Seminar Elaboration
about CET from Intel

Anti Exploit Technology “Control-flow Enforcement Technology”(CET) from Intel

---

Name: Stadelmann, Jakob

INHALTSVERZEICHNIS

I. Introduction

With the introduction of security features such as data execution prevention (DEP) or write-xor-execute against buffer overflow attacks, it is no longer possible to inject malicious code and execute it. The attacker can no longer write bytes in the event of a stack overflow and thus gain control of the machine.
Attackers have therefore needed a new way to execute their malicious code. They use so-called code reuse attacks. In this type of attack, attackers use existing code to then bypass DEP and write-xor-execute via system functions. These attacks are very difficult to detect because they reuse existing code from memory in a creative way. There are three different types of code reuse attacks. A distinction is made between return-oriented programming (ROP), jump-oriented programming (JOP) and call-oriented programming (COP). They all follow the same basic idea, namely to append small, already existing pieces of code. They only differ slightly in terms of implementation.

In order to be able to defend against such exploits, it must be possible to detect manipulation of the program's control flow. The goal is control-flow integrity (CFI), which means that only control flows that have been programmed by the programmer may be executed. Tools such as ROPGuard, ROPecker or Microsoft Control Flow Guard already exist, but they all implement coarse-grained CFI and therefore cannot guarantee maximum CFI.
Intel Control-flow Enforcement Technology is a CPU instruction set extension to implement CFI and to protect against ROP/JOP-type control flow subversion attacks. Control-flow Enforcement means as much as control-flow enforcement. The aim behind it is to ensure the integrity and security of a computer program. This is achieved by ensuring that the program flow only runs in a predefined and secure manner. The aim is to prevent unexpected changes to the program flow, e.g. by malicious code. CET adds a Shadow Stack - Fine Grained CFI - and Indirect Branch Tracking - Coarse Grained CFI - to the Intel ISA.

II. Code reuse attacks

Before Intel's Control-flow Enforcement Technology can be discussed, it is important to understand what CET is intended to protect against. To this end, this section looks at the attacker's side and explains how an attack is usually carried out nowadays and, above all, which techniques are used. The following steps are usually required for an exploit:

1) You need vulnerable software that you have access to. This allows the exploit to be tested again and again and improved until it works perfectly.
2) The software must provide information about the current memory layout.
3) You have to find a way to manipulate the memory. For example, a stack overflow, a heap overflow or a use after free are suitable for this.

Finally, the malicious code must be executed in order to gain control of the machine. This is achieved via code reuse attacks, the basic idea of which is explained using ROP, as ROP was the first appearance of code reuse attacks and is also the most widespread.

A. Return Oriented Programming (ROP)

1) Basic idea: The term 'Return Oriented Programming' was coined in 2007 in the article 'The Geometry of Innocent Flesh on the Bone' by Hovav Shacham [10]. The basic idea is to reuse existing code in memory by exploiting the semantics of the stack [12]. By manipulating return addresses on the stack, attackers can append small pieces of code together to form a new malicious program that was never intended by the original code author. First, however, we will explain how the normal execution flow works. There you have the instruction pointer and the instructions. The branch instructions control the program flow and specify which instructions are executed. After an instruction has been completed, the instruction pointer is incremented and points to the next instruction, which is then executed again [12]. In this way, one instruction follows the next and the instructions are executed one after the other. ROP behaves differently. There are blocks of instructions in the memory that are followed by a return. Such a block of a few instructions followed by a return is also called a gadget. The addresses of these blocks are stored on the stack. In the example in Figure **??**, the instruction pointer points to the first block of instructions. When these instructions are finished, the return is executed. When the return

is executed, the value to which the stack pointer points is copied to the instruction pointer. In this case, the address of the second code block is copied to the instruction pointer. In addition, the stack pointer is incremented by four bytes so that it points to the next element on the stack [3].

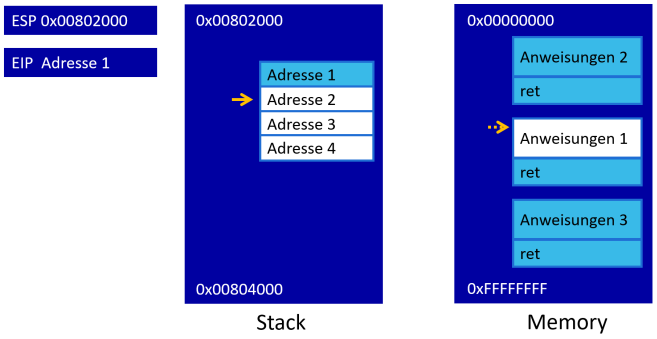

Abbildung 1. Example Programflow ROP

The instruction pointer now points to the second block which is executed, followed by the return. This allows the instruction pointer to be directed to the next code block. The stack pointer therefore controls the program flow and has taken on the role of the instruction pointer.
Since the stack is a 'read and write' memory that can be controlled if the right vulnerabilities are used, there is a way for the attacker to append several such code blocks together. These code blocks then form a new program [12].

2) Gadgets: These blocks, which consist of a few instructions and end with a return, are also called gadgets in the context of ROP. A gadget is a sequence of instructions that end with a return and perform logical operations [12].

Examples of such logical operations would be

- Copying a value into memory
- Calling system functions
- Loading values into certain registers

To illustrate this, the code example for copying data to memory can be seen in Figure 2.

When developing a ROP attack, most of the time is spent looking for gadgets. One source of gadgets are libraries that are linked into the program, either at runtime as dynamic libraries or at compile time as static libraries. One place to look for gadgets for an attack on Windows is Ntdll - a dynamic userland library [12]. There are two reasons for this:

```
POP EAX          //Load value
POP ECX          //Load destination address
mov [ECX], EAX   //Write the value to
                 //the destination address
```

Abbildung 2. Code example: Copy data in memory

- Ntdll is loaded in every process in the system. As an attacker, you don't have to hope that Ntdll is loaded, because it is always loaded. So if you find gadgets in Ntdll, you can also use these gadgets for other exploits.
- Ntdll represents the userland interface to the kernel and uses syscalls. This means that Ntdll is handwritten assembly code [12]. The code is therefore no longer touched once it is running. For attackers, this means that gadgets from Ntdll are available at least in Windows 7, sometimes even in Windows Vista. This gives attackers a lot of flexibility for their exploits and allows them to access a reliable source of gadgets.

By manipulating the stack, the attacker does not have to set the return addresses to the beginning of functions, but can also skip bytes. It is even possible to jump into the opcodes of instructions. These options offer him even more flexibility in his search for gadgets.

3) Usage: As already mentioned, the aim of ROP is to make the memory executable so that attackers can execute their malicious code and thus gain control of the machine. To achieve this, ROP attempts to call one of the following functions:

- Virtual Protect: This function can be used to change the memory protection. The attacker sets the memory to executable.
- Virtual Alloc: This function allows you to allocate executable memory, which would also be of great benefit to an attacker.

III. Software-based CFI

This section discusses the approaches that already exist to prevent such code reuse attacks, or at least to make them more difficult. One starting point for the defense against code reuse attacks is the analysis of the control flow. This has the following background: In ROP, the control flow was changed by manipulated return pointers in such a way that

the attacker was able to chain his gadgets. This enabled the attacker to create a completely new program that attacks the computer in his own way. This type of attack would therefore not be possible if attackers were not able to change the control flow in their favor.
The integrity of the control flow must therefore be guaranteed. This means that the program flow only runs in the predefined and secure manner. Above all, there must be no deviations in the program flow due to malicious code or similar. The defense strategy is therefore called control-flow integrity (CFI).

A. *Basics*

CFI follows the following principle: You have a control flow graph of the code, as shown in Figure 3, and you label the nodes. The nodes are blocks with instructions. The task of the CFI tool is to check whether the output of label A points to label B. If this is the case - i.e. it is the correct program flow - then the CFI tool gives an 'OKAY' and you can move from label A to label B.

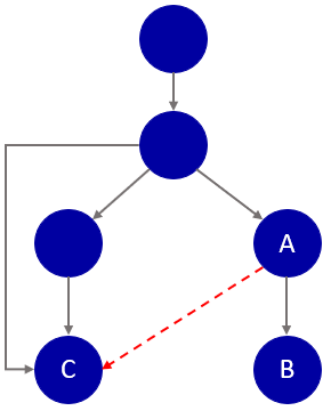

Abbildung 3. Control flow graph CFI

However, in the event that an attacker tries to direct the control flow from label A to label C, the CFI tool will terminate the process because the edge from A to C is not present in the control flow graph [2].
The solution just described is a so-called 'Fine Grained CFI'. The advantage of this solution is that every branch of the control flow graph is checked, which ensures a very high level of security. This is because only the program flow described by the control flow graph is permitted. This means that there can be no unwanted deviations in the program flow and the program flow can only run correctly and safely.
However, there are also problems with this approach. Control flow graph coverage is one of them. The control flow graph is created by a static analysis. However, if the program flow changes at runtime, a hole is created in the control flow graph [2]. This creates a security vulnerability that can be exploited by attackers
. Another problem is the performance overhead that arises when every branch has to be checked. Due to these problems, other solutions were considered and a so-called coarse-grained CFI was developed. A Coarse Grained CFI tries to be very practicable and, above all, as efficient as possible so that performance is not negatively affected too much [2]. To achieve this, the control flow graph must be made practicable.
To do this, the number of labels is reduced by using only a few labels that are used multiple times. This reduces the number of labels, which in turn reduces the number of checkpoints. However, this also means that there are now many more branching options, as the number of labels has been reduced. This means that not only the original program flow is now possible, but also deviations from it. Figure 4 shows a practicable control flow graph.

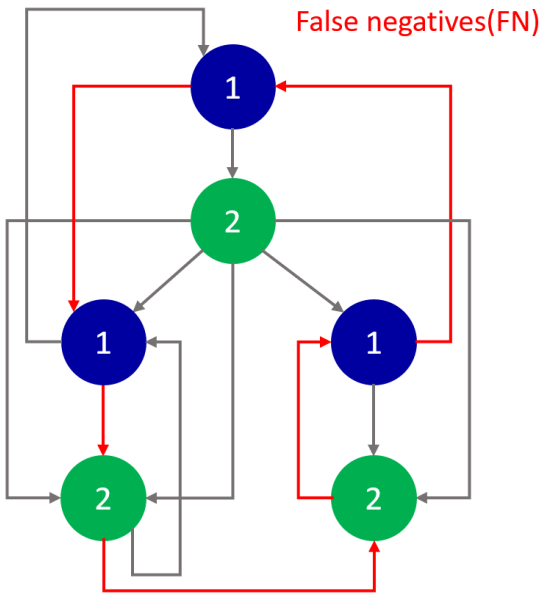

Abbildung 4. Practical control flow graph

The ambiguity of the program flow also results

in false negatives, which can also be seen in Figure **??**. Due to these false negatives, it is possible that an attack is taking place, but you are not able to recognize the attack [2]. The promise of coarse-grained CFI is therefore that efficiency is achieved by reducing the label. On the other hand, the question arises as to how to reduce the false negatives that arise.

B. CFI Policies

To reduce false negatives, CFI tools use so-called policies. The program flow is analyzed and an attempt is made to detect conspicuous behaviour in order to detect an attack.
The first policy is called 'Call Preceded Return Address' [2]. Figure 5 shows an application that calls a library function. A Fine Grained CFI only allows you to return to the statement that comes after the library function call. A Coarse Grained CFI is somewhat looser. Here you can only return to statements that come after a call statement. However, it is not permitted to return to a statement that does not follow a call. The policy thus reduces the number of possible return targets and therefore also the number of false negatives. However, several valid return targets are still permitted, which can be exploited for an attack.

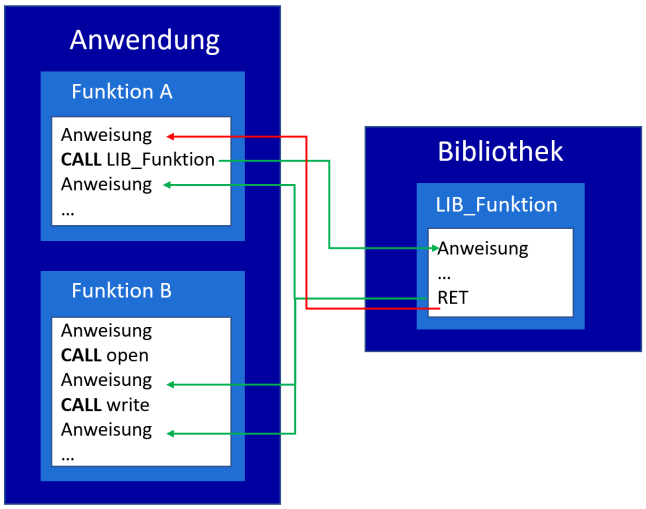

Abbildung 5. Example Call Preceded Return Address

'Chain of Short Sequences' is the second CFI policy [2]. This policy has the following ulterior motive: In a ROP attack, many gadgets consisting of only a few instructions are appended together. You therefore have a long chain of short sequences, which does not often occur in a normal program flow. For this reason, an attempt is made to detect such chains of short sequences in the program flow in order to detect a ROP attack.

C. Overview Coarse Grained CFI Tools

Now that the basics of CFI, coarse-grained CFI and their policies have been explained, this section is intended to provide a rough overview of the common coarse-grained CFI tools and how they work. One of the first observations was that it does not make sense to check every branch of the control flow graph because it is not efficient and you lose a lot of performance. For this reason, a branch is only checked if a critical function or a critical syscall is called in it. Examples of such critical functions would be the VirtualProtect and VirtualAlloc mentioned at the beginning, which are used to set memory to executable or to allocate executable memory.
The first two coarse-grained CFI tools to be looked at in more detail are kBouncer and ROPecker. They are connected to the critical functions and syscalls via a hook.This means that each time these functions are called, the two tools start their checks [2].
For the checks, the two tools apply the CFI policies just explained to the branch information in order to decide whether an attack is taking place. The branch information is obtained from a special register on the Intel CPU called 'Last Branch Records'. This register records the last 16 branches of this thread. ROPecker is also connected to the paging system [2]. For example, it loads two pages of the application code and marks them as executable. The rest is not executable. After the two pages have been checked, the rest is checked. However, the check for the rest is only triggered during the execution of the first two pages.

ROPGuard is another Coarse Grained CFI tool, which is also connected to the critical functions and syscalls. However, ROPGuard does not use the Last Branch Records Register [2]. The register is not provided by every CPU, which means that kBouncer and ROPecker are dependent on the available hardware. ROPGuard, on the other hand, is independent of the hardware and requires no additional support in order to function. ROPGuard bases its analysis on heuristics only. It simulates the behavior of the stack pointer and the decision [2].

is based on this simulation. The following assertion was made: 'Coarse grained CFI measures are sufficient to prevent real world and touring complete ROP attacks' [2]. In practice, however, this claim has proven to be false.
It is possible to bypass the coarse grained CFI tools and ROP attacks were still possible despite the coarse grained CFI tools [2].

D. Microsoft Control Flow Guard

Another CFI technology that should be mentioned is Control Flow Guard (CFG) from Microsoft. CFG is designed to ensure the integrity of indirect calls/jumps. It is a forward branch enforcement that attempts to ensure that calls/jumps can only reach a valid target. A bitmap is loaded into each process for this purpose. This bitmap specifies which virtual addresses are a valid target for an indirect call/jump [4]. Before each indirect call or jump, a check function is called that checks this bitmap. If the address to which the indirect control transfer is to take place is marked as valid, the indirect call/jump is executed. However, if the address is not marked as valid, the process terminates.
Control Flow Guard is also a coarse-grained CFI solution. It does not check whether the function that is indirectly called expects the same number of parameters or whether the parameters are of the correct data type. It is a binary that specifies whether the function can be called or not [4].
Control Flow Guard therefore does not offer maximum security and there are ways in which CFG can be bypassed.
However, the biggest problem with CFG is that most attackers attack the stack. This means that no one bypasses Control Flow Guard by attacking Control Flow Guard [4]. CFG is bypassed, by manipulating return pointers on the stack, against which CFG has no protective measures. The integrity of the return pointers must be ensured and this problem is solved by Intel's Control-flow Enforcement Technology.

IV. CET von Intel

Control-flow Enforcement Technology, CET for short, is an instruction set extension of the Intel processor family. It is important to mention that CET is implemented at hardware level, which significantly increases the security of CET.
However, the implementation at hardware level also has its downsides. CET requires special hardware support to work, and currently CET is only available on a few Intel processors. CET was first introduced in the 11th generation processors (Tiger Lake) [9] and has since been available in the newer generation processors (Alder Lake and Sapphire Rapids).
CET adds two elements to the Intel instruction set architecture. The shadow stack protects the return addresses and prevents them from being manipulated. It therefore protects against backward chaining of gadgets with return pointers.
The second of the new elements is Indirect Branch Tracking. This is protection against indirect branching. The forward chaining of gadgets via calls/jumps is therefore made more difficult. The basic idea behind Indirect Branch Tracking is similar to Control Flow Guard. Both technologies define valid targets for indirect calls/jumps, but they differ in their implementation.
In addition to the processors, the operating system must also support CET. Supported operating systems are Windows 10 (from version 2004, but only Shadow Stack implemented) [6], Windows 11 [7] and Linux from kernel 5.11 (only Indirect Branch Tracking implemented so far, Shadow Stack should be implemented from kernel 6.4) [5]. If support from the processor and operating system side is guaranteed, then CET implements control flow integrity and protects against control flow subversion attacks, such as code reuse attacks [11].

A. Attack Model and Goals of CET

During the development of CET, the following assumptions were made regarding the attacker's capabilities [11]. The opponent can...

- Find software vulnerabilities that allow the adversary to read and write anywhere in virtual memory.
- reveal the complete layout of the address space and know where e.g. stacks, heaps or images are mapped.
- repeat read and write accesses in memory as often as required (arbitrary read + arbitrary write vulnerability).
- Generate stimuli that cause the code to take different paths. This allows the attacker to observe the state of the program and also the state of the stack on these paths.

- Send data to a computation server in order to calculate required payloads for subsequent reads or writes.
- Perform control transfer to existing executable code to change the state of processor registers.

However, the opponent cannot...

- add new code without verification and the existing code is read-only and cannot be modified (e.g. write-xor-execute policy).

Based on these assumptions, the following goals were defined for CET [11]. CET must...

- provide a protection mechanism against code reuse and memory safety errors for new architectural elements, such as new hardware registers and memory.
- to be applicable to CPU authorization levels (user/supervisor).
- to be applicable to CPU modes used by commercially available software, such as 32/64-bit, hypervisor, system management mode, enclaves, etc.
- ensure that control flow protection is also guaranteed at transition points of mode or context changes.
- have minimal impact on performance, memory consumption and code growth.

In addition, the following constraints were defined that CET should comply with [11]:

- Avoid embedding programming language-specific constructs in the Instruction Set Architecture.
- Provide only necessary (minimal) capabilities.
- Retain the stack/function calls.
- No restriction of common software constructs (e.g. tail-calls, co-routines, etc).

B. Shadow Stack

The shadow stack is the first of the newly added elements. The shadow stack is a second stack that is used exclusively for control transfer operations. It is separate from the data stack and only stores return addresses, i.e. no data or parameters are held [1]. The shadow stack enforces that returns can only have the correct address as their destination. Only the original program flow is possible, which means that the shadow stack implements a fine-grained CFI.
Furthermore, the shadow stack is operated at hardware level, which significantly increases security as the CPU is responsible for managing the shadow stack. To prevent unwanted writes to the shadow stack by software, it is write-protected [11]. The CPU enforces that software can only write to the shadow stack in the context of a call. This means that the values on the shadow stack cannot be manipulated by software.

1) Functionality: To ensure that the shadow stack returns to the correct return address, the call statement and the return statement have been modified. The call statement has been modified so that it not only pushes the return address to the data stack, but also pushes a copy of the return address to the shadow stack [11]. The return statement has been modified so that it pops the top return address from both stacks and compares the two addresses [11].
If the return address on the data stack has not been manipulated and is the same as the address that was popped from the shadow stack, then the 'OKAY' is given and the return can be executed. However, if the return address on the data stack has been manipulated, then the comparison of the two return addresses fails and a Control Protection (#CP) exception [8] is thrown. This is a new fault type exception introduced by CET [11]. It informs privileged software that a control flow violation has occurred
. A few new instructions have been introduced for the management of the shadow stack [11], which are now explained in more detail.
The *INCSSP* instruction can be used to increase the shadow stack pointer. *RDSSP* allows you to read the shadow stack pointer. It is also possible to save the previous shadow stack pointer and restore this saved shadow stack pointer. This works with the two instructions *SAVEPREVSSP* and *RSTORSSP*, which are required for the shadow stack switch. With *WRSS/WRUSS* you can write to the shadow stack. There is also a busy flag that you can either set or delete with the *SETSSBSY/CLRSSBSY* instructions. With this busy flag, you have the option of activating or deactivating a shadow stack.

2) Shadow Stack Switch: In the following, the shadow stack switch will be discussed in more detail. You need to be able to switch back and forth between different shadow stacks because different processes and threads are running at the same time [8]. Each of these processes and threads has its own shadow stack. If the scheduler now schedules a new thread, it switches from the shadow stack of the current thread to the shadow stack of the next thread. As already mentioned, the two instructions SAVEPREVSSP and RSTORSSP are provided for the switch in order to carry out the switch in a controlled manner. If the scheduler switches away from an active shadow stack and later returns to this shadow stack, the shadow stack pointer (SSP) must be exactly the same as when the shadow stack was switched away [11]. The SSP must therefore point to the same address on the shadow stack again to ensure the security of the shadow stack.
The switch is a two-stage process: First, *RSTORSSP* is executed to verify the new shadow stack and switch to it. To save a restore point, a so-called *shadow stack restore token*, on the old shadow stack, *SAVEPREVSSP* is executed.

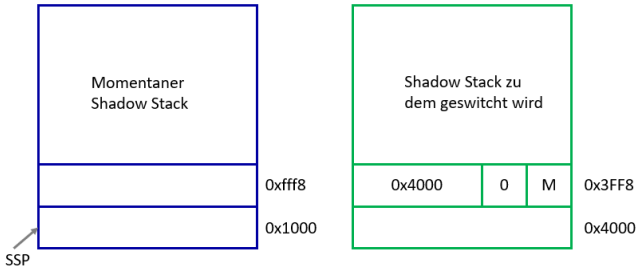

Abbildung 6. Shadow Stacks before Switch

Figure 6 shows two shadow stacks and the aim is to switch from the left shadow stack (active) to the right shadow stack.

The shadow stack pointer currently points to the address 0x1000 of the active shadow stack. There is a *shadow stack restore token* at address 0x3FF8 on the shadow stack to be swapped to. This means that the shadow stack on the right was already active once, but it was switched away from and is now to be restored. The *shadow stack restore token* is a 64-bit value and is structured as follows:

- Bit 63:2 - 4-byte aligned SSP for which the restore point was created. The address indicates where the SSP was located when it was switched away from the shadow stack. This SSP must be located at an address that is 8 or 12 bytes above the address of the *shadow stack restore tokens*. This property is checked in the *RSTORSSP* statement [11].
- Bit 1 - reserved. Must be 0 [11].
- Bit 0 - also called mode bit. If the mode bit is 0, then the *shadow stack restore token* can be used by a *RSTORSSP* instruction in 32 bit mode can be used. If it is 1, the *shadow stack restore token* can be used by a *RSTORSSP* instruction can be used in 64 bit mode [11].

The token therefore specifies that the SSP is to be restored at address 0x4000. The *RSTORSSP* instruction is called with the argument 0x3FF8, as the instruction requires the address of a *shadow stack restore token*. First, *RSTORSSP* compares the mode of the machine with the set mode bit. It then checks whether the reserved bit at position 1 is 0 and whether the address stored in the token, in this case 0x4000, is 8 or 12 bytes away from the address of the token. If all checks were successful, the SSP is now set to 0x3FF8.
In addition, the *shadow stack restore token* is replaced by a *previous SSP token*. This *previous SSP token* has the following form:

- bit 63:2 - previous SSP that points to the top of the old shadow stack. This is therefore the SSP that was active when RSTORSSP was called [11].
- bit 1 - is set to 1 and shows that it is a *previous SSP token* [11].
- bit 0 - also called mode bit. If the mode bit is 0, then the *previous SSP toke* can be used by a *SAVEPREVSSP* instruction in 32 bit mode can be used. If it is 1, the *previous SSP token* can be used by a *SAVEPREVSSP instruction* in 64 bit mode [11].

After switching to the new shadow stack, a restore point can now be created with *SAVEPREVSSP* on the old shadow stack. *SAVEPREVSSP* uses the *previous SSP token* generated by RSTORSSP to create the *shadow stack restore token*. To do this, *SAVEPREVSSP* does not need an operand, it consumes the *previous SSP token* that is on top of the new shadow stack.
SAVEPREVSSP first verifies the address stored in the *previous SSP token*. Then the 8 bytes of the

previous SSP token are popped from the new shadow stack. Then it is checked again whether the bit at position 1 is 1 and whether the mode bit matches the mode of the machine. If everything is correct, a *shadow stack restore token* is pushed to the old shadow stack. After these steps, the system has successfully switched from one shadow stack to another shadow stack [11]. The state of the shadow stacks after these steps can be seen in Figure 7.

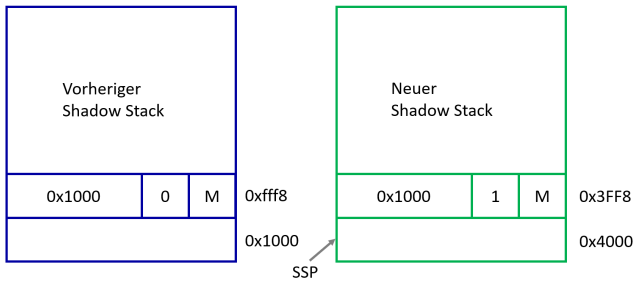

Abbildung 7. Shadow Stacks after Switch

If you do not need a restore point on the old shadow stack, you can also pop the *previous SSP token* from the new shadow stack with the *INCSSP* instruction.

C. Indirect Branch Tracking

The second new element is Indirect Branch Tracking. It introduces the new ENDBR instruction. ENDBR marks valid code targets for indirect calls or jumps [11]. This means that if an indirect call/jump has an instruction other than ENDBR as its target, a #CP exception is thrown. This prevents and reports attempts to direct the control flow to unwanted targets.

Indirect branch tracking is a coarse-grained CFI solution, as the indirect branch does not have to have the correct and secure address to the target. However, it prevents jumping to the middle of functions.

The opcodes of the new instruction were chosen so that it behaves like a NOP instruction on Intel 64 processors that do not support CET [11]. On processors that support CET, ENDBR behaves almost like a NOP instruction because it does not load any additional registers, it does not change the state of the program and it has minimal impact on the performance of the program [11].

Indirect Branch Tracking implements two identical state machines. One is for the user mode and the other for the supervisor mode [11]. The structure of the state machine can be seen in Figure 8. The start state of the state machine is the IDLE state. If an instruction other than an indirect call/jump occurs, the state machine remains in the IDLE state. The state machine only switches to the WAIT_FOR_ENDBRANCH state in the event of an indirect branch (call/jump).

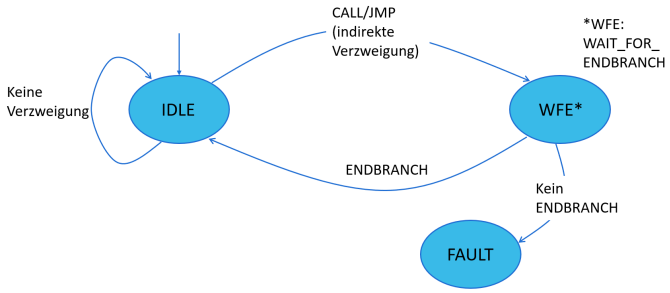

Abbildung 8. Indirect Branch Tracking State Machine

In this state, the state machine will throw a #CP exception with the error code 'ENDBRANCH' if the next instruction is not an ENDBR. In the event that the next statement is the correct ENDBR, the system returns to the IDLE state.

D. results

The security of CET and the performance of CET are now evaluated.

1) Security: One way to evaluate the security of CET is to calculate the average indirect target reduction, or AIR for short. This is a metric that can be used to calculate the strength of control-flow integrity [11]. It represents a set of addresses that can be reached via indirect control transfers. The AIR metric is described with the following formula [11]:

$$\sum_{i=1}^{n} \frac{1 - |T_i|}{S} \tag{1}$$

Here n is the total number of indirect branch points in the program. S represents the set of program addresses to which all indirect branch points can direct the control flow without CFI protection. And T_i represents the set of program addresses to which the ith indirect branch point can direct the control flow with CFI protection measures [11].

A low AIR value means that a large set of addresses can be reached via indirect control transfers. As a

result, the control flow integrity is low. A high AIR value, on the other hand, stands for high control flow integrity, as only a small set of addresses can be reached with indirect control transfers.
On the x86 architecture, indirect control transfers can reach every byte in the program. It follows that S is the size of the program code [11]. Consequently, the AIR value is very low.
With CET activated, a return statement can reach exactly one target in the program, namely the return address at the top of the shadow stack. Indirect calls/jumps can only have an ENDBR as the target.

An average AIR metric of 99.8% was calculated for the SPEC CPU 2006 C/C++ benchmark [11]. Values between 99.3% and 99.99% were represented for individual programs.

Furthermore, the Linux kernel binary was searched for available gadgets [11]. If there are many gadgets in a binary, it is easier for an attacker to carry out a ROP attack. For this reason, the number of gadgets found in a binary gives a good indication of the binary's security.
The Linux kernel was scanned for available gadgets using the ROPGadget tool. The Linux binary used was a default configuration with a size of 25 MB. The ROPGadget tool was restricted to search for gadgets up to a maximum size of 10 bytes. Under these conditions, 197241 gadgets were found - however, it can be assumed that there are even more gadgets in the binary, as the size of the gadgets was limited. This large number of gadgets shows the size of the available attack surface of the binary.
Figure 9 shows the distribution of the different gadget sizes. For example, 18000 gadgets with a size of 2 bytes, 20000 gadgets with a size of 4 bytes and 25000 with a size of 9 bytes were found.

In contrast, in a binary that supports CET, the attacker is restricted to using functions that have an ENDBRANCH and return to the last address on the shadow stack [11]. In the analyzed Linux kernel binary 18412 such functions were found. These functions can also no longer be concatenated with a malicious return, but must be concatenated with an indirect call/jump. The functions had an average size of 214 bytes. This size results in larger side effects, which leads to an increase in complexity when carrying out COP attacks [11].
Figure 10 shows how much the number of gadgets

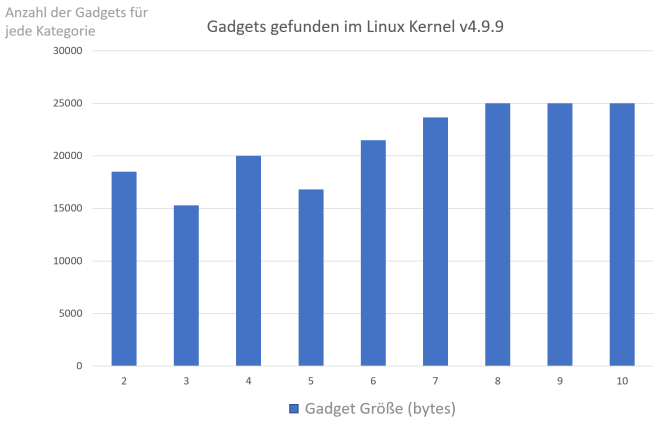

Abbildung 9. Gefunde Gadgets im Linux Kernel v4.9.9 [11]

that can be used for a code reuse attack has been reduced. In addition, the chaining of the remaining gadgets has also become significantly more difficult. By eliminating the majority of unwanted gadgets, the attack surface of the Linux kernel has been reduced many times over. Only a small number of functions can be used for a code reuse attack. The smaller attack surface can be systematically analyzed to eliminate unneeded cases or to address insecure constructs through redesign or targeted checks.
The security of the Linux kernel binary has therefore increased considerably by activating CET.

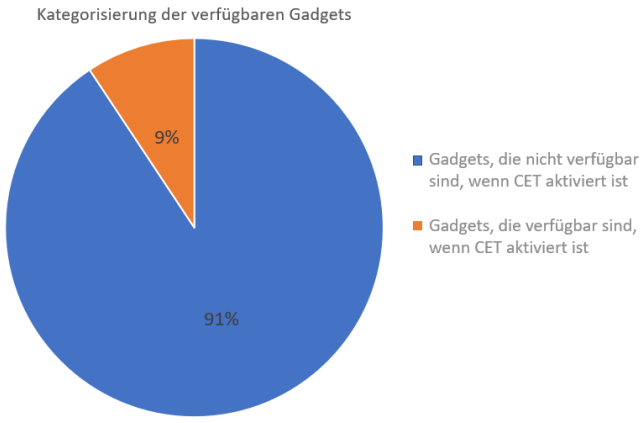

Abbildung 10. Categorization of the available gadgets

2) Performance: The performance of CET and in particular the performance of the shadow stack is particularly interesting. The shadow stack implements a fine grained CFI and this can only be implemented with a performance overhead. The performance of the shadow stack was evaluated using a series of microprocessor benchmarks and

application traces executed on a cycle-accurate processor performance model [11].
The call instruction was updated to make an additional push to the shadow stack and the return instruction was updated to pop the return address from the shadow stack and compare it to the address from the data stack. The geometric mean of the instruction-per-cycle (IPC) loss across the workload traces was approximately 1.65% [11]. The IPC loss ranged from 0.08% to 2.71%.

The impact of Indirect Branch Tracking on performance was evaluated by compiling C/C++ programs from the SPEC CPU 2006 C/C++ using an ICC compiler that supports CET. Since the ENDBRANCH instruction behaves like a NOP instruction on processors - with or without CET support - no noticeable slowdown was measured on average

E. Vulnerabilities

However, CET also has vulnerabilities and can therefore also be circumvented. These include code replacement attacks, counterfeit object-oriented programming (COOP) and data-only corruption, to name a few [8]. This section focuses on code replacement attacks, as this is a very creative solution to circumvent the shadow stack.
The principle is explained using the following example [4]. The figure 11 shows a small call stack. FOO.dll is located on the call stack. Foo.dll has a function called StartWork, which calls another function called DoWork, which is located in a second dll. It is assumed that DoWork is currently in a loop and is calculating something.

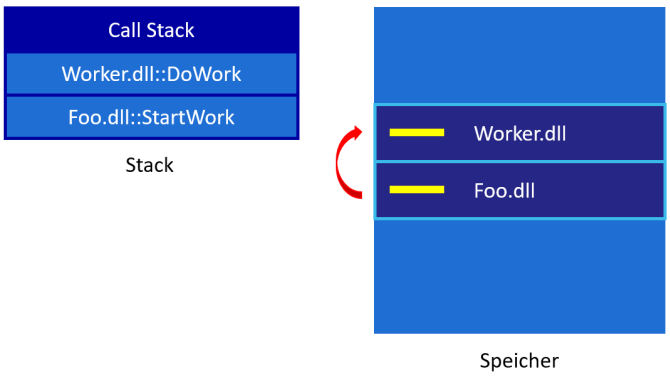

Abbildung 11. Example Code Replacement Attack

During this time, the attacker forces FOO.dll to be unloaded, e.g. by data corruption. At the exact location where FOO.dll was located, the attacker now loads something completely different, e.g. another dll that he has chosen. What the attacker has loaded is therefore located at exactly the same virtual address at which FOO.dll was mapped.
When DoWork now returns, it uses the return address that is stored on the shadow stack. The pointer still points to the virtual address where FOO.dll was loaded. However, something completely different is now located there and the return pointer is now a dangling pointer. It therefore returns to the dll loaded by the attacker, which could be the starting point for a ROP attack.

V. Conclusion

As has just been shown, CET also has its weaknesses and is not completely secure. While CET is a very important step in enforcing control flow integrity and ensuring the security of computers, CET alone is not enough.

It is better to think of control flow integrity and computer program security as a castle made up of many pieces of wall. The pieces of the wall work together and support each other to ensure the common goal of 'security'. One of these wall pieces is CET, which is also a very large and important wall piece and ensures a high level of control flow integrity. But even CET can be circumvented if the other pieces of the wall were not in place.

Therefore, a combination of different technologies is the only way to maximize the security of a computer program. Another piece of the wall would be, for example, Data Execution Prevention (DEP) - which is even an assumption of CET - to prevent the injection of malicious code. PAC (Pointer Authentication Codes), to ensure the integrity of pointers, is also an important part of the castle.

In addition, CET can only really develop its effect against ROP attacks once it has been introduced to the market on a large scale, which will still take some time.

Literatur

[1] *A Technical Look at Intel's Control Flow Enforcement Technology*. URL: url : https : // www . intel . com / content / www / us / en / developer / articles / technical / technical - look - control - flow - enforcement - technology . html (besucht am 25. 04. 2023).

[2] Daniel Lehmann Ahmad-Reza Sadeghi. *The Beast is in Your Memory*. URL: https://youtu.be/iCaw0dwlIRU (besucht am 22. 04. 2023).

[3] Maryam Rostamipour AliAkbar Sadeghi Salman Niksefat. *Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions*. 2017. DOI: 10.1007/s11416-017-0299-1. URL: https://doi.org/10.1007/s11416-017-0299-1 (besucht am 02. 05. 2023).

[4] Joe Bialek. *The Evolution of CFI Attacks and Defenses*. URL: https://youtu.be/oOqpl-2rMTw (besucht am 23. 04. 2023).

[5] Brittany Day. *Intel CET Shadow Stack Support Set To Be Introduced With Linux 6.4*. 2023. URL: https://linuxsecurity.com/news/security-projects/intel-cet-shadow-stack-support-set-to-be-introduced-with-linux-6-4 (besucht am 04. 05. 2023).

[6] Tom Garrison. *Intel CET Answers Call to Protect Against Common Malware Threats*. 2020. URL: https://newsroom.intel.de/editorials/intel-cet-answers-call-to-protect-against-common-malware-threats/#gs.yly7if (besucht am 04. 05. 2023).

[7] Intel. *Windows 11 Security Starts with an Intel Hardware Security Foundation*. 2022. URL: https://www.intel.com/content/www/us/en/content-details/752405/windows-11-security-starts-with-an-intel-hardware-security-foundation-whitepaper.html (besucht am 04. 05. 2023).

[8] Chong Xu Jin Liu Bing Sun. *How to Survive the Hardware Assisted Control-Flow Integrity Enforcement*. URL: https://youtu.be/D4-YwGYYcTg (besucht am 22. 04. 2023).

[9] Jakob Jung. *Intel führt CET-Sicherheit ein*. 2020. URL: https://www.zdnet.de/88380746/intel-fuehrt-cet-sicherheit-ein/ (besucht am 04. 05. 2023).

[10] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)". In: *Proceedings of CCS 2007*. Hrsg. von Sabrina De Capitani di Vimercati und Paul Syverson. ACM Press, Okt. 2007, S. 552–61.

[11] Vedvyas Shanbhogue, Deepak Gupta und Ravi Sahita. "Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity". In: 2019. ISBN: 9781450372268. DOI: 10.1145/3337167.3337175. URL: https://doi.org/10.1145/3337167.3337175 (besucht am 20. 04. 2023).

[12] Omer Yair. *Exploiting Windows Exploit Mitigation for ROP Exploits*. URL: https://youtu.be/gIJOtP1AC3A (besucht am 22. 04. 2023).