

Figure 8 describes the attack strategy on all three defensive concepts visually. One can clearly see that attacking StackGuard in contrast to StackShield and SCADS may be the easiest task in comparison, because of the relatively small distance between the vulnerable pointer and the return address. On the other hand, the defense of StackShield and SCADS seems to be very similar in this case. Both concepts do manage their persistent control flow data within a memory segment different from the one that contains the vulnerable pointer. Therefore the chance for a successful exploit depends on how simple it is, to find out the address of the segment storing the control flow data, to reach it and modify the return address.

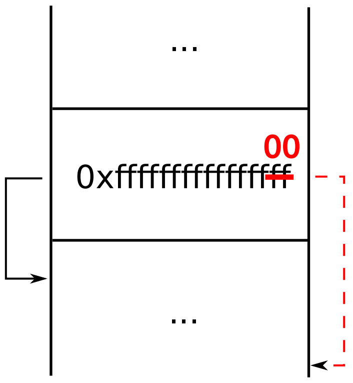
In the article[17], Bulba and Kil3r did not mention the presence of ASLR when trying to use the indirect pointer overwrite exploit. Though, we want to extend the analysis to an environment with ASLR activated for two reasons. First, Bulba and Kil3r have already proven the vulnerability of StackGuard against indirect pointer overwriting without ALSR, and we do not want to reinvent the wheel. Second, ASLR is a necessary component for the security of SCADS later on, regarding this example. So to apply the same standards, we also test the indirect pointer overwrite on StackGuard and StackShield with ASLR activated.

The attack mechanism is closely related to the *ret2ret* attack strategy[19, section 8.1], which can be applied when ASLR without DEP is in use. Except for the direction, to which the vulnerable pointer shall be modified. When using the *ret2ret* strategy, normally the shellcode is being placed below the modified pointer on the stack. With `strcpy` as present vulnerability, it is easy to modify the least-significant byte of such a pointer towards lower addresses, as `strcpy` automatically inserts a null terminator as last ASCII-character to the output buffer. This makes it possible to reach a NOP-slide [19, section 8.1], delivered with the overflowing payload.

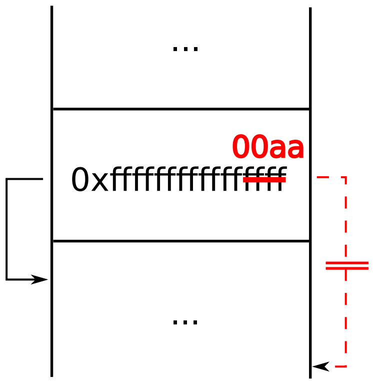
In our case, to be more precise in the case of SCADS and StackGuard, we want to modify the vulnerable pointer in such a way that it points to a location above itself, more specifically to point to the return address which definitely lies above the pointer on the stack. Altering the whole address of the pointer's value with the intention to hit the return address would be an impossible task, due to unknown addresses because of ASLR. So it is necessary to reduce the modification to as few bytes as possible, for a high chance to hit the target destination. Under these circumstances, the null terminator, which is inserted by `strcpy`, does not come in handy. Figure 9 shows some exemplary results of more or less well modified pointers.

We need to raise the least-significant byte of the pointer instead of lowering it. That is why we cannot let `strcpy` overwrite the least-significant byte with the null terminator, as the produced address would, by all means, be lower than or at least equal to the original. Thus, the produced address would become useless for the presented exploit, as it would fail to reach its destination. In case we try to manipulate the least-significant byte by ourselves, `strcpy` would then alter the second least-significant byte of the pointer with the inserted null terminator, and consequently lower the address once again instead of raising it. The only chance we have, is to set the least-significant byte to a static value and execute the exploit often enough, to let ASLR produce the type of address we need, to successfully raise it towards the return address, similar to the example shown by figure 9(c).

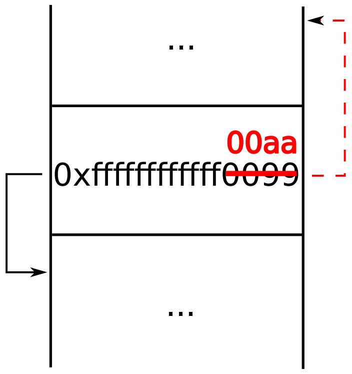
$$\begin{aligned} &vulnerable\_pointer = \mathtt{0x????????????00XX} \\ &with \quad \mathtt{"??"} \in [0, 2^8 - 1], \ \mathtt{"XX"} < attack\_vector_{LSB} \end{aligned} \tag{3}$$



(a) Exploit **lowering** the pointer



(b) Exploit **lowering** the pointer



(c) Exploit **raising** the pointer

Figure 9. Exemplary results of pointer modifications using `strcpy` to alter at most two least-significant bytes.

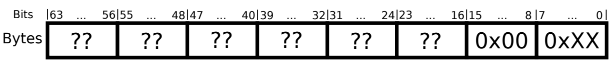

Bits |63 ... 56|55 ... 48|47 ... 40|39 ... 32|31 ... 24|23 ... 16|15 ... 8|7 ... 0|
Bytes
??
??
??
??
??
??
0x00
0xXX

Figure 10. Illustration which bytes, and in which way these bytes of a vulnerable pointer must be modified, for the exploit to succeed with StackGuard.

To be more specific, in case of StackGuard, we need the pointer to meet the conditions of equation 3 (also shown by figure 10 in detail). While anything above the second least-significant byte needs not to fulfill any condition because these bytes are not altered, the second least-significant byte must have the value `0x00`, because there the null terminator is inserted by strcpy, and the least-significant byte must be of a lower value than the one we insert into the attack vector to modify it. This way we would successfully raise the pointer's value to a higher address, but to hit the return address exactly, we would also need the difference between the two least-significant bytes of the vulnerable pointer and the self-constructed attack vector, to be the same as the distance between the return address' stack slot and the address the vulnerable pointer points to. $addr(return\_address)$ here denotes the location of the return address on the stack.

$$\begin{aligned} &|addr(return\_address) - vulnerable\_pointer| \\ &= \\ &|vulnerable\_pointer_{LSB} - attack\_vector_{LSB}| \end{aligned} \tag{4}$$

this can only be achieved by brute-force and the number of random bytes which need to fulfill static requirements, define the probability of the event to be met. As the address the pointer needs to hit as target is more or less different from the original pointer's value, depending on the defensive concept that is used, this also results in a more or less high chance for the brute-force attack to be successful. Intuitively, the exploit may therefore be most effective on StackGuard in theory. If all the described conditions are fulfilled, the return address is modified to point to the `secret` function, and its execution shall take place as soon as the `vuln_function` returns.

4.3.1. StackGuard.

We first want to take a look at the exploit's success or failure on StackGuard. As we explained, the exploit is very likely to not succeed on the first try, but it is necessary to execute it a couple of times to raise the chances for a successful exploitation. For this purpose we wrote a wrapper script which invokes the program a couple of times. The script invokes the StackGuard-protected binary 10000 times and just redirects the output of the program to stdout. In case the program does not exit successfully, a message is printed to indicate a segmentation fault. In case of a success though, we expect the output to show the phrase printed by the `secret` function. The binary that is being executed is the compilation of listing 14, using the attack vectors described by listing 13. In this scenario, the brute-force mechanism is twisted, as we do not alter the input data (i.e. the attack vectors), but leave them static for all executions. We simply let ASLR perform the brute-force mechanism, by the randomization of the stack's starting address, so that we have the chance to hit the needed, randomized address, which fits our attack vector precisely.

As a result, our indirect pointer overwrite exploit on StackGuard, using the script for 10000 invocations, has executed the `secret`-function eight times successfully, i.e. the exploit in its entirety was successful. Two times we have modified the canary, thus enabled StackGuard to notice the buffer overflow, two times we have modified some addresses erroneous, causing a segmentation fault and the rest of the 10000 invocations has not achieved a meaningful event. This can be already classified as success for the exploit, as we were able to manage to redirect the control flow. Nevertheless, we have tested a couple of more invocations and summarized the result in table 1 to strengthen the statistical value.

	success	detection	segfault
amount	477	262	262
percentage	0.0477 %	0.0262 %	0.0262 %

Table 1. Table listing the amount of events occurred after one million executions of the StackGuard exploit.

The second row of the table represents the percentage of the three possible events with 1 million invocations. The table shows that the number of detections and segmentation faults are closely related or are identical in this case. This can by explained by the fact that any detection leads to an abortion of the program, thus being caught by the wrapper script to print that a segmentation fault occurred. But in general, the exploit could also lead to a segmentation fault, by the modification of sensitive data different from canaries, which would then cause the program to terminate. In our example this did not happen even once. But more important is the fact that the testing sequence achieved 477 successful exploitations within 1 million executions. The second row of the table shows the percentage of the events in contrast to the number of tries, leaving the column, which indicates the successful tries, at far less than one percent. Nevertheless, regarding the very small execution time of the program, which may not only be of artificial nature if we think of background daemons with a high response-rate, this kind of exploit can be relevant in real-world matters.

4.3.2. SCADS.

We now take a look at SCADS as competitor. We use exactly the same exploit as we did for StackGuard, but to ensure the operability of the exploit, we need to alter the first attack vector when testing with SCADS. We need to ensure that the modified pointer points somewhere into the control stack, directly on a return address in the best case. To find out how exactly we must modify the attack vector to fulfill this condition (still depending on random factors because of ASLR), we take a look at the theory of the data stack allocation presented in section 2.1.2. We know that the data stack is initially set 64 MiB below the control stack, and afterwards its 24 least-significant bits are randomized. This knowledge suffices, to limit the location of the control stack, relative to the data stack, roughly. With the information from equations 1 and 2, equation 5 can define the rough limitation of the data stack (DS) location relative to the control stack (CS), considering a maximal stack extension of 8 MiB. Equation 6 contains the same information in "simpler" readable form.

$$\begin{aligned} InitAddress_{DS} + (4 * 2^{24} - 2^{23}) &\leq InitAddress_{CS} \\ &< InitAddress_{DS} + (4 * 2^{24} + 2^{23}) \\ InitAddress_{DS} + \frac{7}{2} * 2^{24} &\leq InitAddress_{CS} \\ &< InitAddress_{DS} + \frac{9}{2} * 2^{24} \end{aligned} \tag{5}$$

$$\begin{aligned} InitAddress_{DS} + 56MiB &\leq InitAddress_{CS} \\ &< InitAddress_{DS} + 72MiB \end{aligned} \tag{6}$$

The essential knowledge we can retrieve from these equations is that the data stack is at least 56 MiB away from the control stack initially (as long as the maximal stack extension is set to 8 MiB). Moreover we know that the control stack lies above the data stack. Therefore we can now define, that the vulnerable pointer we modify with the exploit, must be raised for at least 56 MiB to even come close to the control flow data on the control stack. Though the value must be greater in general, because the equations define the initial addresses of the stacks (at the end of the initialization phase), before data has been pushed onto them – which may eventually increase the gap of the stack pointers later on.

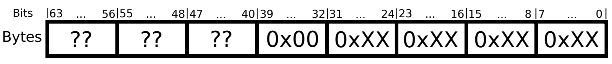


Figure 11. Illustration which bytes, and in which way these bytes of a vulnerable pointer must be modified, for the exploit to succeed with SCADS.

We now define how the attack vector must look like to have a chance for successful exploitation on the SCADS binary. As we need to raise the pointer for at least 56 MiB, we need to increase the fourth least-significant byte of it at the minimum. If we do modify less than the fourth least-significant byte, we are not able to reach the control stack at all. This leads us to a pointer adjustment defined by equation 7 and figure 11. The attack vector itself can be deduced from this equation.

$$vulnerable\_pointer = \texttt{0x??????00XXXXXXXX} \tag{7}$$

Similar to the attack vector we used for StackGuard (equation 3), the "XX" define the last bytes of the actual attack vector that do already overwrite the vulnerable pointer. The other bytes fill the buffer of our example framework and are therefore not mentioned here. The "??" are the bytes of the pointer that are not being altered, the "XX" are the four bytes we have to adjust to make it point to the control stack, and we want to recall that the null byte in the middle of the address occurs due to the behavior of `strcpy`, which places the null byte as terminator beyond the last (non-null) character that was copied. The bytes modified by the exploit can be chosen randomly, as we need to brute-force the execution anyway. Only the least-significant byte needs to be 8 byte aligned, as the placement of the return address to be hit, is also 8 byte aligned. The other three bytes can be chosen arbitrarily, and we let ASLR generate the correct, randomized address for the exploit once again.

	success	segfault
amount	0	1 000 000
percentage	0.0 %	100.0 %

Table 2. Table listing the amount of events occurred after 1 million executions of the SCADS exploit.

Table 2 shows the result of the one million executions of the exploit, this time with the SCADS binary. We used the same script for the brute-force attack as the one used for StackGuard, only the binary to be executed has been replaced in the source code of the script. There is no more *detection* column in the table, because SCADS does not check for buffer overflow occurrence. The result looks completely different from the one of the StackGuard test case as no successful exploitation took place and moreover all executions terminated with a segmentation fault. The reason for this result does not lie within an unwise choice of the attack vector, but the reason itself is the randomized gap between the control and data stack. ASLR does not randomize more than 32 bit of any segment's address and therefore, the fifth least-significant byte of the control stack was never zero. Moreover the address we generated with the attack vector always points into a non-allocated memory region, due to the null byte in the middle, and so causes the segmentation fault on every execution.

The only remaining option for the exploit to work on SCADS, is a brute-force attack that modifies more of the vulnerable pointer than just the lowest part of it. As the first four bytes of the address are known to be `0x00007fff` one must set the remaining 4 bytes to any randomly chosen, 8 byte aligned value and try to execute the exploit often enough to raise the chances of the return address to be located at the chosen address. However, this leaves the exploit with a chance for success that is equal to the probability of choosing a number between $[0, 2^{29}]$. Hence, a successful exploitation can be considered highly unlikely. We repeated the test, by setting the address in the attack vector to the value `0x00007fff854c5f48`, which is completely randomly chosen, and let ASLR do the brute-force attack for us once again. Several million executions yielded the exact same result as shown by table 2, though. Therefore we were not able to attack SCADS with an indirect pointer overwrite by the execution of a brute-force attack.

As a conclusion we can say that this exploit is unable to harm SCADS in our test case. The success is dependent on the probability of hitting the randomized address of need. However, this probability is rather low due to the randomized distance of the control and data stacks, making it necessary to perform a long enduring brute-force attack.

4.3.3. StackShield.

Finally, we inspect the behavior of StackShield towards the presented exploit. Once again we need to alter the attack vector accordingly. We modified the brute-force script to now alter the first attack vector in each iteration of the loop. In detail, we now pass the 4 bytes to our exploitation framework, that do overwrite the value of the vulnerable pointer, thus the address where the control flow data is supposed to lie. As already mentioned, the data segment is not affected by ASLR, as we do not generate position independent binaries. So our task is to reach the data segment, and within the data segment the area that stores the saved return addresses, to overwrite it and consequently to manipulate the control flow of the program successfully.

Although the data segment is not affected by ASLR, and therefore has a static address, the address may vary when a binary is recompiled and may therefore be unknown to an attacker. Because of that one must somehow find out where the data segment lies. We rely on the fact that the address (or even the top) of the text segment is known. This is a reasonable assumption, because if an attacker owns the binary he wants to attack (locally for example), then he can dump it and read out the necessary address. The attack strategy is then altered as follows. We start on a 4 byte aligned address, preferably at the end of the text segment to speed up the brute-force attack, and then increase the address by 4 byte every time we execute the exploit, until we succeed, which means that we managed to modify the redundant return address area.

Note that we were not able to successfully compile binaries with StackShield on a 64-bit system, as it seems to not be supported yet. Because of that, we performed the test case on a 32-bit Linux system, which is why we are using 4 byte addresses instead of 8 byte addresses.

We were successfully able to overwrite the return address and redirect the control flow to the execution of the `secret` function, after hitting the safe storage area of the data segment at the illustrated address. This is clearly a prove for the vulnerability of StackShield against indirect pointer overwriting exploits. The main reason for the lack of protection against the presented exploit originates from the missing randomization of the data segment's base address, though. In case the data segment would also be affected by ASLR, we may probably define StackShield as equally secure as SCADS against this type of exploit, because the only difference between these concepts would be the direction towards which the vulnerable pointer must be modified, to point to the control flow data.

Summarizing, SCADS was the only defensive concept in this case that could successfully prevent an exploitation through an indirect pointer overwrite attack. Both StackShield and StackGuard were unable to secure the integrity of the return instruction pointer, thus making it possible to redirect the control flow of the program. Although, it is necessary to mention that we used StackGuard with *random canaries*[22, chapter 4]; a more effective protection would be the use of *xor canaries*[22, chapter 4]: “The XOR random canary method was introduced in StackGuard version 1.21” [22, chapter 4].

But it seems that XOR canaries are not yet implemented in the GCC stable release. *GCC 4.6.3* produces random canaries only when using the flag `-fstack-protector`. XOR canaries would be more useful against an indirect pointer overwrite exploit, because they would not allow to alter the return address without knowing the value of the canary, because an additional XOR encryption is performed with XOR canaries. Nevertheless, our task is to present an efficient protection mechanism that does not need to include any further machine code. To accomplish this task, even though the impact on the performance may be negligible for programs with short execution times and regarding this aspect, SCADS was able to compete with StackShield and StackGuard.

4.4. Format String Attacks

In this section, we introduce another example of an exploit, though more a vulnerability which we want to test our three test candidates on. Format string attacks were widespread due to the dirty production of source code, as format string vulnerabilities originate from programming errors solely. Such vulnerabilities can be abused in many different ways, well described in [24, chapter 3]. We are not testing all of these presented attack variations, but instead focus on the visualization of process memory, using format string exploits. The extraction of memory information from the stack or any other location is often a very gladly used instrument, to prepare for a more serious exploit, or to fix non-working exploits, with the help of newly gained knowledge. Therefore it is also an important task for a protection mechanism, such as SCADS, to prevent attackers from being able to read sensitive information, by hiding it or making its location unknown.

Listing 15: Attack vector to exploit framework III.

```
char attack_vector[32] =
    "%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p"
```

Listing 16: Exploit framework III.

```
#include <stdio.h>

int main(int argc, char **argv){
  char format[64];

  snprintf(format, 32, "%s", argv[1]);
  printf(format);

  return 0;
}
```

We now want to demonstrate how easy it can be to retrieve useful information from process memory, that should not be available to attackers, and we want to emphasize that SCADS is able to significantly complicate the process of information retrieval for format string exploits. Listing 16 shows an exemplary format string vulnerability within a sample program, inspired by the example shown in [24, chapter 3]. Actually, modern compiler would generate a warning, when compiling the code of listing 16, due to the missing arguments to the `printf` function. However, we want to concentrate on the benefits, an attacker can gain, when assuming that such vulnerabilities find their way into either SCADS-, StackGuard- or StackShield-binaries.

The mechanism of an exploit abusing a format string vulnerability is rather simple. Format string exploits, as the name tells, rely on the abuse of format strings, which can give great power to an attacker when gaining full control over them. In our example, the focus lies on the `printf` function, which receives a format string as first parameter. Format strings, as parameters, are generally in use with the `ellipsis` parameter ("…"), which eventually allows the function to take a non-restricted number of parameters, whose usage and the context is defined by the format string itself. Any `%`-character with another attached character of arbitrary value, implicitly defines a parameter, that needs to be passed to the function outside of the format string. The most important thing about it is, that the format-function expects as many parameters as defined by the format string. If the number of defined parameters in the format string is lower than the ones passed, then the rest of the parameters is just ignored, i.e. not used. But more important, if the number of defined parameters is higher than the ones passed, then the missing parameters are anyway read from the stack, where they are expected to lie. This leads to the fact, that whatever lies on the position of the missing parameters on the stack is used and can be controlled by the attacker if he has the power over the format string.

In the framework of listing 16, the format string that is passed to `printf` is completely user-defined (or attacker-defined). The only thing happening before it is passed to `printf` is that it is being stripped to 32 characters. So the format string that the attacker has control of in this example, can at most have 32 digits. With the power to control the content of the format string, we try to read useful data from the stack with our attack that gives hints on the location of return addresses or similar. By inserting a couple of `%p` into the attack vector, as described by listing 15, we gradually read 8 bytes step by step on 64-bit systems.

4.4.1. StackGuard. From section 3 and figure 6 we remember the memory layout of StackGuard. So with our format string exploit, apart from reading local variables from the stack, we are also able to read canaries, saved frame pointers (if present) and return addresses. This is a huge advantage to an attacker, as the regular protection given by canaries (apart from terminator canaries[22, section 2.2]), is dependent on the missing knowledge of an attacker, regarding the exact value of the canary. This way an attacker can find out the exact value of the canary and may use this knowledge for a continuing exploit, which imitates the canary in its attack vector to prevent a buffer overflow from being detected, for example. Furthermore, the saved frame pointer can be used by an attacker to easily specify the location of self-placed shellcode on the stack (in case DEP is not active). And the read return address could be used to find the location of the text segment, in case of a binary with position independent code and ASLR activated. Eventually this illustrates that there lies many information on the stack of a binary, compiled with StackGuard activated, that can be well used to support any following exploits.

4.4.2. StackShield. The stack layout with StackShield is not much different than the one with StackGuard being enabled, except for the missing canaries. For that reason, the useful information that can be retrieved via a format string exploit is also much the same. One can read the saved frame pointer, as well as the return address. Both can be useful for reasons that have been explained before. But there is one good aspect about StackShield in this case as there is no information placed on the stack, which tells about the location of the storage area in the data segment (containing redundant return addresses) and therefore such information cannot be extracted with a format string exploit. In the end, there is no direct way to gain an advantage for the permanent modification of any return address by using such an exploit.

4.4.3. SCADS. We already know the memory layout of a SCADS compiled program. All control flow data is separated from the regular data by a large unallocated memory gap, lying between the control and the data stack. So if the format string exploit is only able to read from the data stack everything is fine and no advantage can be gained. On the other hand, if the exploit allows to read from the control stack that is fatal, because all useful information lies on the control stack and even the control stack's location itself is supposed to stay unknown.

Unfortunately, if we use the `printf` function of any legacy library not protected by SCADS, the exploit allows to read from the control stack because the legacy function uses `%rsp` as stack pointer, thus the function reads its parameters from the control stack. On the control stack, the exploit may find return addresses and saved frame pointers of the data stack which are both sensitive data that is useful to know as explained above. But at least, as we do not maintain a frame pointer for the control stack, there is no direct way for the exploit to find out the control stack's location, even though it reads from the control stack itself.

The long-term solution to circumvent the problem that `printf` reads from the control stack in case of a format string exploit, is to port the LibC by compiling it entirely with SCADS. This way the `printf` function no longer uses the stack pointer of the control stack but the one of the data stack. That leads to the improvement that any executed format string exploit can only read from the data stack, where it finds no useful information in form of a canary or frame pointer. The data stack stores only buffers, local variables and saved data registers, leaving no hint for the location of the control stack and therefore no support to modify control flow data can be earned. So porting of the LibC to SCADS is a necessary and helpful task, which we already fulfilled on FreeBSD for this and other various reasons (also see section 7).

4.5. Summary

SCADS is a concept that focuses on the security of the return instruction pointer, which is a potential vulnerability that is present in many programs. And so we presented exploits which target to modify either the return instruction pointer or a saved frame pointer. Exploits that succeed without manipulating a return address or frame pointer at all, might not be prevented well by SCADS. But by the protection of the return instruction pointer in the first place, SCADS is capable of protecting against many exploit, including *Return-Oriented-Programming* [25], *ret2ret*, *ret2libc*, and more [19], which use this control flow data as initial starting point for the execution of malicious machine code.

Nevertheless there are also exploits that successfully modify the control flow without touching the return addresses. For example, exploits that focus on the modification of the *Global Offset Table (GOT)* that supports position-independent code [8], or any *virtual function table (vtable)* that implements dynamic dispatch in C++ [12, sec. 2.1]. SCADS is yet not intended to protect against this kind of exploits, and until it is not extended to do otherwise in the future, we suggest to use an additional protection mechanism along with SCADS, which completes the defense that SCADS supplies.

5. Performance Evaluation

Besides our main task to guarantee a high security level with SCADS, our second goal is to negatively affect the performance of any newly compiled program as little as possible. Accordingly, this section now analyzes the behavior of SCADS when it comes to execution times. In general, gaining the highest security level and the best performance is mutually exclusive. There is always a more or less large trade-off between these two goals. With SCADS we want to achieve a compromise which addresses both aspects in a satisfying, symbiotic way.

When SCADS and StackGuard are compared regarding their performance there is an essential difference of which can be observed. Thus, we want to define two different categories of speed limitations:

- **Static performance overhead**: An overhead which does only occur at the start or the end of a process or user level thread. This type of performance overhead is therefore static in the meaning of being independent of the effective runtime of the running process. It does only affect the performance of the whole process once on each execution and hence the overall overhead that is produced by a static performance overhead can be approximated.

- **Dynamic performance overhead**: An overhead which is part of the regular control flow of a process or user level thread, which excludes the initialization and the finalization mechanics of a process. This makes the overhead become a repeatable constraint. It can be part of loops inside the program, which cause the performance overhead to rise in a dynamic way, depending on the runtime and execution control flow of a process. Estimating the impact of dynamic performance overheads on the overall performance of a program is a much more complex task, with the need of a control flow analysis to find out how often a performance-decreasing instruction is reached.

5.1. Theoretical Approach

According to the nomenclature introduced above, we can classify StackGuard and SCADS in theory before we analyze their performance on practical examples. StackGuard inserts various machine instructions at the beginning (prolog) and ending (epilog) of a function to be compiled. These instructions are therefore executed, every time a new stack frame is created and removed. Obviously the performance overhead created by StackGuard is therefore dependent on the total number of calls to functions with incorporated canaries that are made. So StackGuard's performance overhead consists of dynamic performance overhead.

On the other hand, when we take a look at SCADS, we know that for its implementation an extensive initialization phase has to be performed. This initialization phase allocates the data stack and is only executed once when a SCADS compiled program is started, therefore it is a pure static performance overhead. Apart from the initialization phase, SCADS does not include further machine instructions into the control flow of a program in the regular case, but alters the behavior of various machine instructions to fulfill the usage of the control and the data stack. The only dynamic performance overhead that occurs is the storage of regular data registers on the data stack and additionally the stack extension for large stack frames, but this is an operating system dependent issue (as explained in section 2.2.3). As we cannot use `push` and `pop` instructions on the data stack, these are simulated with `sub`/`mov` and `mov`/`add` instructions. From the official Intel optimization manual for their x86-64 architectures [6, sec. C-25ff. / table C-19 and C-19a], we can retrieve the information, that `push` and `pop` both have a latency of approximately 1.5 clock cycles, while `mov` combined with `add` or `sub` has a latency of 2 clock cycles on modern Intel architectures. Due to the fact that all modern processors do use an execution pipeline [15] for the queued instructions, the overhead that may occur when two instructions need to be fetched and decoded instead of one, would also result in an additional processor cycle in the ideal case. So in case a function stores regular data registers on the data stack, the performance overhead, even though it is dynamic, is small compared to the non-SCADS case.

Summarizing, StackGuard's performance overhead solely consists of dynamic overhead while SCADS' overhead is mainly based on static overhead, with a rare component of dynamic overhead. Hence, we can assume that SCADS may preserve the performance of a program better than StackGuard when it has a rather long execution time, while StackGuard-compiled programs may be faster on short execution times.

5.2. Practical analysis

To verify our theoretical assumptions, we now move on to the practical performance analysis with appropriate examples, which shall illustrate the behavior of the two concepts and the raw compilations without any security measures activated. For our measurements, we do not choose common real world software from *SPEC CPU2006* [9], for example, because in theory, it is already known where the performance bottle-necks of SCADS and StackGuard lie. To illustrate the impact of these properties much clearer, we deemed it appropriate to take measurements with self-constructed test cases focussing on the strengths and weaknesses of SCADS and StackGuard. The results of these test cases can be applied to real world programs accordingly.

Listing 17: Fibonacci code sample.

```
#include <stdio.h>
#include <stdlib.h>

long long fib(int val){
  if(val == 0) return 0;
  if(val == 1) return 1;
  else   return fib(val-1) + fib(val-2);
}

int main(int argc, char **argv){
  printf("%lli\n", fib(atoi(argv[1])));

  return 0;
}
```

5.2.1. Fibonacci as Exemplary Test Case.

At first, we want to present a simple code sample we compiled with the four different configurations to analyze the number of generated instructions and the time of execution. The sample is illustrated by listing reflist:fibonacci. It fulfills the computation of any specified *fibonacci number*, by calling the function `fib` recursively. Consequently, the `fib` function is the part of the code which takes up the lion's share of the program's execution time, depending on the input parameter. Because of that, we focus on the analysis of the `fib` function and ignore the rest of the code, though we keep in mind that the overall execution time depends on every single machine instruction of the program, including the initialization phase in the SCADS case.

	number of instructions
GCC	24
GCC (StackGuard)	30 (+1)
Clang	24
Clang (SCADS)	24

Table 3. Listing of the number of machine instructions generated for the fibonacci function from listing 17 by GCC, GCC(StackGuard), Clang and Clang(SCADS).

In the appendix section, the x86-64 assembly of the `fib` function of each of the four compilations can be reviewed by taking a look at the files *fibonacci_gcc_asm.txt*, *fibonacci_stackguard_asm.txt*, *fibonacci_clang_asm.txt* and *fibonacci_scads_asm.txt*. The four files were created by the usage of `objdump` and may help out to identify the number of instructions that were generated. Table 3 lists the information about the number of instructions every compiler configuration generated during compilation. The number of machine instructions generated by the GCC-compiler rises significantly when StackGuard is activated. On the other hand, the Clang-compiler produces the same amount of machine instructions in this example, no matter whether SCADS is activated or not. Even though some of the instructions are rarely executed, due to the fact that *if-conditions* are present in the sample code, this table provides a first indicator on the possible performance advantage of SCADS towards StackGuard.

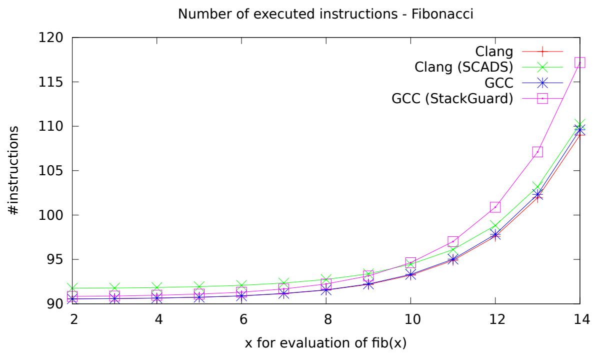


Figure 12. Number of executed instructions when running the program defined by listing 17.

To further analyze the performance, we present graphs which display the number of executed instructions and the overall execution times, in dependence of the input parameter to the `fib` function. Figure 12 displays the number of instructions (excluding those executed by the kernel) that are executed when the program is run with the corresponding parameter. Initially SCADS executes more instructions than the other three binary versions, which is caused by the initialization phase. But later on, approximately at a value of 10 for the input parameter, the influence, produced by StackGuard's canary management is large enough to outrun SCADS in terms of executed machine instructions. From then on the slope of the curve, which represents StackGuard, rises significantly, while the SCADS curve stays close to the ones of GCC and Clang, because it does not produce any further performance overhead than the static one from the initialization.

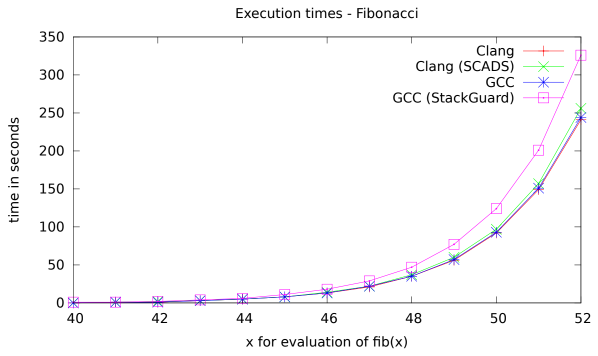


Figure 13. Execution times of the program defined by listing 17.

Of course, the number of executed instructions is closely related to the overall performance/execution times we have measured. On figure 13 the execution times of the program defined by listing 17 are displayed from parameter 36 unto 52. It is clearly visible that the StackGuard curve departs significantly from the three other curves represented by SCADS, GCC and Clang. This result can be ascribed to the values presented by table 3 and figure 12 earlier. All but the StackGuard compilation share the same number of machine instructions, generated for the `fib` function and the raised number of instructions is a reason for an emerging performance overhead, causing the StackGuard program to be up to almost 80 seconds slower than the other three versions.

This result verifies our thesis that StackGuard's performance overhead rises significantly with a high number of function calls, while SCADS' performance overhead is constant and arises from the initialization phase. Though the overhead caused by the initialization phase is not visible in figure 13, because it lies in the range of microseconds, which cannot be measured reliably, due to noise issues. It can only be seen implicitly, by having a look at the raised number of instructions that are executed, as shown on figure 12.

5.2.2. Fibonacci with Included Handicap. From section 2.2.3 we know about the special behavior of the data stack extension, when it comes to the management of large stack frames on Linux. There we have already mentioned that the mechanism may lead to a performance overhead. The example we present in this section shall address this thesis, to show how large the impact on the performance may be when a stack frame has a much larger size than usual. Although this is a Linux-specific problem, we want to analyze the performance influence of it, for the sake of completeness. Listing 18 shows a modified version of the fibonacci sample from listing 17.

Listing 18: Fibonacci code sample with handicap included.

```
#include <stdio.h>
#include <stdlib.h>

long long fib(int val){
  char dummy[4*4096];
  dummy[0] = 42;

  if(val == 0) return 0;
  if(val == 1) return 1;
  else   return fib(val-1) + fib(val-2);
}

int main(int argc, char **argv){
  printf("%lli\n", fib(atoi(argv[1])));

  return 0;
}
```

The code framed by oval boxes, represents the changes that have been made to the original fibonacci function. We have artificially added a buffer of size $4 * 4096$, which has actually no other purpose than increasing the amount of instructions that are executed at runtime. For that reason, the additional instructions, and with them the whole test case, are here entitled as a handicap, because they do only intend to decrease SCADS' performance for analytic purpose.

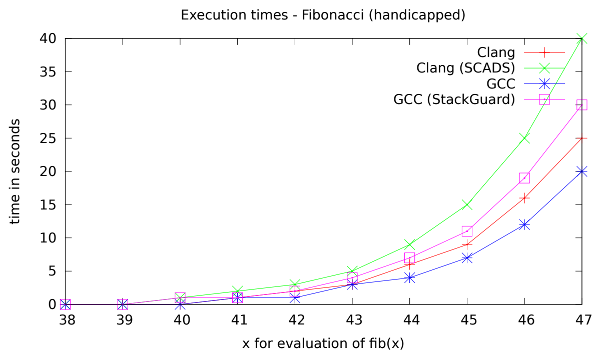


Figure 14. Execution times of the program defined by listing 18.

When the source code is compiled with the SCADS-enabled compiler version, the stack frame allocation of the `fib` function becomes more complex than with any of the other compiler configurations of our test environment. As explained in section 2.2.3, the data stack must be extended in steps of single pages, to allow the correct allocation of guard pages by the operating system. This leads to the result that the stack extension needs to be handled by 9 instructions instead of just 1 in this case. The generated assembler code can be viewed by taking a look at the file *fibonacci_scads_handicapped_asm.txt* in the appendix section. The SCADS compiler builds a chain of machine instructions which decrease the stack pointer by a page and perform a write access at its current position successively, until the needed stack frame size has been achieved. This chain of instructions affects the performance negatively when performed often enough, because in the non-SCADS case, the stack extension can be achieved with one single instruction. The way SCADS handles large stack frames is the only considerable property which introduces a dynamic performance overhead, excluding the overhead caused by the `push`/`pop` simulation, which is very small in comparison. Apart from that, SCADS' performance overhead is almost completely static, which is responsible for the very satisfying execution times of SCADS in general.

To demonstrate the performance overhead at its limits, we have once again chosen the fibonacci function for our test case. This time we have measured the execution up to the input parameter 47 to illustrate both, the asymptotic tendency of the four curves, as well as the turning point, from where the execution with SCADS consumes significantly more time than with StackGuard. Figure 14 shows the performance results of the execution of the program, defined by listing 18, with the four compiler configurations we have already used before. Up to a parameter value of about 40, the execution time with SCADS is very close to the one with StackGuard, even though the measurements are a not very stable due to the very short execution times in this range. Starting at value 41, the gap between the curves of SCADS and StackGuard begins to grow in size. At parameter value 47 we observe an execution time of SCADS which accounts for about 130% of StackGuard's.

The behavior of SCADS regarding the performance is much worse with this test case than the one of the other three compiler configurations. The lack of performance can be attributed to the stack extension mechanism only, though. Remember, that the graph of figure 13 shows the performance results on the non-handicapped version of the fibonacci program, where SCADS' performance was very satisfying. It is thus proven that the complicated stack extension, which SCADS must use for large stack frames on Linux, is its worst bottle-neck. It is necessary to emphasize, however, that the presented evaluation represents a *worst-case scenario* for SCADS, and does not represent the expected performance in general cases. On FreeBSD, the behavior regarding the stack extension is the same for small as well as for large stack frames. We did not present any results for example listing 18, measured on FreeBSD, because it does not cause a performance overhead as no further machine instructions are generated compared to listing 17.

5.2.3. Run-Length Encoding as Test Case.

So far we have presented rather artificial examples which do intend to show the performance behavior of SCADS in extreme scenarios, due to the amount of recursion that was part of the test cases. We now have a look to another example which does not include recursion. Listing 19 illustrates a program with more real-world relevance, to demonstrate how SCADS behaves in general situations. It is a simple form of a run-length encoding algorithm which receives a file name through the command-line, and stores the compressed result of the input file in another file. To be more specific, the program calls the function `poll`, which reads 256 bytes of data from the input file, until `EOF` is reached. The read data is then further processed in the while loop of the main function. Although the buffer `local_buf` is not needed, we inserted it artificially because real-world examples may contain functions with buffers, which therefore are protected with a stack canary by StackGuard.

	file size (GiB)	exec times (s)
GCC	1	24.2
GCC (StackGuard)		23.6
Clang		24.3
Clang (SCADS)		24.5
GCC	5	148.2
GCC (StackGuard)		148.0
Clang		148.7
Clang (SCADS)		148.5
GCC	10	361.2
GCC (StackGuard)		360.6
Clang		367.7
Clang (SCADS)		354.4
GCC	15	573.7
GCC (StackGuard)		563.8
Clang		591.5
Clang (SCADS)		551.6

Table 4. The execution times of the run-length-encoding from listing 19 for different sizes of the input file.

Listing 19: Run-Length Encoding code sample.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUF_SIZE 256

typedef struct record{
  char current;
  long counter;
} __attribute__(( packed)) record ;

int poll(char *buf, int n, FILE *stream){
  char local_buf[BUF_SIZE];

  int ret = fread(local_buf, 1, n,
      stream);
  strncpy(buf, local_buf, BUF_SIZE);

  return ret;
}

int main(int argc, char **argv){
  FILE *input = fopen(argv[1], "r");
  FILE *output = fopen("out.dat", "w");
  char buffer[BUF_SIZE];
  record rec;

  rec.current = 0; rec.counter = 0;

  while(1){
    int ret = poll(buffer, BUF_SIZE,
       input);
    for(int i = 0; i < ret; i++){
      if(buffer[i] &=& rec.current)
          rec.counter++;
      else{
        if(rec.counter != 0) fwrite((
            const void *)&rec, 5, 1,
            output);

        rec.current = buffer[i];
        rec.counter = 1;
      }
    }

    if(ret < BUF_SIZE){
      fclose(input); fclose(output);
      break;
    }
  }
}
```

By the nature of the run-length encoding program, the execution time is dependent on the size of the input file, which shall be compressed. For that reason we have generated four files of different size (1, 5, 10 and 15 GiB), by reading data from `/dev/urandom`. We have once again used the four compiler configurations (GCC, Clang, StackGuard and SCADS) for this test case and measured the execution times of the program with each of the four different files as input. The results of the measurements are contained in table 4 numerically, and in figure 15 graphically.

From the results we can see, that the execution times of all four compiler configurations do not differ significantly. In fact, SCADS seems to be the fastest contestant in this case, although only slightly. But this observation may be bound to the large amount of *IO-activity* that must be performed when executing the compression program defined by listing 19. The large files in the range of multiple gigabytes, we use as input to our test environment, cause the need for many read accesses to the hard drive, eventually resulting in the high percentage of IO activity. One may think, that the IO activity overshadows the actual CPU performance, but due to the complex caching hierarchy implemented in modern processors such as the one we used for our measurements, and the accompanied pre-fetching strategies that are performed in the background, this prediction is not completely valid. The results may also reflect the IO activity, but they can nevertheless be used to analyze

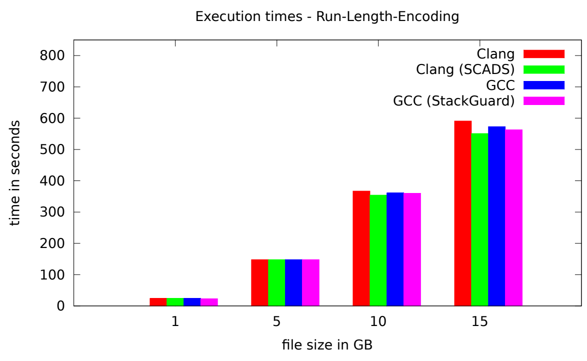


Figure 15. Execution times of the program defined by listing 19.

the overall performance of our four contestants, including the CPU performance.

6. Memory Efficiency

We now want to have a look at the memory efficiency of SCADS. The design of SCADS is intended to split up the data of the unified stack, without generating any new data. In detail, the control stack takes the return instruction pointers and saved frame pointers from the unified stack, while all other data remains on the data stack. As long as no frame pointer is used for the control stack, which would make need of storing it on the control stack itself, the amount of data managed by SCADS is invariant compared to the unified model. But one must distinguish between *allocated* and *used* memory. The data stack must be allocated during the initialization phase, with a minimum of one page size. So initially instead of the size of the unified stack, SCADS has an additional memory occupation of one page size for the newly allocated data stack. However, this stack is filled during runtime, as more data is allocated on the stack, thereby decreasing the memory overhead in comparison to the unified model. In contrast, the control stack is already allocated by the kernel, but it stores only a small amount of data per stack frame. That is why the control stack is unlikely to fill up the memory that has been allocated for it, with regular programs (programs using recursion may be an exception). Figure 16 schematically demonstrates the initial memory usage of the control and the data stack in a SCADS runtime environment. The shaded, unused memory gets filled step by step when the stacks grow during runtime.

In addition, the action of stack alignment may cause differences in the process of memory allocation when SCADS is used. By default, 16 byte alignment is enforced for stack frames to allow the correct functionality of various machine instructions on the x86 architecture, which require such an alignment. For example, this is a necessity for the use of 16 byte floating point registers, as the system bus raises an exception, if such instructions are used with not correctly aligned memory addresses. Therefore, SCADS also preserves

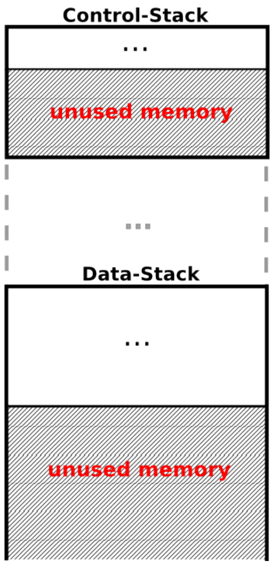


Figure 16. Schematic memory usage of control and data stack after the initialization phase.

this alignment for the data stack, if not defined otherwise by the user. The control stack, on the other hand, needs not to be further aligned at all, because the instructions used on addresses of the control stack require 8 byte alignment at most. Only in case we want to use legacy libraries (or general legacy code), which use the control stack as general purpose stack, a 16 byte alignment may be needed and can be selected by the user, by passing a command-line flag (see section 2.3.3).

As the return addresses and the saved frame pointers are missing on the data stack, the alignment may or may not have to be adjusted appropriately. To be exact, in case the frame pointer of the data stack is in use, there is no difference between the unified and the data stack, because the return address and the saved frame pointer both consume 16 bytes together on a 64-bit system. Thus moving these 16 bytes to the control stack leaves the data stack still 16 byte aligned, implicitly. But if the frame pointer is not used, then the resulting gap after moving the return address to the control stack is 8 bytes large, which may cause the need for an alignment enforcement in case the amount of bytes stored in the local area of the stack frame is not a multiple of 16 already.

This theoretical thought clarifies that SCADS may produce an 8 byte memory overhead for stack frames, when the data stored in the local area is not a multiple of 16 bytes, as already explained. But on the other hand, this condition must be applied to the unified stack in a modified way as well. Any stack frame of the unified stack does need a re-alignment in case the data of the local area does not sum up to a multiple of 8 bytes with the additional condition, that it is no multiple of 16 bytes. Equations 8 and 9 describe the mentioned conditions in a more formal way. The here defined variable $framesize$ contains all bytes stored in the corresponding stack frame, excluding the return instruction pointer (if even present on the stack).

$$\begin{aligned} &DS.framesize|16 \;\Rightarrow\; \text{no re-alignment}\\ &\text{otherwise} \;\Rightarrow\; \text{re-alignment} \end{aligned} \tag{8}$$

$$\begin{aligned} &(US.framesize + RIP_{size})|16 \;\Rightarrow\; \text{no re-alignment}\\ &\text{otherwise} \;\Rightarrow\; \text{re-alignment} \end{aligned} \tag{9}$$

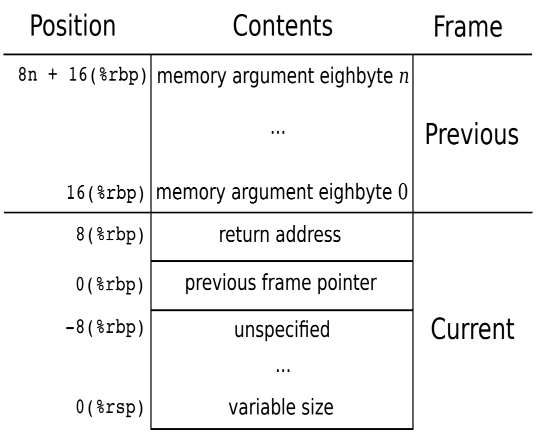
The resulting core statement of this reasoning is that there may be cases where a re-alignment is needed for the data stack but not for the unified stack on some stack frames, and there may also be situations where the scenario is twisted vice versa. In the end it depends on the size of the data stored in each frame only. This includes the local area with its local variables, buffers, and the stored registers that have been pushed onto the stack. If the size of this composite data does not fit any of the equations 8 or 9, the corresponding concept needs a re-alignment for the given stack frame.

7. Compatibility Issues

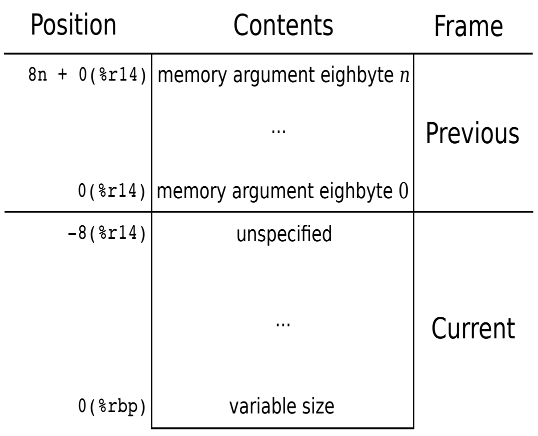
By the term *legacy code* we refer to external machine code that has not been compiled with the same compiler or enabled security flag, like the one used for compiling an associated source code of a program. In our case, an example for incorporating legacy code into a binary could be achieved by compiling a source code file with the Clang/LLVM-compiler and SCADS enabled, while linking the *standard LibC* to the binary, which has been compiled with a GCC version that does not support SCADS at all. This results in mixed machine code and the important question that comes to mind is whether the machine code, that has been compiled by different entities, is fully compatible to each other to guarantee correct functionality.

Even though, to fulfill a fully consistent security level by the usage of a compiler-based security mechanism like SCADS, it is necessary to compile all of the code a binary uses with the security feature enabled. But nowadays, common software often uses different libraries, and moreover the used libraries are dependent of further libraries, which leads to a long chain of code to be linked to a binary either statically or dynamically. Therefore, in general it is rather impractical to recompile all externally linked libraries at once. That is why it is an important task to analyze the compatibility of SCADS-enabled machine code to arbitrary legacy code. In terms of anticipation, our analysis has yielded that there are three types of incompatibilities to legacy code that inhabit SCADS:

- Incompatibility to legacy functions with more than six parameters
- Incompatibility to legacy functions, which take a SCADS function as call-back parameter
- Incompatibility to legacy functions, which need specific alignment (to handle floats for example)



(a) Stack frame representing the calling convention according to the **AMD64** ABI with `%rsp` as stack pointer and `%rbp` as frame pointer.



(b) Stack frame representing the calling convention according to the **SCADS** ABI with `%rbp` as stack pointer and `%r14` as frame pointer.

Figure 17. Calling conventions of AMD64 and SCADS.

7.1. Incompatible Parameter Amount

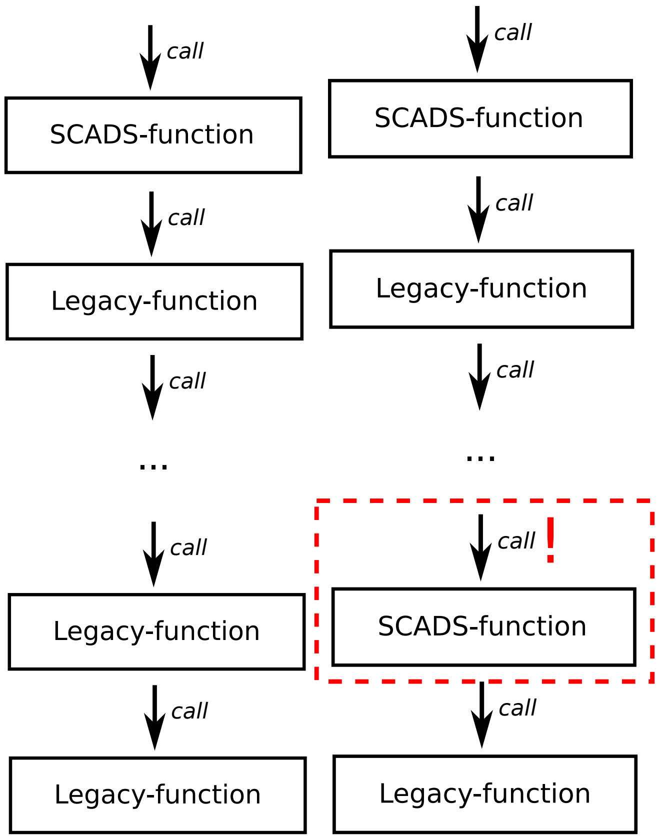
The way SCADS is designed and implemented, it changes the calling convention defined by [3, sec. 3.2.1, fig 3.3]. Figure 17 and table 5 show the different calling convention of SCADS compared to the AMD64 calling convention from [3].

It is clear to see that both calling conventions are identical up to parameter six. All of the parameters from one to six are handled with SCADS via the same registers like with AMD64. Hence, there is no change in behavior with a SCADS-binary when only legacy functions with less than seven parameters are used. Consequently, SCADS is compatible to legacy code that contains functions with at most six parameters. If a legacy function with seven or more parameters, however, is used by a SCADS-enabled program, then the called function wants to read stack parameters by referencing `%rbp`. In our implementation, `%rbp` is used as the stack pointer of the data stack rather than a base pointer. Thus, the called function uses the stack pointer of the data stack to read its parameters, but the offsets to the frame pointer referencing the parameters

	AMD64	SCADS
parameter 1	*%rdi*	*%rdi*
parameter 2	*%rsi*	*%rsi*
parameter 3	*%rdx*	*%rdx*
parameter 4	*%rcx*	*%rcx*
parameter 5	*%r8*	*%r8*
parameter 6	*%r9*	*%r9*
parameter 7	16(*%rbp*)	0(*%r14*)
parameter 8	8 + 16(*%rbp*)	8 + 0(*%r14*)
	...	...
parameter n	$8(n-7)+16(\%rbp)$	$8(n-7)+0(\%r14)$

Table 5. Table listing the (integer) parameter passing according to the AMD64 ABI [3] and the SCADS ABI with frame pointer in use.

do not coincide with the offsets to the stack pointer. Hence, in general SCADS is incompatible to legacy functions with more than six parameters (although there may be some cases when the behavior does not change and the parameters are even correctly referenced).



(a) Call hierarchy without alternating function types (b) Call hierarchy with an alternating structure

Figure 18. call hierarchy with mixed function types.

Even if we would decide to replace the data stack's frame pointer `%r14` with `%rbp`, so that the legacy function uses the frame pointer for parameter referencing in a SCADS environment, this would not fix the incompatibility. The data stack misses some data, like the RIP and the saved frame pointer, which affects the offsets to the parameter section on the stack (see table 5). Hence, even if a legacy function uses the data stack's frame pointer to access its parameters, the offsets would differ from the AMD64 ABI, leaving SCADS still incompatible to legacy functions with parameters stored on the stack.

7.2. Incompatibility of Call-Back Functions

During our analysis, we stumbled across one more incompatibility which differs from the already described scenario. In general, when a legacy function with less than seven parameters is called, the program behaves correctly, as long as the called function does not leave the legacy context. That is supposed to mean that a legacy function must not call a SCADS-compiled function after being called by a SCADS-compiled function itself. Figure 18 describes the scenario that results in an undefined behavior of the program. As long as a called legacy function does not leave its legacy context, meaning it does call only further legacy functions and nothing else, the behavior of the program is correct. But when at any point a legacy function makes a call to a SCADS function, the problem occurs. In general, this type of problem can only occur when one of the legacy functions that are called receives a pointer to a function to call it later on. This scenario is defined as a *call-back*. So to analyze the problem, we must take a look at legacy functions that take a function as parameter for a call-back, and in our case, this function needs to be compiled by SCADS.

The incompatibility once more arises from the fact, that SCADS functions and legacy functions use different registers for management of the stack pointer (we only refer to the data stack here when referring to SCADS). SCADS uses the `%rbp` register for maintenance of the data stack's stack pointer while any other common legacy compilation uses `%rsp` as stack pointer, and `%rbp` as frame pointer if present. When a legacy function is entered that makes use of a frame pointer, it modifies the `%rbp` register, thus alters the state of the data stack's stack pointer. To correctly return to the SCADS context, it is therefore necessary that the `%rbp` register gets restored before the context switch is made, either by a `return` or a `call` instruction. If the `%rbp` register is not correctly restored before returning to or calling a SCADS function, the state of the data stack, defined by its stack pointer is undefined, and therefore results in an undefined behavior of the program as soon as the function starts its execution. That is exactly the problem that occurs when a program inhabits an alternating call hierarchy, that mixes the context switches as shown in figure 18(b), as it is the case in a call-back hierarchy. If a legacy function, called by a SCADS function, does never call another SCADS function at all, the context of SCADS is correctly restored (or maybe not even touched if no frame pointer was used). But in case it calls a SCADS function before it returns, it does not fulfill an automatic context recovery, which leaves %rbp in an invalid state as already explained.

Listings 20 and 21 demonstrate visually where the difference lies between a call-back hierarchy that does not restore the context correctly, and a normal call hierarchy without alternating calls, which does restore the context as supposed to. The sample code from these listings represents an artificially inlined calling chain, to illustrate in which sequence the machine instructions are executed. The control flow represented by the machine instructions must be interpreted sequentially, although it shall convey the impression of a function call hierarchy. The framed machine instruction represents the context restoration in both examples. As can be easily seen, in listing 20 the context restoration by Legacy-function_1 is made before any other SCADS function is called. Therefore the behavior of the whole code snippet should be correct. Unlike in the example of listing 20, the code from listing 21 does restore the context of the superior SCADS-function_1 way too late. The machine instruction highlighted in red, marks the point where the undefined behavior occurs. At this point the function SCADS-function_2 saves its frame pointer by using the %rbp register, which still contains the value of the stack pointer of the legacy function, which is in our case an address that points somewhere into the control stack. Any usage of the data stack's frame pointer then accesses the control stack at an undefined location. Because of that, the call-back function SCADS-function_2 likely causes an abortion of the program, or in any case, the behavior of the program is undefined and erroneous.

Listing 20: Exemplary assembly code that correctly restores the context of the function SCADS-function.

```
### SCADS-function ###
push %r14
mov %rbp, %r14
...
### Legacy-function_1 ###
push %rbp
mov %rsp, %rbp
...
### Legacy-function_2 ###
push %rbp
mov %rsp, %rbp
...
pop %rbp
ret
### Legacy-function_1 ###
pop %rbp
ret
```

It may come to mind, that the usage of %rbp as stack pointer of the data stack is actually the problem which causes the incompatibility. But actually, it is not, because every register which may potentially be modified within a legacy function would produce the same problem. Nevertheless, we have developed a method to fix this incompatibility. The origin of the problem, as we already found out, lies within the fact, that the stack pointer of the data stack does not get restored in time, when a call-back hierarchy is present, as shown by listing 21 exemplary. The solution, is to integrate a recovery mechanism for the %rbp register that fulfills the correct context restoration, even in a call-back scenario, and does not affect the behavior of a program incorrectly in any case.

Listing 21: Exemplary assembly code inhabiting an incompatibility by the call to SCADS-function_2.

```
### SCADS-function_1 ###
push %r14
mov %rbp, %r14
...
### Legacy-function ###
push %rbp
mov %rsp, %rbp
...
### SCADS-function_2 ###
push %r14
mov %rbp, %r14
...
pop %r14
ret
### Legacy-function ###
pop %rbp
ret
```

For this purpose we have implemented a *legacy compatibility mode* into the LLVM back-end, which stores the value of %rbp to a global memory location, right before every function call that is made. And for completeness, at the beginning of each function prolog, the value of %rbp is read from this global location and restored before any other action is performed within the function's body.

Listing 22 shows the fixed version of listing 21 when the compatibility mode is enabled. Both of the newly added machine instructions are highlighted with a fringe. The solution does not need to change any code of the legacy function. The SCADS functions are now able to maintain their context recovery by themselves as the global memory slot decouples the task of context recovery from the legacy function. When the function SCADS-function_2 is called, it operates on the data stack as requested, thus leaving the program in a correct state. It is necessary to mention that SCADS functions calling each other directly, are now accompanied by two redundant machine instructions, but the behavior of the functions remains unaltered.

The implementation of the compatibility mode requires a global symbol to be visible to every module to be compiled. This symbol requires 8 byte of memory for a 64-bit system, and therefore does not produce mentionable memory overhead. The performance overhead on the other side, may be noticeable as it is needed to integrate the recovery mechanism into every

Listing 22: Exemplary assembly code with *legacy compatibility mode* enabled (solution to the problem from listing 21).

```
### SCADS-function_1 ###
push %r14
mov %rbp, %r14
...
mov %rbp, GLOB_MEMORY_SYMBOL
### Legacy-function ###
push %rbp
mov %rsp, %rbp
...
### SCADS-function_2 ###
mov GLOB_MEMORY_SYMBOL, %rbp
push %r14
mov %rbp, %r14
...
pop %r14
ret
### Legacy-function ###
pop %rbp
ret
### SCADS-function_1 ###
...
```

SCADS function as long as no control-flow analysis is made to incorporate more information into the compilation process to reduce the number of additional machine instructions in case. We integrated the call-back compatibility mode in such a way that it can be enabled by passing the flag `-enable-legacy-callback-compat` to the compiler (to be more precise, to the LLVM back-end integrated into the *llc* binary).

7.3. Incompatibility of Stack Alignments

While the data stack is automatically aligned by the compiler as necessary, we have implemented the control stack in such a way, that it does not perform any additional alignment. On a 64-bit system, the control stack only stores 8 byte values which automatically enforces 8 byte alignment on the control stack. This way, no space on the control stack is wasted, and the memory segment is filled without any gaps. If a coarser grained alignment is needed, it may be necessary to force-align the control stack by adding gaps of unused memory to every created stack frame. In case a frame pointer is in use, the control stack stores exactly 16 bytes in the corresponding stack frame, which cancels the necessity of adding memory gaps to fulfill 16 byte alignment for example. On the other hand, if no frame pointer is used, then a stack frame of the control stack is 8 byte large, due to the return instruction pointer being the only data structure that is stored in the frame. So in this case, a coarser grained alignment than 8 byte can only be fulfilled by the insertion of unused memory gaps.

The execution of legacy code may require 16 byte alignment for specific tasks. As any legacy code uses `%rsp` as stack pointer, it performs all of its local memory operations on the control stack instead of the data stack, which is referenced by `%rbp`. This is the origin of the incompatibility, because the data stack is 16 byte aligned anyway, but the control stack may only be 8 byte aligned, which can result in problems. Some data types handled by specific machine instructions, such as *SIMD* operations to handle large floating point numbers for example, need 16 byte aligned addresses as operands, otherwise they cause an exception to abort the program.

To allow a correct functionality of legacy code, which makes use of machine instructions that require 16 byte alignment on the stack, we have implemented another compatibility mode, that can be enabled by the user if needed. To activate the feature, the user must pass a flag to the LLVM back-end, called `-enable-legacy-stack-alignment`, when invoking the compilation process. The mode ensures 16 byte alignment on the control stack, by adding an 8 byte memory gap to each stack frame that does not make use of the frame pointer. This way every stack frame of the control stack starts on a 16 byte aligned address, which allows any legacy code to place the data of its call frames, so that it can be correctly processed by any machine instructions that require the now enforced alignment.

The implementation of the enforced alignment requires to generate two additional instructions per function to be compiled, which does not use a frame pointer for the data stack. To be more precise, it is necessary to extend the control stack in each function prolog, using the `sub` instruction, and shrink it once again in each function epilog, by making use of the `add` instruction. As the control stack is only extended and shrunk by the use of `push` and `pop` in general, it is not possible to modify already present `sub` and `add` instructions to fulfill the task of re-alignment as we described it. Therefore the additional instructions may cause a performance overhead, that depends on the percentage of stack frames, which do not use a frame pointer.

7.4. Using SCADS before Initialization

The initialization phase described in section 2.1 is responsible for the allocation of the data stack and eventually sets up the runtime environment of SCADS, to allow the usage of the two stacks as soon as the `main` function starts to run. As we already told, the initialization mechanism is implemented in a function which is wrapped around the `main`. Therefore we can rather logically conclude that it is not possible to use the data stack in any code sections that are executed before the allocation of the data stack's memory segment is done. We can extend this observation by stating that, before the `main` function, only legacy code may be executed which makes use of the original unified stack solely. This also includes the initialization phase itself which is compiled with the GCC compiler, thus its machine code does not make use of two stacks. If this condition is not met, and a SCADS-enabled function is executed before the `main`, the program is most likely to abort with a segmentation fault, because the function would then use `%rbp` as the data stack's stack pointer, which does either not contain any value so far, or points somewhere on the unified stack so that the function overwrites local data unexpectedly.

In general, the explained difficulty is not necessarily a problem, when a legacy LibC is in use. But as soon as one intends to use a SCADS-compiled version of the standard C library, a program is not able to run anymore, if the compilation process is not changed in a specific way. Before a normal program's `main` is invoked, a lot of preparatory work is performed. The linker ensures that many modules/functions like `_start` and `_init`, for example, normally invisible to the developer of a program, are executed before the actual program is run, to fulfill some necessary preparation tasks, like the allocation of memory segments and the setup of environment variables. To perform these tasks, these pre-main functions do make use of various library functions from the standard C library, which is also the reason for the difficulty that arises when a SCADS-version of the standard C library is used instead. All these pre-main functions are run before the `main`, but moreover also before the initialization phase which is the direct predecessor of the `main`. Therefore the usage of library functions results in undefined behavior, because the data stack is yet not allocated but already used when a SCADS-enabled function of the LibC is called before the initialization took place.

One first important measure is to eliminate any usage of library calls within the initialization module itself. For that reason, we have implemented the initialization module to operate without using any library calls, but instead make use of bare system calls, as necessary a couple of times. But the modification of the initialization module solely is not sufficient to let a program successfully use a SCADS version of the standard LibC. We were yet not able to tweak the linker configuration in such a way that it uses two versions of the LibC, one (the legacy library) for the code that is executed before the `main`, and the other one (the SCADS version) for the rest. Therefore, the remaining option to fix the problem was also to eliminate calls to LibC-functions from the pre-main functions, such as `_start`.

As the `_init` function makes use of many chained library functions, it would be a complex task, to set up an overlay framework, which overwrites all the used functionality and eventually gets compiled by the GCC to be linked to the program. Instead we have set up an overlay framework which overwrites the root-functions `atexit`, `_init` and `_ini_tls` of the program's initialization mechanism. We did overwrite the functions with no functionality at all, just to eliminate the calls to library functions. Although the action of these functions do contribute to some essential mechanisms of a program's runtime environment, we have implemented the overlay framework as described, to prove the basic and successful usage of a SCADS-enabled LibC in practice. For a stable compilation and official porting of the standard C library to a SCADS version, a more sophisticated overlay or a modification of the linker is necessary.

Nevertheless, we have used the presented overlay to compile the standard C library of FreeBSD with our SCADS-enabled compiler and were able to test its correct functionality for various cases. The functions of the overlay framework do contain the prefix `__wrap_` within their name, because we used the linker's support for overwriting function calls, by passing the flags `-Xlinker -wrap=FUNCTION_NAME`. Finally, it is worth mentioning that the described problem is only existent when it comes to the porting of the standard C library. Apart from the LibC, normally no library functions do get involved in the initialization of a general program's environment and therefore are not used before the `main` function. Hence, most libraries apart from the LibC can be ported and used as a SCADS-version straightforward, without introducing similar problems.

8. Conclusion

SCADS is a security measure whose task is to hinder attackers from exploiting programs, or to be more accurate, to prevent attackers from redirecting a program's control flow unexpectedly by modifying a return instruction pointer. Although we have implemented SCADS as a prototype for the x86-architecture within the C-compiler *Clang/LLVM*, it is actually a generic concept. In general, SCADS can be useful with every programming language that does not perform boundary checks on buffers by specification, thus making it easy to gain write access on the stack with the help of a buffer overflow. The bisection of Clang and LLVM into a front-end- and a back-end-compiler allow for a modular support of arbitrary programming languages. SCADS has been fully implemented in the LLVM back-end, so any programming language which has its own Clang front-end can be used to compile SCADS binaries with it.

According to the security analysis from section 4, SCADS' focus for protection lies on implicit control flow data of each frame, including return addresses as well saved frame pointers. In fact, the usage of a RIP is almost independent of a program's source code. The only option that can be set by the C programmer to affect the organization of a function's stack frame is the *inline* keyword. If a function is forced to be inlined, its stack frame is enrolled into the program's text segment at places where a call to it are made. This, however, just leads to the elimination of a function's RIP because no jump is followed to be returned on afterwards. The security analysis we have performed in section 4 has proven, that SCADS is well capable of defending the RIP of non-inlined functions against a couple of exploits we have tested.

The performance analysis from section 5 has shown that the performance overhead, which SCADS produces, was only small in our test case environment. SCADS performance leakage is mainly static, due to the initialization phase that must be performed on a program's invocation. Furthermore, the memory efficiency analysis from section 6 illustrats that SCADS compensates its initial memory overhead at runtime. Initially, SCADS requests approximately one additional page size of memory for the allocation of the data stack. Nevertheless, this is just the initial overhead and it is rather small compared to some related work, shown in section 3. Hence SCADS can also be used in an environment which does not supply plentiful memory for each program.

Finally, the compatibility analysis from section 7 states that SCADS is at least partially compatible to legacy code. Its greatest problem, however, are functions with more than six parameters due to the AMD64 calling convention. Most of the functions of the standard LibC do not have more than six parameters, and for the remaining one we have ported the standard LibC of FreeBSD to SCADS. For any other library that has not been compiled with SCADS yet, further incompatibilities may occur, making it also necessary to port the library to a SCADS version in future.

References

[1] Cwe-124: Buffer underwrite ('buffer underflow'). http://cwe.mitre.org/data/definitions/124.html.

[2] StackShield: A Stack Smashing Technique Protection Tool for Linux, January 2000.

[3] System V Application Binary Interface - AMD64 Architecture Processor Supplement. http://www.x86-64.org/documentation/abi.pdf, Oct 2013.

[4] H. Peter Anvin. klibc. http://www.ohloh.net/p/klibc.

[5] ISO/IEC Committee. International standard iso/iec 9899:201x, programming languages - c. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf, Apr 2011.

[6] Intel Corporation. Intel 64 and ia-32 architecture optimization reference manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[7] Intel Corporation. Intel architecture - instruction set extensions programming reference. http://download-software.intel.com/sites/default/files/319433-015.pdf, July 2013.

[8] Oracle Corporation. Oracle solaris 11 information library - linker and libraries guide - global offset table (processor-specific). http://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-74186.html.

[9] Standard Performance Evaluation Corporation. Spec cpu2006. http://www.spec.org/cpu2006/index.html.

[10] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 81, pages 346–355, 1998.

[11] David A. Kaplan, Advanced Micro Devices Inc. (AMD). Hardware Based Return Pointer Encryption. Patent US20140173293 A1, June 2014.

[12] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in c++. In *ACM Sigplan Notices*, volume 31, pages 306–323. ACM, 1996.

[13] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending Embedded Systems Against Control Flow Attacks. In *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, SecuCode'09, pages 19–26, New York, NY, USA, 2009. ACM.

[14] Jian-Jing Fu and Ji-Lin Wang. Software engineering, 2009. wcse '09. wri world congress. In *SCISM: A Solution for General Buffer Overflow Protection*, May 2009.

[15] GNU. Processor pipeline description. http://gcc.gnu.org/onlinedocs/gccint/Processor-pipeline-description.html.

[16] Brendan P. Kehoe. Zen and the art of the internet - a beginner's guide to the internet. (1), Jan 1992.

[17] Bulba Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, May 2000.

[18] Microsoft. Data execution prevention. http://technet.microsoft.com/en-us/library/cc738483%28v=ws.10%29.aspx.

[19] Tilo Müller and Lexi Piminedis. ASLR Smack & Laugh Reference. In *Seminar on Advanced Exploitation Techniques*. RWTH Aachen University, Germany, 2008.

[20] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 1996.

[21] Baiju Patel. Intel Memory Protection Extensions (MPX) Design Considerations. http://software.intel.com/en-us/blogs/2013/07/23/intel-memory-protection-extensions-intel-mpx-design-considerations, July 2013. Intel.

[22] Gerardo Richarte. Four Different Tricks to Bypass StackShield and StackGuard Protection. Technical report, Core Security Technologies, April 2002.

[23] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2:1–2:34, March 2012.

[24] Team Teso Scut. Exploiting Format String Vulnerabilities. http://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf, September 2001.

[25] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls on the x86. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–61, Alexandria, VA, US, October 2007. University of California, San Diego, ACM Press.

[26] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[27] Saravanan Sinnadurai, Qin Zhao, and Weng-Fai Wong. Transparent Runtime Shadow Stack: Protection against Malicious Return Address Modifications. 2008.

[28] TIOBE Software. TIOBE Programming Community Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, December 2014.

[29] Yves Younan. *Effcient countermeasures for software vulnerabilities due to memory management errors*. PhD thesis, Katholieke Universiteit Leuven, May 2008.