SCADS

Separated Control- and Data-Stacks

Christopher Kugler and Tilo Müller

Department of Computer Science
Friedrich-Alexander University of Erlangen-Nuremberg

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 paper, we present a compiler-level protection called *SCADS: Separated Control- and Data-Stacks*. In our approach, we protect return addresses and saved frame pointers on a separate stack, called the *Control-Stack (CS)*. 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 (DS)*, we protect sensitive pointers of a program's control flow from being overwritten by buffer overflows. As we make control flow information simply unreachable for buffer overflows, many exploits are stopped at an early stage of progression with only little performance overhead. To substantiate the practicability of our approach, we provide SCADS as an open source patch for the LLVM compiler infrastructure for AMD64 hosts.

Key words: Stack-based Buffer Overflows, LLVM, Separate Control Stack

1 Introduction

As of 2013, C is still the most frequently used programming language (17.89 %) closely followed by Java [1]. Unlike Java and many other high-level languages, C does not check the boundaries of a buffer at runtime or compile time, leading to the threat of so-called buffer overflow vulnerabilities. With respect to stack-based buffer overflows, the root of exploits is often the stack design that stores *control information as well as data* alternating on the same stack. For that reason, buffer overflows can be exploited by specially crafted inputs that manipulate the return address of a subroutine call to affect the program flow in a controlled manner. This manipulation can be achieved by either redirecting the return address to previously injected shellcode [2], or by reshaping existing code of the target process into a new program logic [3]. According to the National Vulnerability Database, the number of software flaws classified as buffer overflows is still growing. In total, 729 CVEs for buffer errors were reported in 2013 [4]. Furthermore, buffer errors are still the most common threat today, namely 14.60 % of all reported software vulnerabilities were buffer overflows in 2013.

1.1 Related Work: State-of-the-Art Buffer Overflow Protection

During the last two decades, many protection mechanisms were developed that limit the consequences of maliciously abused buffer overflows. In 1998, Cowan et al. proposed a compiler-level extension called *StackGuard* [5] that guards return addresses on the stack by so-called *canaries*. A canary is a random value on the call stack that is placed between a return address and a buffer. Before the control flow jumps back to a return address, the integrity of the canary is checked to test whether a write operation has accessed memory beyond the buffer boundaries. A drawback of this approach is the additional instruction sequence, which is executed with each return from a subroutine call, inducing notable performance overhead. However, StackGuard is frequently in use today and available as a compiler extension for GCC, LLVM and Visual Studio.

In 2000, the GCC patch *StackShield* [6] was published, introducing another compiler-level extension. A so-called *shadow stack* holds a second copy of each return address and writes this copy back to the runtime stack whenever a subroutine call ends. In 2008, this idea was revisited for the binary rewriting tool *TRUSS (Transparent Runtime Shadow Stack)* [7], with the difference that a return address is compared to its shadow copy rather than enforcing its integrity by restoring a backup value. Consequently, the TRUSS approach is more inefficient than the StackShield approach because a comparison operation for each subroutine call is slower than a single copy operation. Note that none of these solutions, including StackGuard, StackShield, and TRUSS, is entirely secure, as shown in the literature [8, 9, 10].

In 2001, the first widely available version of *Address Space Layout Randomization (ASLR)* was published as part of PaX, a Linux kernel patch for security enhancements. ASLR randomizes the virtual address space of a process, possibly including the stack, the heap, the data and also the code section (depending on the OS version and compiler options). Simple buffer overflow exploits are thwarted by ASLR since correct branch addresses to injected shellcode become harder to predict for an attacker. In difference to the afore mentioned solutions, ASLR modifies the environment of a binary and not the binary itself. However, ASLR alone is not secure against many other exploitation techniques, as summarized in the literature [10, 11, 12, 13, 14].

In 2003, the *NX-bit (No eXecute)* was introduced as part of the AMD64 architecture and is now available on all modern x86 CPUs. NX is a hardware extension that prevents the execution of injected shellcode by marking data pages, e.g., stack pages, as non-executable within the page table. The invention of NX considerably complicates the creation of buffer overflow exploits as it becomes impossible to run shellcode on the stack, or any other data page marked as non-executable. However, more advanced exploitation strategies, known as *return-into-libc* [3] and *Return-Oriented Programming (ROP)* [15], bypass the protection of NX by running existing code gadgets from executable pages in a newly composed order. These techniques often succeed in the presence of both NX and ASLR. Recent research papers generalized ROP to a wider class of instruction sets [16, 17] and to a smaller base of necessary gadgets [18].

1.2 Contribution: Separated Control- and Data-Stacks

As outlined in the last section, the race between countermeasures and exploitations in the field of buffer overflows is still ongoing today. With several high-quality publications about Return-Oriented Programming in the last few years [3, 15, 16, 17, 18, 19], the attacking side seems to be presently at an advantage over the defending side. Looking at the way many ROP exploits work today, they are successful because return addresses can oftentimes still be overwritten. None of the countermeasures mentioned above prevent return addresses from being overwritten.

To guarantee the integrity of return addresses in a secure and highly efficient manner, we propose the compiler-level extension *SCADS: Separated Control- and Data-Stacks*. With SCADS, we propose a protection mechanism that prevents return addresses from being overwritten by writing beyond the boundaries of a buffer. We remove return addresses from the *Data-Stack (DS)* and place them on a separate *Control-Stack (CS)*. Unlike StackShield and TRUSS, we do not introduce a shadow stack holding a copy of control information, but use a separated stack for control information. As a consequence, SCADS does not have to deal with comparison operations or backup recoveries, but natively works on two stacks with mutually exclusive content types.

Using a single stack for data and control information is a de-facto standard for computer programs of the last decades. But this design is neither required by the OS nor by the x86 architecture. Therefore, similar to StackGuard and StackShield, SCADS can be implemented as a compiler-level extension without support from the OS or hardware. In contrast to previous compiler extensions, SCADS does not involve an extra sequence of instructions at the end of a subroutine call, thus minimizing runtime overhead.

Specifically, our contributions are as follows:

1. *Design Concepts of SCADS* (Sect. 2): We propose concrete design choices for the implementation of a separated CS and DS on AMD64 systems. For example, we explain the relative position of the two stacks within the virtual address space, discuss whether the stacks grow up- or downward, and reason which registers are used as a stack or base pointer.

2. *Implementation of SCADS* (Sect. 3): Based on our design concepts, we present an open source patch for the LLVM compiler infrastructure, which we make available under an NCSA Open Source License. We first developed this patch on Linux and later shifted towards FreeBSD, because the FreeBSD project announced to move from GCC to LLVM as its default compiler.

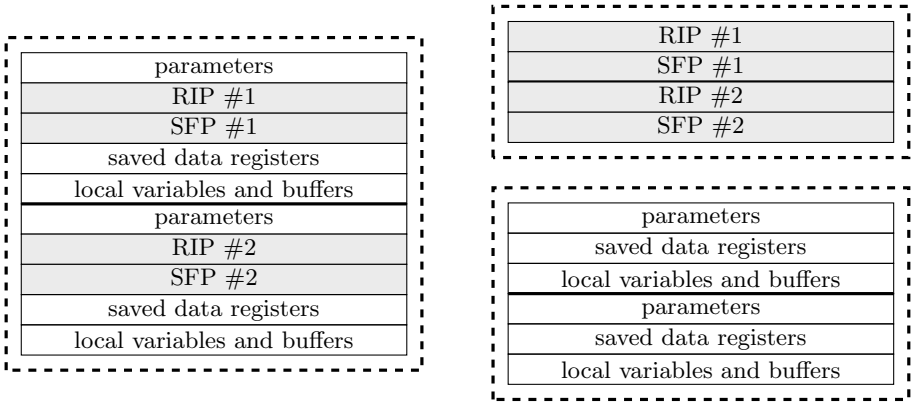
3. *Evaluation of SCADS* (Sect. 4): Based on the practical implementation of SCADS, we present an evaluation of our approach regarding its security, performance and compatibility. In comparison to other compiler-level extensions, especially StackGuard, we present improved performance results. However, we also point to some compatibility issues of SCADS running on current FreeBSD and Linux hosts.

2 Design Concepts of SCADS

One of our design goals for SCADS was the creation of a protection mechanism that modifies the compiling and linking process of a software package without the need to change underlying operating system or hardware level components. Consequently, allocating and handling the second stack must be designed in a way that is compatible with current system environments. From the beginning of the design phase, we focused on the AMD64 architecture running UNIX systems like Linux and FreeBSD as a target platform.

2.1 Separating Control Flow Information from Data

As illustrated in Fig. 1, SCADS is based on the idea of separating information placed on a single call stack into two entities: *control information* and *data*. Control information is placed on the *Control-Stack (CS)*, while data is placed on the *Data-Stack (DS)*. We classify return addresses and frame pointers as control information and everything else as data, especially parameters, local variables, and buffers of any data type. Due to this separation, buffer errors cannot be directly exploited to overwrite return addresses, and so it becomes hard for an attacker to redirect a program's control flow via the manipulation of data entries.



(a) Single call stack. (b) Separated CS and DS.

Fig. 1. Frames of a single call stack in comparison to separated frames in SCADS. The CS stores *Return Instruction Pointers (RIPs)* and *Saved Frame Pointers (SFPs)*.

Note that, although many buffer overflow vulnerabilities are thwarted with SCADS, it is impossible to protect the control flow of all imaginable C programs. The principle of separating control flow information from data is stretched to its limits when it comes to *function pointers*. Should we classify function pointers as control information or as regular data? How do we handle buffers of function pointers, or even more complex data structures that involve function pointers?

The laconic answer is that we *classify function pointers as data*. The reason is that C is not a type safe language when a programmer uses explicit casts. Untyped function pointers can be casted to and from any data type, or even be computed at runtime, such that it is impossible to reliably cover all function pointers at compile time.

Hence, as it is impossible to cover additional control elements that are *explicitly introduced by the programmer*, we decided to position SCADS as a protection mechanism for control elements that are *implicitly introduced by the compiler*, i.e., return addresses and frame pointers. The use of explicit function pointers in C is rare, at least compared to the number of implicit return addresses, and so we leave it to the programmer to protect information that is deliberately introduced. Due to this "imperfection", we designed SCADS in a way that is compatible with established protections like ASLR and NX. In this sense, SCADS is not a replacement for ASLR or NX, but an additional protection mechanism to thwart the root of many of today's ROP exploits.

2.2 Stack Alignments in the Virtual Address Space

Inside the virtual address space of an AMD64 process, there are approximately 128 terabytes of free space between the call stack and the heap of a process. We make use of this area to place the second stack, which arises from splitting the common call stack into a CS and a DS, as illustrated in Fig. 2.

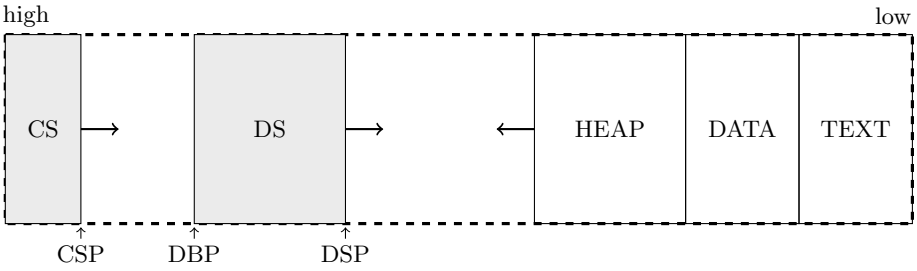


Fig. 2. Virtual address space layout of a user mode process compiled with SCADS.

The position of the CS corresponds to the old unified stack, whereas the DS is moved to a new position in the area between the CS and the heap. Besides its position, the CS closely corresponds to the old stack because x86 instructions like `CALL` and `RET` implicitly operate on the stack which is referred by the `RSP` register. For the DS, on the other hand, we can use arbitrary CPU registers as stack and base pointers, as we explain in Sect. 2.4.

From this perspective, the CS is the old stack while the DS is a new stack, and all data (apart from return addresses and frame pointers) are moved from the CS to the newly created DS. In practice, the CS, which stores only two elements per subroutine call, is smaller than the DS, which stores all remaining elements.

An exception constitute highly recursive subroutine calls that create only little or no local variables on the DS.

Collisions and interferences of the DS with the CS and the heap are excluded for several reasons. First, the stack size in modern OS is limited to several megabytes. For example, the default stack limit in Linux is eight megabytes, and this limit is applied to both the CS and the DS. Second, the virtual address space between the CS and the heap spans about 128 terabytes in 64-bit processes. And third, the empty address space between the stacks and the heap is not mapped into a running process. If an attacker tries to overwrite buffers in the DS with gigabytes of data, in order to hit control flow information in the CS, the process crashes with an access violation error for unmapped memory pages.

To support the compatibility of SCADS with the principles of ASLR, we let the OS choose a random base address for the CS and additionally compute a random offset between the CS and the DS at load time. As a consequence, the relative position of return addresses is not predictable for an attacker, such that exploits are thwarted that write to arbitrary memory locations, e.g., due to an uncontrolled format string [20]. We have chosen to randomly select the least-significant 24 bits of the DS base address, as we explain in Sect. 3.2 in detail.

2.3 Stack Growth Direction

Talking about stacks, a typical design question is the direction to which a stack grows, i.e., upwards or downwards. Traditionally, stacks grow downward in the x86 architecture because back in the days of 16-bit CPUs, the address space was very limited and having the heap growing upward, while the stack grows downward, gave the programmer most flexibility. If both the stack and the heap were growing in the same direction, generally less memory could be used before a collision. However, on modern 64-bit systems, collisions are not an issue anymore and therefore, we revisit the decision to let stacks grow downward.

As illustrated in Fig. 2, we have chosen to maintain the growth direction for both stacks to be *downwards*, basically due to engineering constraints from the hardware and the OS. Since CPU instructions like `CALL` and `PUSH` reduce the `RSP` register, and instructions like `RET` and `POP` increase the `RSP` register, it seems reasonable to let the CS grow downward to benefit from these x86 instructions.

For the DS, on the other hand, we originally considered to reverse the growth direction to be *upwards*. This might involve a minor improvement against exploits, because if a stack is growing down, a buffer error can overwrite all older stack elements, whereas if a stack is growing up, a buffer error can only overwrite new stack elements. Assuming that generally more older stack elements exist around a vulnerable buffer than new ones, reversing the growth direction of the DS limits the damage caused by a buffer error. However, this is not an effective protection mechanism on its own, and we eventually chose the DS to be growing downward due to OS constraints. In Linux, the `mmap` syscall offers an option, called `MAP_GROWSDOWN`, to allocate downward growing stacks, but none to allocate upward growing stacks. As patching the OS kernel was not an option

for our design, and since handling an automatically growing stack inside the user mode induces notable performance overhead, we eventually chose the DS to grow downward.

2.4 Stack- and Base-Pointer Registers

Each stack must be managed by a separate stack pointer, such that two CPU registers are occupied as stack pointers in SCADS. We refer to these pointers as CSP for the CS and DSP for the DS, as illustrated in Fig. 2. By design, the AMD64 architecture provides only a single stack pointer, namely `RSP`, and in addition to it a base pointer, namely `RBP`. As stated above, we are forced to assign `RSP` to CSP because instructions like `CALL` and `RET` implicitly operate on the stack which is referred by `RSP`. However, there is no need to keep track of a base pointer for the CS because the frame size of the CS is always constant. Exactly two pointers, namely the RIP and the SFP, are stored per CS frame as illustrated in Fig. 1. Hence, the `RBP` register becomes expendable for the CS, and since x86 instructions like `CALL` and `RET` never change `RBP` implicitly, we can freely re-assign it for other purposes. We decided to assign `RBP` to DSP, meaning that `RBP` does not serve as a base pointer anymore but as a stack pointer for the DS.

If *Frame Pointer Omission (FPO)* is set, which is a default compiler optimization by LLVM to omit the need for base pointers, DBP is not required. In that case, using `RSP` as CSP and `RBP` as DSP is sufficient for the design of SCADS and no extra registers must be occupied. However, if FPO is not set, or cannot be used, we assign `R15` to DBP, i.e., we misuse the last general purpose register of AMD64 as base pointer for the DS. Due to the high number of available GPRs in AMD64, the occupation of `R15` does not really affect the runtime performance of a compiled program. To the contrary, today's compilers like GCC and LLVM leave registers like `R14` and `R15` largely unused in order to maintain a common code base with IA-32.

However, there is another problem with our design: We cannot use the x86 instructions `PUSH` and `POP` to store regular data on the DS, like parameters for function calls, since `PUSH` and `POP` implicitly refer to the `RSP`, which points to the CS. As we want to store parameters on the DS, and not on the CS, `PUSH` must be transformed into a `SUB` / `MOV` sequence as illustrated in Listing 2. Likewise, `POP` must be transformed into a `MOV` / `ADD` sequence, as also illustrated in Listing 2.

Note that the performance penalty arising from these instruction sequences is minimal, if present at all, because compilers like GCC and LLVM rarely use the `PUSH` instruction today. To the contrary, they deploy a single `SUB` instruction followed by a sequence of `MOV` instructions to store multiple parameters efficiently on the stack. The same holds true for a sequence of `POP` instructions, which is often replaced by a more efficient `ADD` instruction. Only frame pointers are frequently stored and restored with `PUSH` / `POP` during function epilogues and prologues. Frame pointers, however, are stored on the CS and can therefore benefit from `PUSH` and `POP` without restrictions.

Listing 1. Push and pop instructions on the Control-Stack.

```
// push instruction
   push %rbx

// pop instruction
   pop %rbx
```

Listing 2. Push and pop instructions on the Data-Stack.

```
// push simulation
   sub 8, %rbp
   mov %rbx, (%rbp)
// pop simulation
   mov (%rbp), %rbx
   add 8, %rbp
```

3 Implementation of SCADS

We implemented SCADS, based on the design concepts that were outlined in the last section, in practice as a compiler-level patch for the LLVM infrastructure. This patch is publicly available on our webpage, licensed under an NCSA Open Source License which is similar to BSD and MIT licenses (and typically used for code in the LLVM project). In the following, we address selected points of our implementation; please refer to the LLVM patch itself for a comprehensive technical description.

The patch comprises 14 files that were either extended or newly added to the LLVM project. We focused on x86-64 as LLVM back end for the AMD64 architecture and on Clang as an LLVM front end for C. As it turned out, it is possible to implement SCADS solely in the back end of the LLVM infrastructure, and therefore we basically support other front ends, like DragonEgg, too. We were also able to compile other programming languages, like C++ and Objective-C, with SCADS (although this requires further testing).

The majority of patches were applied to files that are specific for the x86-64 architecture. For example, the file *X86FrameLowering.cpp*, which handles the creation and removal of stack frames as it defines function prologues and epilogues, involves a major part of the SCADS implementation. Particularly, the methods `emitPrologue` and `emitEpilogue` are modified within our patch in a way that they redirect control information and data to either the CS or the DS. Although we had to make several changes to core files of the compiler infrastructure, we implemented SCADS in a way that LLVM remains fully backward compatible. To this end, we introduced the following new compiler flags that handle the usage of SCADS in the back end:

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

The flag `-num-stacks` is the flag that essentially turns SCADS on or off by defining the number of stacks that are used in the runtime environment. This number is currently restricted to “1” (SCADS disabled) or “2” (SCADS enabled), but might be extended in future, e.g., to store explicitly defined function pointers,

as discussed in Sect. 2.1, on a third stack. The latter two flags enable compatibility modes that we had to implement to deal with subroutine calls into legacy code.

3.1 The Control-Stack

The CS replaces the plain old call stack of user mode programs and is automatically allocated by the OS at load time. That is, it is not necessary to allocate the CS manually from within SCADS and the CS base address is directly affected by the kernel implementation of ASLR. As aforementioned, by using `RSP` as stack pointer, the maintenance of return addresses is inherently implemented by `CALL` and `RET` without modifications. Additionally, the `PUSH` and `POP` instructions can be used to store frame pointers on the CS, in contrast to data on the DS.

Note that frame pointers are often omitted in LLVM due to the extensive use of FPO as a default compiler optimization. If no frame pointers are saved on the CS, we store only an 8-byte return address per stack frame, unless `-enable-legacy-stack-alignment` is set. If this flag is set, the return address is followed by an 8-byte dummy value, i.e., regardless of whether FPO is enabled or not, we then store 16-byte stack frames on the CS. The reason are compatibility issues with calls into legacy libraries that require stack frames to be aligned to 16-byte boundaries. Additionally, some machine instructions may operate more performant on 16-byte aligned stack frames.

3.2 The Data-Stack

Unlike the CS, the DS must be allocated and handled explicitly by SCADS with further modifications in the build and linking process. To store regular data on the DS at runtime, the DS must first be allocated (before the `main` function is invoked) and then subsequent access to local variables, parameters, and buffers can be redirected to the DS.

In terms of memory efficiency and performance, an important requirement for the DS is to grow automatically just as the CS. To implement an automatically growing stack, kernel support is preferable such that erroneous access to an unmapped page below the stack pointer yields the allocation of that page. On UNIX based operating systems, the system call `mmap` usually provides this possibility: In Linux, the flag `MAP_GROWSDOWN` can be passed on to `mmap` to allocate growing stacks, whereas `MAP_STACK` can be passed on in FreeBSD for that purpose. Note that *stacks are generally not shrinking automatically*, neither with SCADS nor on common computer systems. If a growing stack hits the system-wide stack limit, e.g., due to recursion with large stack-based buffers, the stack stays this size until the process is quit. There is no concept for automatic stack deallocation in modern operating systems.

The code we execute in LLVM to allocate the DS under Linux is illustrated in Listing 3. As a result of this allocation, we receive an anonymous, non-executable memory section between the CS and the heap with the initial size of one page, including read- and write-privileges. The `MAP_GROWSDOWN` flag allows the DS to

grow in size on a per-page basis, managed by the kernel just like the CS. If the reallocation encompasses more than one page at once, e.g., due to buffers greater than 4096 bytes, the reallocation step must be split up into single pages at compiler-level.

Listing 3. Allocation of the DS by means of the system call `mmap` under Linux.

```
int control_stack_address = 0;
void *data_stack_address = (void*)(((long
    long)&control_stack_address) - ((long long)DATA_STACK_OFFSET));
const int MMAP_PROT = PROT_READ | PROT_WRITE;
const int MMAP_FLAGS = MAP_PRIVATE | MAP_ANONYMOUS | MAP_GROWSDOWN;
const int INITIAL_STACK_SIZE = PAGE_SIZE;
const int fd = -1;
const int offset = 0;
void *data_stack_ptr = invoke_mmap_syscall(data_stack_address,
    INITIAL_STACK_SIZE, MMAP_PROT, MMAP_FLAGS, fd, offset);
```

As the DS is bound to a memory section which is not allocated automatically by the kernel at load time, ASLR has no influence on the positioning of the DS base address. As a consequence, although ASLR is enabled, the offset between the CS and the DS would remain constant without further modifications on the compiler or linker level. An attacker could in some scenarios misuse this information to write into the CS, and to eventually modify the control flow. Therefore, we preset the base address of the DS to lie $8 * StackSizeLimit$ below the CS, where $StackSizeLimit$ is usually 8 MB, and then randomize the 24 least-significant bits of the base address, as shown in Equation 1 and 2:

$$Base_{DS,static} = Base_{CS} + 8 * StackSizeLimit \tag{1}$$

$$Base_{DS,final} = Base_{DS,static} + RandomEntropy \in [-2^{23}, 2^{23} - 1] \tag{2}$$

This computation prevents the DS from overlapping with the CS, as well as from lying immediately below the CS, because with a $StackSizeLimit$ of 8 MB, equation (1) leads to a static base address of the DS which is 64 MB below the CS. By randomizing the least-significant 24 bits of that address in equation (2), the DS is relocated to at most 48 MB below the CS. Basically, the randomization can be improved by selecting more than 24 bits randomly in future, but then further load time checks would be required to ensure that the DS does not collide with memory sections allocated by the runtime linker for dynamically linked libraries. By randomizing only the least-significant 24 bits, this circumstance is ruled out on 64-bit systems. Additionally, we have to align the DS to 16-byte boundaries, such that only 20 bits of the DS address space are effectively randomized. This might raise the question whether brute force attacks on the address space layout can successfully be thwarted. For 32-bit processes, however, the kernel implementation of ASLR randomizes exactly 20 bits of stack addresses, as well, and that turned out to be sufficient in many scenarios.

3.3 Build and Linking Process

As outlined in the last section, the DS is not automatically created by the OS at load time but must be allocated by a process on its own at an early stage of its runtime. One possibility to achieve this is to hard-code the initialization phase of SCADS into the LLVM compiler such that the DS allocation is placed into each program before the `main` function is invoked. However, we decided not to integrate the initialization phase of SCADS into the compiler, as it does not fall in the area of responsibility of a compiler but of a linker. Therefore, we encapsulated the initialization phase as a separate object file that is bound to a program at link time.

Listing 4. Command line for the build and linking process of SCADS.

```
clang -emit-llvm -S -o <intermediate_name> <source_name>
llc -march=x86-64 -num-stacks=2 -o <asm_name> <intermediate_name>
clang -Xlinker --wrap=main -o <binary_name> <asm_name> init_module.o
```

In Listing 4, an exemplary command line is shown that builds and links a binary with SCADS. The initialization module `init_module.o` comprises a function called `__wrap_main`. By passing the flags `-Xlinker --wrap=main` to the LLVM linker, every call to the `main` function is replaced with a call to the function `__wrap_main`. To invoke the original entry point, `__wrap_main` calls the `main` function after the initialization phase is finished. For the future, we plan to patch the LLVM linker in a way that it automatically binds the initialization module to binaries, i.e., without the need to manually pass on all configuration parameters of SCADS.

Note that when `__wrap_main` is entered, the runtime environment consists of only a single stack that was allocated by the OS. The command line arguments of a program, i.e., the `argc` and `argv` parameters for `main`, are therefore initially written to the single stack rather than the DS. As a consequence, the `argc` and `argv` parameters must be migrated to the DS during the initialization phase, because we classify all command-line arguments and parameters as regular data. Hence, after the allocation of the DS is finished, the command-line arguments are migrated to the DS and finally the `argc` variable and the `argv` pointer are restored to the registers `EDI` and `RSI` to comply with the System V AMD64 calling convention [21] for `main`.

4 Evaluation of SCADS

We now present an evaluation of SCADS regarding its *security* (Sect. 4.1), its *performance and efficiency* (Sect. 4.2), and finally its *compatibility* (Sect. 4.3).

4.1 Security

Classic binary exploits that write beyond the boundaries of a buffer to manipulate the return address are predestinated to fail with SCADS, because buffers are

located on the DS while return addresses are located on the CS. Consequently, instead of modifying return addresses, buffer errors can only corrupt regular data on the DS until they reach the unallocated area between the DS and CS, which leads a program to terminate.

Although this explains SCADS' immunity against the most simple type of buffer overflow exploits, the task of giving a more substantial line of reasoning for the security of SCADS is difficult. We do not seek to verify the security of SCADS in a formal manner, but focus on known exploitation techniques and compare the security of SCADS with that of other protection mechanisms. Recall that SCADS was not designed as a stand-alone protection mechanism but to collaborate with established OS- and hardware-level protections like ASLR and NX. The motivation to deploy SCADS in addition to ASLR and NX is mainly the known exploitation technique of *Return-Oriented Programming (ROP)* which often defeats ASLR and NX in practice today.

Contrary to ASLR and NX, the StackGuard protection is competing with SCADS, not only because it is a compiler-level extensions, but also because it is largely incompatible with SCADS. While StackGuard and SCADS can be combined with ASLR and NX, a combination of both techniques seems rather pointless. Either a canary is placed in front of a return address (StackGuard), or the return address is moved to a separated stack (SCADS).

To understand the security benefit of SCADS, recall that SCADS is the first protection mechanism that *prevents return addresses from being overwritten*. Previous solutions either complicate the launching of shellcode (ASLR and NX) or verify the integrity of a return address *after* it has been overwritten (StackGuard). As indicated in Sect. 2.1, SCADS protects implicit control flow information, particularly return addresses and frame pointers, but no function pointers that were explicitly introduced by the programmer. Comparing this "weakness" of SCADS with StackGuard, however, StackGuard does not place a canary in front of each function pointer either. Hence, also with StackGuard, only implicit control information is protected.

In our experiments, we were able to produce examples for both scenarios: C programs that can be exploited in spite of StackGuard but not with SCADS, and the other way round. For example, SCADS can be more secure than StackGuard with respect to vulnerabilities that give an attacker *random write access to relative stack addresses*. There are many exploits in combination with such vulnerabilities, which are also entitled as *indirect pointer overwrites* [22]. On the other hand, StackGuard can be more secure than SCADS when explicit function pointers get overwritten which are lying in older stack frames than the vulnerable buffer. With SCADS, the control flow could be redirected to point to another predefined function, whereas with StackGuard, the canary of the current stack frame would be violated, leading to a termination of the program.

The strength of SCADS is that it prevents exploits relying on a chain of ROP gadgets placed at the top of the stack which is referred by `RSP`. With SCADS, it is not easily possible to place a chain of ROP gadgets near to the `RSP`, but only near to the `RBP`, i.e., on the DS. The `RSP` is implicitly used by `RET` instructions

and hence, the position of the RSP is one of the essential parts of ROP exploits. To bypass this obstacle, an attacker would have to redirect the RSP to the DS, or any other data page that holds user input, in a controlled manner. This is, however, assumed to be difficult for practical vulnerabilities.

4.2 Performance and Memory Efficiency

As seen in the last section, the security of SCADS is on par with that of StackGuard. The advantages of SCADS in comparison to StackGuard only turn out when it comes to performance. We can say that the performance overhead of SCADS is mainly static, due to the extended initialization phase, while StackGuard shows a dynamic runtime overhead, due to extended function epilogues. More precisely, StackGuard involves extra operations to verify the integrity of a canary at the end of each subroutine call, while SCADS involves a constant number of additional operations during its initialization phase.

In the following, we present detailed performance results for a recursive implementation of the Fibonacci sequence. The Fibonacci programs, which are similar to the implementations in Listing 5 and 6 in the appendix, were compiled with four different compiler settings: Clang, Clang/SCADS, GCC, and GCC/StackGuard, each with FPO enabled. An analysis of the number of assembler instructions yields that both Clang and GCC generate 24 instructions per recursive subroutine call. Interestingly, the Clang/SCADS configuration generates exactly 24 instructions, too, whereas the GCC/StackGuard configuration generates 30 instructions. In other words, the number of instructions that are executed per subroutine call increases by 20% for Fibonacci when comparing StackGuard and SCADS.

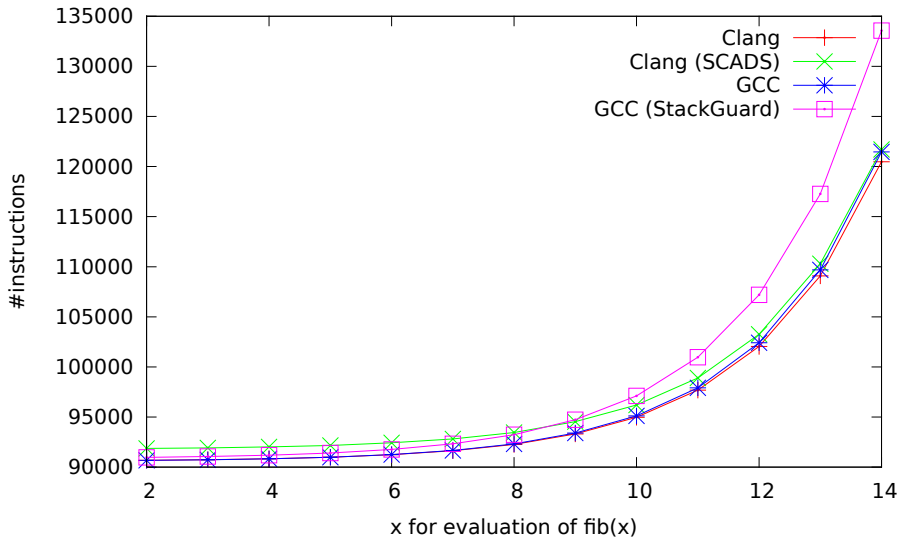


Fig. 3. Number of instructions for the recursive computation of a Fibonacci number.

This effect is illustrated in Fig. 3, showing the overall number of executed instructions per Fibonacci number. As we expected, SCADS initially executes more instructions than the other variants because of the initial allocation of the DS. Later on, approximately at the tenth Fibonacci number, the impact of StackGuard's canary management outruns SCADS in terms of executed machine instructions. From there on, the slope of the StackGuard curve rises significantly faster in comparison to the SCADS curve.

Of course, the number of executed instructions is closely related to the execution time of a program, as shown in Fig. 4. It can also be seen that the StackGuard curve departs significantly from the curves that represent SCADS, GCC and Clang. For the 52nd Fibonacci number, for example, the program compiled with StackGuard is up to 80 seconds slower than the other variants. In contrast to this, the static overhead caused by the initialization phase of SCADS is not visible in Fig. 4 as it lies in the range of microseconds (that cannot even be measured reliably due to noise issues).

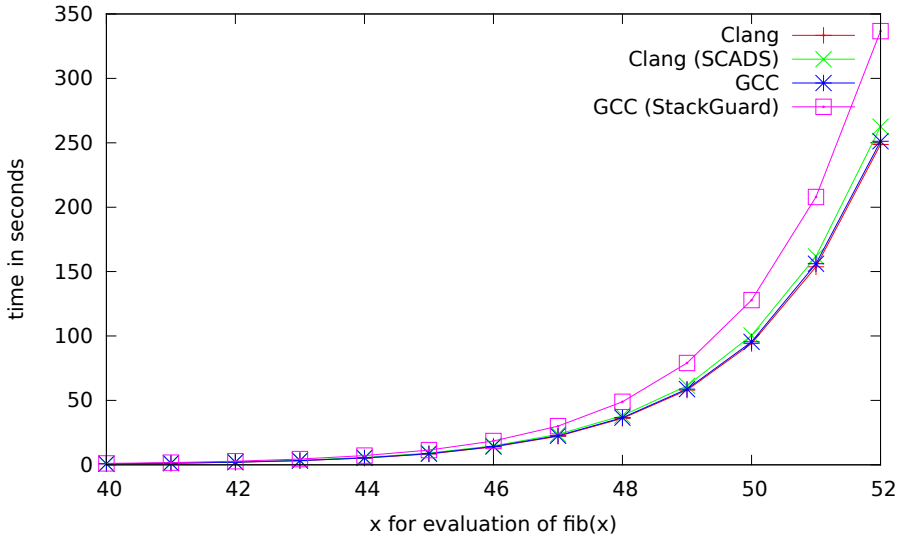


Fig. 4. Execution times of a program that computes the Fibonacci sequence recursively.

With respect to memory efficiency, StackGuard adds an additional canary to each stack frame, whereas SCADS does not add extra values on the CS or DS. However, talking about memory efficiency we must differentiate between *allocated memory* and *actually used memory*. As both the CS and the DS are initially allocated with the size of one page, SCADS "wastes" at most one page compared to StackGuard.

4.3 Compatibility

Another important topic that must be discussed when talking about a new protection measure is its compatibility to legacy code. First of all, software packages must be recompiled to gain security from SCADS, and inline assembly statements must possibly be adapted manually, especially if they make use of the stack or base pointer. However, StackGuard suffers from the exactly same limitations, i.e., StackGuard also requires the recompilation of software packages and is not fully compatible with all inline assembly statements. For both SCADS and StackGuard, functions that were written entirely in assembly language cannot be protected by an automated recompilation.

It is not unlikely that solutions like StackGuard and StackShield were favored over the separation of control flow information from data due to compatibility concerns in the past. As modern software generally has a high amount of dependencies on various libraries, which are possibly closed source, it is a desirable property for a compiler extension to preserve binary compatibility.

The compatibility with existing binary code was one of the most critical challenges during the development phase of SCADS. Basically, we focused on three types of incompatibilities: First, SCADS was incompatible to legacy functions that require a 16-byte alignment of the CS. We solved this issue by introducing the Clang flag `-enable-legacy-stack-alignment` that aligns the CS to a 16-byte boundary. Second, SCADS was incompatible to legacy functions that take a SCADS function as a call-back parameter. We solved this issue by introducing the Clang flag `-enable-legacy-callback-compat` that stores a backup of the `RBP` before SCADS functions are called from legacy code. When a SCADS function returns, the `RBP` is recovered and then the legacy function can proceed.

AMD64	`RDI`	`RSI`	`RDX`	`RCX`	`R8`	`R9`	`RBP+16`	`RBP+24`	..
SCADS	`RDI`	`RSI`	`RDX`	`RCX`	`R8`	`R9`	`R15`	`R15+8`	..

Table 1. The System V AMD64 ABI in comparison to the SCADS calling convention. If FPO is used, parameters are referenced by `RSP` and `RBP` rather than `RBP` and `R15`.

Third, SCADS is not compatible with legacy functions that take more than six parameters because the new memory layout and register occupation of SCADS alters the calling convention. Legacy functions which are consistent with the AMD64 ABI store the first six parameters in registers, and remaining parameters are placed on the stack and referenced by `RBP`, as shown in Tab. 1. But within the runtime environment of SCADS, `RBP` is not a base pointer but a stack pointer to the DS and parameters are referenced by `R15`, as shown in Tab. 1. This causes the offsets of parameters to be different in SCADS and legacy code.

This restriction can be counteracted in several ways. First of all, functions with more than six parameters are rarely used in C and for the remaining functions, wrapper functions can be implemented. To this end, we could loosen the security definition of SCADS and pass on parameters to legacy code via the CS rather than the DS. Another way is to provide as many libraries natively in SCADS

as possible. We successfully compiled the entire BSD LibC with SCADS under FreeBSD and plan to port other libraries soon. Note that this is a practical way under FreeBSD but not under Linux, because FreeBSD is moving from GCC to LLVM as its default compiler, such that FreeBSD packages can easily be recompiled with LLVM/Clang. Under Linux, however, most packages are written in a GCC-specific C dialect that fails to compile with LLVM/Clang.

5 Conclusions and Future Work

Buffer overflows are binary vulnerabilities, which are caused by missing range checks on buffer boundaries, and are still an inherent problem of widely used programming languages like C. In the recent past, exploitation techniques like ROP have impressively shown that OS- and hardware-level protections like ASLR and NX are often insufficient and must be combined with further protections. In this paper, we have presented *SCADS (Separated Control- and Data-Stacks)*, which introduces the separation of regular data from implicit control flow data on two stacks. Return addresses and frame pointers are stored on the *Control-Stack (CS)*, while regular data, including buffers, are stored on the *Data-Stack (DS)*.

In comparison to other compiler-level extensions, especially StackGuard, SCADS shows effectively no runtime overhead but introduces only a short initialization phase. Both SCADS and StackGuard protect return addresses as well as saved frame pointers without support from the OS or hardware, but require C programs to be recompiled to benefit from this protection. The level of security reached by SCADS is on par with that of StackGuard. The most severe limitation of SCADS is currently its compatibility to legacy code libraries. As we changed the AMD64 calling conventions from the seventh parameter onwards, legacy functions with more than six parameters cannot be called. This could be solved either by passing parameters on the CS rather than the DS, or by recompiling an entire UNIX distribution like FreeBSD with SCADS.

Today, SCADS is compatible with the latest x86 architecture, namely AMD64, as well as UNIX based OSes like Linux and FreeBSD. However, support from the OS- and hardware-level could assist the approach of SCADS in future, e.g., by letting the OS loader automatically allocate two stacks per process at load time, possibly growing downward. Furthermore, any store to and retrieval from the DS is currently implemented by `SUB/MOV` and `MOV/ADD`, because using `PUSH/POP` results in an access to the CS. Although this does not involve a performance drawback with today's compilers, future hardware could be extended to support a second stack natively, e.g., by a second `RSP` with dedicated `PUSH/POP` instructions.

Acknowledgement.

This work was supported by the German Research Foundation as part of the Transregional Collaborative Research Centre *Invasive Computing* (SFB/TR 89). A special thanks goes to Daniel Lohmann for interesting discussions on SCADS.

A Appendix

Listing 5. Recursive Fibonacci compiled with plain Clang (FPO disabled).

```
<fib>:
 0:   push   %rbp
 1:   mov    %rsp,%rbp
 4:   sub    $0x20,%rsp
 8:   mov    %edi,-0xc(%rbp)
 b:   cmpl   $0x0,-0xc(%rbp)
 f:   jne    <fib+0x1b>
11:   movq   $0x0,-0x8(%rbp)
19:   jmp    <fib+0x52>
1b:   cmpl   $0x1,-0xc(%rbp)
1f:   jne    <fib+0x2b>
21:   movq   $0x1,-0x8(%rbp)
29:   jmp    <fib+0x52>
2b:   mov    -0xc(%rbp),%eax
2e:   sub    $0x1,%eax
31:   mov    %eax,%edi
33:   callq  <fib>
38:   mov    -0xc(%rbp),%edi
3b:   sub    $0x2,%edi
3e:   mov    %rax,-0x18(%rbp)
42:   callq  <fib>
47:   mov    -0x18(%rbp),%rcx
4b:   add    %rax,%rcx
4e:   mov    %rcx,-0x8(%rbp)
52:   mov    -0x8(%rbp),%rax
56:   add    $0x20,%rsp
5a:   pop    %rbp
5b:   retq
```

Listing 6. Fibonacci implementation compiled with SCADS (FPO disabled).

```
<fib>:
 0:   push   %r15
 2:   mov    %rbp,%r15
 5:   sub    $0x20,%rbp
 9:   mov    %edi,-0x14(%r15)
 d:   cmpl   $0x0,-0x14(%r15)
12:   jne    <fib+0x1e>
14:   movq   $0x0,-0x10(%r15)
1c:   jmp    <fib+0x58>
1e:   cmpl   $0x1,-0x14(%r15)
23:   jne    <fib+0x2f>
25:   movq   $0x1,-0x10(%r15)
2d:   jmp    <fib+0x58>
2f:   mov    -0x14(%r15),%eax
33:   sub    $0x1,%eax
36:   mov    %eax,%edi
38:   callq  <fib>
3d:   mov    -0x14(%r15),%edi
41:   sub    $0x2,%edi
44:   mov    %rax,-0x20(%r15)
48:   callq  <fib>
4d:   mov    -0x20(%r15),%rcx
51:   add    %rax,%rcx
54:   mov    %rcx,-0x10(%r15)
58:   mov    -0x10(%r15),%rax
5c:   add    $0x20,%rbp
60:   pop    %r15
62:   retq
```

References

1. TIOBE Software. TIOBE Programming Community Index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, December 2013.
2. Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 1996.
3. 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.
4. National Cyber Security Division. National Vulnerability Database: Automation of Vulnerability Management. `http://nvd.nist.gov/`, December 2013.
5. Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic

Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium (USENIX '98)*, San Antonio, Texas, US, January 1998. Oregon Graduate Institute of Science and Technology.
6. StackShield: A Stack Smashing Technique Protection Tool for Linux, January 2000.
7. Saravanan Sinnadurai, Qin Zhao, and Weng-Fai Wong. Transparent Runtime Shadow Stack: Protection against Malicious Return Address Modifications. 2008.
8. Bulba Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, May 2000.
9. Gerardo Richarte. Four Different Tricks to Bypass StackShield and StackGuard Protection. Technical report, Core Security Technologies, April 2002.
10. Peter Silberman and Richard Johnson. A Comparison of Buffer Overflow Prevention Implementations and Weaknesses. In *Black Hat Briefings*, Las Vegas, July 2004.
11. 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.
12. Tyler Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, July 2002.
13. Tilo Müller and Lexi Piminedis. ASLR Smack & Laugh Reference. In *Seminar on Advanced Exploitation Techniques*. RWTH Aachen University, Germany, 2008.
14. Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy*, San Francisco, California, May 2013. Horst-Goertz Institute for IT Security, Ruhr-University Bochum, IEEE Computer Society.
15. Erik Buchanan, Ryan Roemer, and Stefan Savage. Return-Oriented Programming: Exploits Without Code Injection. In *Black Hat USA Briefings '08*, Las Vegas, NV, US, July 2008. University of California, San Diego.
16. Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, pages 27–38, Alexandria, VA, US, October 2008. University of San Diego.
17. Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 559–572, Chicago, IL, US, October 2010. ACM.
18. Edward Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Security Symposium (USENIX '11)*, San Francisco, CA, August 2011. Carnegie Mellon University, Pittsburgh.
19. 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.
20. Team Teso Scut. Exploiting Format String Vulnerabilities. `http://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf`, September 2001.
21. System V Application Binary Interface - AMD64 Architecture Processor Supplement. `http://www.x86-64.org/documentation/abi.pdf`, Oct 2013.
22. Yves Younan, Wouter Joosen, and Frank Piessens. Code Injection in C and C++: A Survey of Vulnerabilities and Countermeasures. Technical report, Katholieke Universiteit Leuven, Department of Computer Science, Belgium, July 2004.