Seminarausarbeitung
über CET von Intel
im Sommersemester
2023

Anti Exploit Technik "Control-flow Enforcement Technology"(CET) von Intel

---

Nachname, Vorname:	Stadelmann, Jakob
Matrikelnummer:	00121716
Studiengang:	Informatik
Semester:	4
E-mail:	jas4613@thi.de
Abgabedatum:	6. Juni 2023
Modul:	IT-Sicherheit
Prüfer:	Prof. Dr. Stefan Hahndel
Zweitprüfer:	Prof. Dr. Thomas Grauschopf

INHALTSVERZEICHNIS

I. Einführung

Durch die Einführung von Security Features, wie data execution prevention (DEP), oder write-xor-execute gegen Buffer Overflow Angriffe, ist es nicht mehr möglich bösartigen Code zu injezieren und diesen auszuführen. Der Angreifer kann bei einem Stack Overflow nicht mehr bytes schreiben und dadurch Kontrolle über die Maschine erlangen.
Deswegen haben Angreifer einen neuen Weg gebraucht, um ihren bösartigen Code auszuführen. Sie verwenden sogenannte code reuse attacks. Bei dieser Art von Angriff verwenden Angreifer bestehenden Code, um dann über Systemfunktionen DEP und write-xor-execute auszuhebeln. Diese Angriffe sind sehr schwer zu entdecken, da sie bereits existierenden Code aus dem Speicher auf eine kreative Art und Weise wiederverwenden. Es gibt drei unterschiedliche Arten von code reuse attacks. Man unterscheidet zwischen return oriented programming (ROP), jump oriented programming (JOP) und call oriented programming (COP). Sie verfolgen alle dieselbe Grundidee, nämlich kleine, bereits existierende Codestückchen aneinandezuhängen. Lediglich bei der Umsetzung unterscheiden sie sich ein wenig. Um sich gegen solche Exploits verteidigen zu können, muss eine Manipulation des Kontrollflusses des Programms erkannt werden können. Das Ziel ist Control-flow Integrity (CFI), das bedeuted es dürfen nur Kontrollflüsse ausgeführt werden, die vom Programmierer programmiert wurden. Es existieren bereits Tools, wie ROPGuard, ROPecker oder Microsoft Control Flow Guard, die jedoch alle eine Coarse Grained CFI umsetzen und daher auch nicht maximale CFI gewährleisten können.
Intel Control-flow Enforcement Technology ist eine CPU Befehlssatzerweiterung zur Implementierung von CFI und zum Schutz vor ROP/JOP-artige Kontrollfluss-Subversionsangriffe. Control-flow Enforcement bedeutet so viel wie Kontrollfluss Durchsetzung. Das Ziel dahinter ist es die Integrität und Sicherheit eines Computerprogramms zu gewährleisten. Dies erreicht man dadurch, dass der Programmfluss nur auf die vordefinierte und sichere Weise verlaufen darf. Dabei sollen unerwartete Änderungen des Programmflusses durch z.B. bösartigen Code verhindert werden. CET fügt einen Shadow Stack - Fine Grained CFI - und Indirect Branch Tracking - Coarse Grained CFI - zur Intel ISA hinzu.

II. Code reuse attacks

Bevor auf die Control-flow Enforcement Technology von Intel eingegangen werden kann, muss verstanden werden gegen was CET schützen soll. Dazu wird in diesem Abschnitt die Seite des Angreifers betrachtet und es wird erklärt, wie heutzutage meistens ein Angriff abläuft und vor allem welche Techniken dazu verwendet werden. Für einen Exploit werden heute meist folgende Schritte benötigt:

1) Man braucht eine angreifbare Software, auf die man Zugriff hat. Dadurch kann der Exploit immer wieder getestet und solange verbessert werden, bis er perfekt funktioniert.
2) Die Software muss Informationen über das aktuelle Speicher Layout bieten.
3) Man muss einen Weg finden, um den Speicher zu manipulieren. Dazu eignen sich zum Beispiel ein Stack Overflow, ein Heap Overflow, oder ein Use after Free.

Abschließend muss der bösartige Code noch ausgeführt werden, um die Kontrolle über die Maschine zu erlangen. Dies erreicht man über code reuse attacks, deren Grundidee anhand von ROP erläutert wird, da ROP die erste Erscheinung von code reuse attacks war und auch am weitesten verbreitet ist.

A. Return Oriented Programming (ROP)

1) Grundidee: Der Begriff 'Return Oriented Programming' wurde 2007 in dem Artikel 'The Geometry of Innocent Flesh on the Bone' von Hovav Shacham geprägt [10]. Die Grundidee ist es existierenden Code im Speicher wiederzuverwenden, indem man die Semantik des Stacks ausnutzt [12]. Durch die Manipulation von return-Adressen auf dem Stack können Angreifer kleine Codestückchen so aneinanderhängen, dass sie ein neues bösartiges Programm formen, das nie im Sinne des ursprünglichen Code-Autors war.
Zuerst wird jedoch erklärt, wie beim normalen Ausführungfluss die Ausführung abläuft. Man hat dort den Instruction-Pointer und die Anweisungen. Die Branch-Anweisungen kontrollieren den Programmfluss und geben an welche Anweisungen ausgeführt werden. Nachdem eine Anweisung fertig ist, wird der Instruction-Pointer erhöht und zeigt auf die nächste Anweisung, die dann auch wieder ausgeführt wird [12]. So folgt eine Anweisung auf die nächste und die Anweisungen werden nacheinander ausgeführt.

Bei ROP verhält sich das ganze anders. Dort hat man im Speicher Blöcke von Anweisungen liegen, die von einem return gefolgt werden. So einen Block von wenigen Anweisungen, die von einem return gefolgt werden, nennt man auch Gadgets. Auf dem Stack liegen die Adressen dieser Blöcke. In dem Beispiel in Abbildung 1 zeigt der Instruction-Pointer auf den ersten Block von Anweisungen. Wenn diese Anweisungen fertig sind, wird das return ausgeführt. Bei der Ausführung des returns wird der Wert auf den der Stack-Pointer zeigt in den Instruction-Pointer kopiert. In diesem Fall wird die Adresse des zweiten Codeblocks in den Instruciton-Pointer kopiert. Außerdem wird der Stack-Pointer noch um vier bytes erhöht, sodass er auf das nächste Element auf dem Stack zeigt [3].

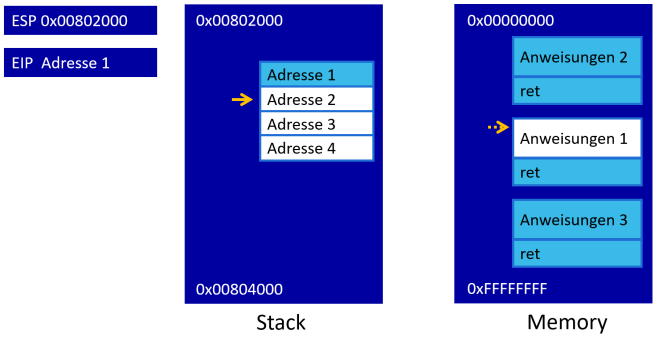

Abbildung 1. Beispiel Programmfluss ROP

Der Instruction-Pointer zeigt nun auf den zweiten Block welcher ausgeführt wird, anschließend folgt wieder das return. Wodurch der Instruction-Pointer auf den nächsten Codeblock gelenkt werden kann. Es kontrolliert also der Stack-Pointer den Programmfluss und hat die Rolle des Instruction-Pointers übernommen.
Da der Stack ein 'read and write' Speicher ist, den man kontrollieren kann, wenn man die richtigen Schwachstellen nutzt, gibt es eine Möglichkeit für den Angreifer mehrerer solcher Codeblöcke aneinanderzuhängen. Diese Codeblöcke bilden dann ein neues Programm [12].

2) Gadgets: Diese Blöcke, die aus wenigen Anweisungen bestehen und mit einem return enden, nennt man im Zusammenhang mit ROP auch Gadgets. Ein Gadget ist eine Sequenz von Anweisungen die mit einem return enden und logische Operationen ausführt [12].

Beispiele für solche logischen Operationen wären:

- Kopieren eines Wertes in den Speicher
- Aufrufen von Systemfunktionen
- Laden von Werten in bestimmte Register

Zur Veranschaulichung ist in Abbildung 2 das Codebeispiel für das Kopieren von Daten in den Speicher zu sehen.

```
POP EAX          //Laden eines Wertes
POP ECX          //Laden der Zieladresse
mov [ECX], EAX   //Schreiben des Wertes
                 //zur Zieladresse
```

Abbildung 2. Codebeispiel: Kopieren von Daten in den Speicher

Bei der Entwicklung eines ROP-Angriffes wird die meiste Zeit dafür verwendet nach Gadgets zu suchen. Eine Quelle für Gadgets sind Bibliotheken, die in das Programm verlinkt sind, entweder zur Laufzeit als dynamische Bibliotheken oder zur Kompile-Zeit als statische Bibliotheken. Ein Ort an dem für einen Angriff auf Windows nach Gadgets gesucht werden sollte ist Ntdll - eine dynamische userland Bibliothek [12]. Dafür gibt es zwei Gründe:

- Ntdll ist in jedem Prozess im System geladen. Man muss als Angreifer also nicht hoffen, dass Ntdll geladen ist, denn es ist immer geladen. Wenn man also Gadgets in Ntdll findet, kann man diese Gadgets auch für andere Exploits verwenden.
- Ntdll stellt das userland interface zum Kernel dar und nutzt syscalls. Dies bedeutet, dass es sich bei Ntdll um handgeschriebenen Assembly-Code handelt [12]. Der Code wird also nicht mehr angefasst, sobald er einmal läuft. Das bedeutet für Angreifer, dass Gadgegts aus Ntdll mindestens in Windows 7 verfügbar sind, teilweise sogar in Windows Vista. Der Angreifer erhält dadurch viel Flexibilität für seine Exploits und kann auf eine verlässliche Quelle von Gadgets zurückgreifen.

Dadurch, dass der Angreifer den Stack manipuliert, muss er die return-Adressen nicht auf den Anfang von Funktionen setzen, sondern er kann auch bytes überspringen. Es ist sogar möglich in die Opcodes von Anweisungen zu springen. Diese Möglichkeiten bieten ihm noch mehr Flexibilität auf der Suche nach Gadgets.

3) Usage: ROP hat, wie schon erwähnt, zum Ziel den Speicher ausführbar zu machen, damit Angreifer ihren bösartigen Code ausführen können und so Kontrolle über die Maschnine zu erlangen. Um dies zu erreichen versucht ROP eine der folgenden Funktionen aufzurufen:

- Virtual Protect: Mit dieser Funktion kann man den Speicher-Schutz verändern. Als Angreifer setzt man den Speicher auf ausführbar.
- Virtual Alloc: Diese Funktion erlaubt es ausführbaren Speicher zu allozieren, was einem Angreifer auch von großem Vorteil wäre.

III. Softwarebasierende CFI

In diesem Abschnitt wird behandelt, welche Ansätze schon existieren, um solche code reuse attacks zu verhindern, oder sie zumindest zu erschweren. Ein Ansatzpunkt für die Verteidigung gegen code reuse attacks ist die Analyse des Kontrollflusses. Dies hat folgenden Hintergrund: Bei ROP wurde der Kontrollfluss durch manipulierte return-Pointer so verändert, dass der Angreifer in der Lage war seine Gadgets zu verketten. Dadurch konnte der Angreifer ein völlig neues Programm erschaffen, welches in seinem Sinne den Computer angreift. Diese Art von Angriffen wäre also gar nicht möglich, wenn Angreifer nicht in der Lage wären den Kontrollfluss in ihrem Sinne zu verändern.
Es muss also die Integrität des Kontrollflusses gewährleistet werden. Das bedeutet, dass der Programmfluss nur auf die vordefinierte und sichere Weise abläuft. Vor allem darf es zu keinen Abweichungen des Programmflusses durch bösartigen Code oder Ähnlichem kommen. Die Verteidigungsstrategie heißt also Kontrollfluss-Integrität oder auf Englisch Control-flow Integrity (CFI).

A. Grundlagen

CFI verfolgt folgendes Prinzip: Man hat einen Kontrollflussgraphen vom Code, wie in Abbildung 3 zu sehen, und man labelt die Knoten. Die Knoten sind Blöcke mit Anweisungen. Die Aufgabe des CFI Tools ist es zu überprüfen, ob der Ausgang von Label A auf Label B zeigt. Wenn das der Fall ist - es handelt sich also um den richtigen Programmfluss - dann gibt das CFI Tool ein 'OKAY' und es darf von Label A zu Label B übergegangen werden.

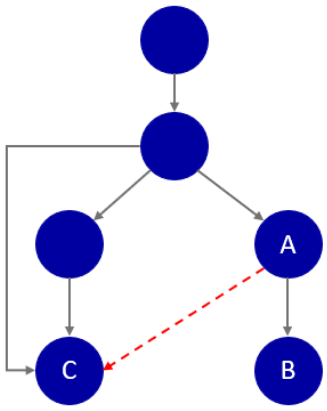

Abbildung 3. Kontrollflussgraph

Für den Fall jedoch, dass ein Angreifer versucht den Kontrollfluss von Label A auf Label C zu lenken, dann wird das CFI Tool den Prozess terminieren, da die Kante von A nach C nicht im Kontrollflussgraphen vorhanden ist [2].
Bei der gerade beschriebenen Lösung handelt es sich um eine sogenannte 'Fine Grained CFI'. Der Vorteil dieser Lösung ist, dass jeder Zweig des Kontrollflussgraphen überprüft wird, wodurch man eine sehr hohe Sicherheit gewährleisten kann. Denn es ist nur der Programmfluss erlaubt, der durch den Kontrollflussgraphen beschrieben wird. Es kann also zu keinen ungewollten Abweichungen des Programmflusses kommen und der Programmfluss kann nur auf die korrekte und sicher Weise ablaufen.
Es gibt jedoch auch Probleme mit diesem Ansatz. Die Kontrollflussgraph-Abdeckung ist eines davon. Der Kontrollflussgraph wird durch eine statische Analyse erstellt. Wenn sich jedoch der Programmfluss zur Laufzeit ändert, entsteht ein Loch im Kontrollflussgraphen [2]. Es bildet sich eine Sicherheitslücke, die von Angreifern ausgenutzt werden kann.
Ein weiteres Problem ist der Performance-Overhead, der entsteht wenn jede Verzweigung überprüft werden muss. Aufgrund dieser Probleme wurde über andere Lösungen nachgedacht und es wurde eine sogenannte Coarse Grained CFI entwickelt. Eine Coarse Grained CFI versucht sehr praktikabel zu sein und vor allem so effizient wie möglich, damit die Performance nicht zu sehr negativ beeinflusst wird [2]. Dafür muss der Kontrollflussgraph praktikabel gemacht werden.

Dazu reduziert man die Anzahl der Label, indem man nur ein paar Label verwendet, die jedoch mehrfach verwendet werden. Dadurch hat man nicht mehr so viele Labels, was zur Folge hat, dass sich die Anzahl der Checkpoints reduziert. Das bedeutet jedoch auch, dass es nun viel mehr Möglichkeiten der Verzweigung gibt, da die Anzahl der Label reduziert wurde. Es ist nun also nicht mehr nur der originale Programmfluss möglich, sondern auch Abweichungen davon. In Abbildung 4 ist so ein praktikabler Kontrollflussgraph zu sehen.

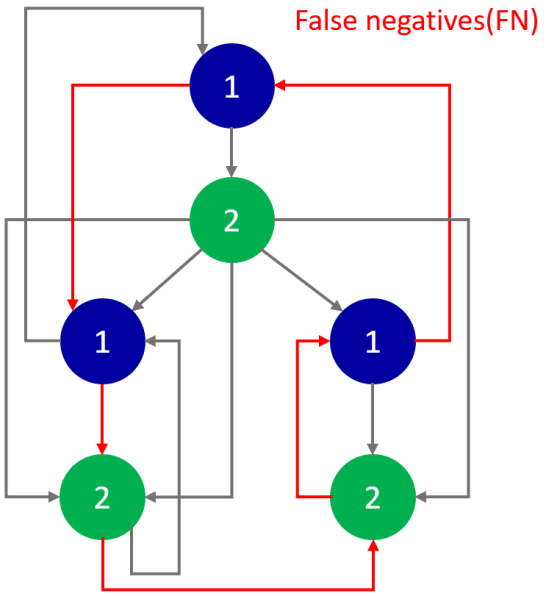

Abbildung 4. Praktikabler Kontrollflussgraph

Die Mehrdeutigkeit des Programmflusses hat auch False Negatives zur Folge, die auch in Abbildung 4 zu sehen sind. Durch diese False Negatives kann es sein, dass ein Angriff stattfindet, man aber jedoch nicht in der Lage ist den Angriff wahrzunehmen [2]. Die Promisse der Coarse Grained CFI ist also, dass man durch die Reduzierung der Label Effizienz erhält. Auf der anderen Seite stellt sich die Frage, wie man die entstandenen False Negatives reduzieren kann.

B. CFI Policies

Um die False Negatives zu reduzieren verwenden CFI Tools sogenannte Policies. Dabei wird der Programmfluss analysiert und es wird versucht auffälliges Verhalten zu entdecken, um so einen Angriff zu erkennen.

Die erste Policy nennt sich 'Call Preceded Return Address' [2]. In Abbildung 5 ist eine Anwendung zu sehen, die eine Bibliotheksfunktion aufruft. Eine Fine Grained CFI erlaubt nur zu der Anweisung zurückzukehren, die nach dem call-Aufruf der Bibliotheksfunktion steht. Eine Coarse Grained CFI ist etwas lockerer. Dort darf man nur zu Anweisungen zurückkehren, die nach einer call-Anweisung stehen. Jedoch ist es nicht erlaubt zu einer Anweisung zurückzukehren, die nicht nach einem call steht. Die Policy reduziert dadurch die Anzahl der möglichen return-Ziele und somit auch die Anzahl der False Negatives. Es sind aber immer noch mehrere valide return-Ziele erlaubt, die für einen Angriff ausgenutzt werden können.

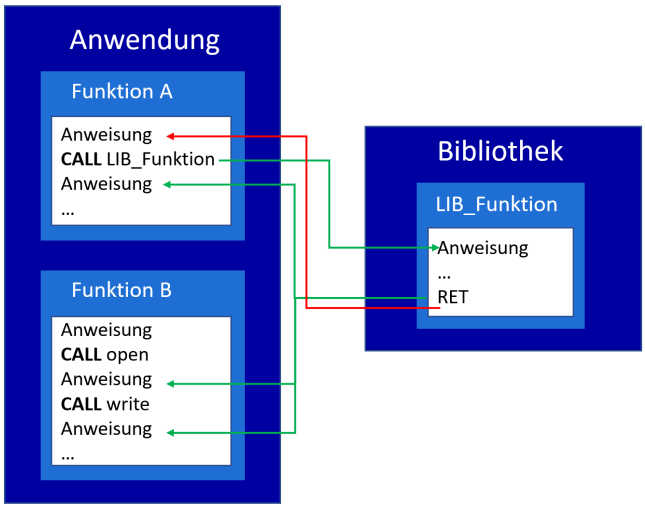

Abbildung 5. Beispiel Call Preceded Return Adress

'Chain of Short Sequences' ist die zweite CFI Policy [2]. Diese Policy verfolgt jenen Hintergedanken: Bei einem ROP-Angriff werden viele Gadgets aneinandergehängt, die nur aus wenigen Anweisungen bestehen. Man hat also eine lange Kette von kurzen Sequenzen, was in einem normalen Programmfluss nicht oft vorkommt. Aus diesem Grund versucht man im Programmfluss solche Ketten von kurzen Sequenzen zu entdecken, um so einen ROP-Angriff zu erkennen.

C. Überblick Coarse Grained CFI Tools

Nachdem nun die Grundlagen von CFI, Coarse Grained CFI und deren Policies nähergebracht wurden, soll dieser Abschnitt einen groben Überblick über die gängigen Coarse Grained CFI Tools und deren Funktionsweisen verschaffen.

Eine der ersten Beobachtungen war, dass es nicht Sinn macht jeden Zweig des Kontrollflussgraphen zu checken, weil das nicht effizient ist und man viel Performance einbüßt. Aus diesem Grund wird ein Zweig nur überprüft, wenn darin eine kritische Funktion oder ein kritischer Syscall aufgerufen wird. Beispiele für solche kritischen Funktionen wären die anfangs erwähnten VirtualProtect und VirtualAlloc, die verwendet werden um Speicher auf ausführbar zu setzten oder ausführbaren Speicher zu allozieren.
Die ersten beiden Coarse Grained CFI Tools, die genauer betrachtet werden sind kBouncer und ROPecker. Sie sind über eine Hook an die kritischen Funktionen und Syscalls angeschlossen. D.h. jedes mal, wenn diese Funktionen aufgerufen werden, starten die beiden Tools ihre Checks [2].
Für die Checks wenden die beiden Tools die gerade erläuterten CFI Policies auf den Verzweigungsinformationen an, um so zu entscheiden, ob ein Angriff stattfindet. Die Verzweigungsinformationen erhalten sie von einem speziellen Register auf der Intel CPU, das sich 'Last Branch Records' nennt. Dieses Register zeichnet die letzten 16 Zweige dieses Threads auf.
ROPecker ist zudem noch an das Paging System angeschlossen [2]. Es lädt zum Beispiel zwei Pages des Anwendungscodes und markiert sie als ausführbar. Der Rest ist nicht ausführbar. Nachdem die zwei Pages überprüft wurden, wird der Rest gecheckt. Der Check für den Rest wird aber erst während der Ausführung der ersten beiden Pages ausgelöst.

ROPGuard ist ein weiters Coarse Grained CFI Tool, welches auch an die kritischen Funktionen und Syscalls angeschlossen ist. ROPGuard verwendet jedoch nicht das Last Branch Records Register [2]. Das Register wird nämlich nicht von jeder CPU zur Verfügung gestellt, was bedeuted, dass kBouncer und ROPecker von der verfügbaren Hardware abhängig sind. ROPGuard dagegen ist unabhänig von der Hardware und benötigt keine zusätzliche Unterstützung, um zu funktionieren. ROPGuard basiert seine Analyse nur auf Heuristiken. Es simuliert das Verhalten des Stack-Pointers und auf dieser Simulation basiert die Entscheidung [2].
Es wurde folgende Behauptung aufgestellt: 'Coarse Grained CFI Maßnahmen reichen aus, um real world und touring complete ROP-Angriffe zu verhindern' [2]. In der Praxis hat sich diese Behauptung jedoch als falsch herausgestellt.
Es ist möglich die Coarse Grained CFI Tools zu umgehen und ROP-Angriffe waren trotz der Coarse Grained CFI Tools immer noch möglich [2].

D. Microsoft Control Flow Guard

Eine weitere CFI-Technologie, die erwähnt werden sollte, ist Control Flow Guard (CFG) von Microsoft. CFG soll Integrität bei indirekten calls/jumps gewährleisten. Es handelt sich um ein Forward Branch Enforcement, das versucht sicherzustellen, dass calls/jumps nur ein valides Ziel erreichen können. Dafür wird eine Bitmap in jeden Prozess geladen. Diese Bitmap gibt an, welche virtuellen Adressen ein valides Ziel für einen indirekten call/jump sind [4]. Vor jedem indirekten call oder jump wird eine Check-Funktion aufgerufen, die diese Bitmap überprüft. Wenn die Adresse, zu der die indirekte Kontrollübertragung stattfinden soll, als valide markiert ist, dann wird der indirekte call/jump ausgeführt. Wenn jedoch die Adresse nicht als valide markiert ist, dann terminiert der Prozess.
Bei Control Flow Guard handelt es sich auch wieder um eine Coarse Grained CFI Lösung. Es wird nämlich nicht überprüft, ob die Funktion die man indirekt aufruft dieselbe Anzahl an Parametern erwartet, oder ob die Parameter vom korrekten Datentypen sind. Es handelt sich um ein Binary, das angibt, ob die Funktion aufgerufen werden kann oder nicht [4]. Control Flow Guard bietet also nicht maximale Sicherheit und es gibt Wege, wie CFG umgangen werden kann.
Das größte Problem von CFG ist jedoch, dass die meisten Angreifer den Stack angreifen. Das heißt niemand umgeht Control Flow Guard, indem er Control Flow Guard angreift [4]. CFG wird umgangen, indem return-Pointer auf dem Stack manipuliert werden, wogegen CFG keine Schutzmaßnahmen hat. Die Integrität der return-Pointer muss sichergestellt werden und dieses Problem wird durch die Control-flow Enforcement Technology von Intel behoben.

IV. CET von Intel

Control-flow Enforcement Technology, kurz CET, ist eine Befehlssatzerweiterung der Intel Prozessoren-Familie. Wichtig zu erwähnen ist, dass CET auf Hardware-Ebene implementiert ist, was die Sicherheit von CET erheblich steigert.

Jedoch hat die Implementierung auf Hardware-Ebene auch seine Schattenseiten. CET erfordert eine spezielle Hardware Unterstützung, um zu funktionieren, und aktuell ist CET nur auf wenigen Intel-Prozessoren verfügbar. Erstmals wurde CET in den Prozessoren der 11. Generation (Tiger Lake) eingeführt [9] und ist seitdem auch in den Prozessoren der neueren Generationen (Alder Lake und Sapphire Rapids) verfügbar.
CET erweitert die Intel-Befehlssatzarchitektur um zwei Elemente. Der Shadow Stack schützt die return-Adressen und verhindert, dass sie manipuliert werden können. Er schützt folglich gegen eine Rückwärtsverkettung von Gadgets mit return-Pointern.
Das Zweite der neuen Elemente ist Indirect Branch Tracking. Dabei handelt es sich um einen Schutz bei indirekter Verzweigung. Es wird also die Vorwärtsverkettung von Gadgets über calls/jumps erschwert. Von der Grundidee ist Indirect Branch Tracking ähnlich zu Control Flow Guard. Beide Technologien legen nämlich valide Ziele für indirekte calls/jumps fest, in der Umsetzung unterscheiden sie sich jedoch.
Neben den Prozessoren muss zusätzlich auch das Betriebssystem CET unterstützen. Unterstützende Betriebssysteme sind Windows 10 (ab Version 2004, jedoch nur Shadow Stack implementiert) [6], Windows 11 [7] und Linux ab Kernel 5.11 (nur Indirect Branch Tracking bis jetzt implementiert, Shadow Stack soll ab Kernel 6.4 implementiert sein) [5]. Wenn eine Unterstützung von Prozessoren- und Betriebssystemseite gewährleistet ist, dann implementiert CET Kontrollfluss-Integrität und schützt vor Kontrollfluss-Subversionsangriffen, wie Code Reuse Attacks [11].

A. Angreifer-Modell und CET-Ziele

Bei der Entwicklung von CET wurden folgende Annahmen getroffen, was die Fähigkeiten des Angreifers betreffen [11]. Der Gegner kann...

- Software Schwachstellen finden, die dem Gegner erlauben überall im virtuellen Speicher zu Lesen und zu Schreiben.
- das komplette Layout des Adressraumes aufdecken und weiß wo z.B. Stacks, Heaps oder Images gemapped sind.
- Lese- und Schreibzugriffe im Speicher beliebig oft wiederholen (Arbitrary Read + Arbitrary Write Schwachstelle).
- Stimuli erzeugen, die den Code andere Pfade nehmen lassen. Dadurch kann der Angreifer den Zustand des Programms und auch den Zustand des Stacks auf diesen Pfaden beobachten.
- Daten an einen Computation Server senden, um sich benötigte Payloads für nachfolgende reads oder writes zu berechnen lassen.
- Kontrollübertragung zu existierenden ausführbaren Code durchführen, um den Zustand von Prozessor-Registern zu verändern.

Der Gegner kann jedoch nicht...

- neuen Code ohne Verifikation hinzufügen und der existierende Code ist read-only und nicht modifizierbar (z.B. write-xor-execute policy).

Basierend auf diesen Annahmen wurden folgende Ziele für CET festgelegt [11]. CET muss...

- einen Schutzmechanismus gegen Code-Reuse und Speichersicherheitfehler für neue architektonischen Elemente, wie neue Hardware-Register und Speicher, zur Verfügung stellen.
- anwendbar sein auf CPU Berechtigungsstufen (user/supervisor).
- anwendbar sein auf CPU Modi die von handelsüblicher Software verwendet werden, wie 32/64-bit, hypervisor, system management mode, enclaves, etc.
- sicherstellen, dass der Kontrollfluss-Schutz auch an Übergangspunkten von Modi- oder Kontextwechseln gewährleistet ist.
- minimale Auswirkungen auf Performance, Speicherverbrauch und Codezuwachs haben.

Des Weiteren wurden noch folgende Constraints definiert die CET einhalten soll [11]:

- Vermeiden von Einbettung programmiersprachenspezifischer Konstrukte in die Instruction Set Architecture.
- Bereitstellen nur notwendiger (minimaler) Fähigkeiten.
- Beibehalten des Stacks/ der Funktionsaufrufe.
- Keine Einschränkung gewöhnlicher Software-Konstrukte (z.B. tail-calls, Co-Routinen, etc).

B. Shadow Stack

Der Shadow Stack ist das erste der neu hinzugefügten Elemente. Der Shadow Stack ist ein zweiter Stack der exklusiv für Kontrollübertragungsoperationen verwendet wird. Er ist separat vom Daten Stack und speichert nur return-Adressen, d.h. es werden keine Daten oder Parameter gehalten [1]. Der Shadow Stack erzwingt, dass returns nur die korrekte Adresse zum Ziel haben können. Es ist also nur der originale Programmfluss möglich, das bedeutet, dass der Shadow Stack eine Fine Grained CFI umsetzt.
Des Weiteren wird der Shadow Stack auf Hardware-Ebene betrieben, wodurch die Sicherheit deutlich erhöht wird, da die CPU für die Verwaltung des Shadow Stacks verantwortlich ist. Um ungewollte Writes auf den Shadow Stack durch Software zu verhindern, ist er schreibgeschützt [11]. Die CPU erzwingt, dass Software nur im Kontext eines calls auf den Shadow Stack schreiben kann. Daraus folgt, dass die Werte die auf dem Shadow Stack liegen nicht durch Software manipuliert werden können.

1) Funktionsweise: Damit der Shadow Stack gewährleisten kann, dass zu der korrekten return-Adresse zurückgekehrt wird, wurden die call Anweisung und die return Anweisung angepasst. Die call Anweisung wurde so modifiziert, dass sie die return-Adresse nicht nur auf den Daten Stack pusht, sondern zusätzlich noch eine Kopie der return-Adresse auf den Shadow Stack pusht [11]. Die return Anweisung wurde so modifiziert, dass sie von beiden Stacks die oberste return-Adresse poppt und die beiden Adressen miteinander vergleicht [11].
Wenn die return-Adresse auf dem Daten Stack nicht manipuliert wurde und gleich zu der Adresse ist, die vom Shadow Stack gepoppt wurde, dann wird das 'OKAY' gegeben und der return kann durchgeführt werden. Falls jedoch die return-Adresse auf dem Daten Stack manipuliert wurde, dann schlägt der Vergleich der beiden return-Adressen fehl und es wird eine Control Protection (#CP) Exception geworfen [8]. Dabei handelt es sich um eine neue Fault Type Exception die von CET eingeführt wurde [11]. Sie informiert priveligiertere Software, dass eine Kontrollflussverletzung stattgefunden hat.
Für das Management des Shadow Stacks wurden ein paar neue Anweisungen eingeführt [11], die nun genauer erläutert werden.
Mit der Anweisung *INCSSP* kann man den Shadow Stack Pointer erhöhen. *RDSSP* erlaubt es den Shadow Stack Pointer zu lesen. Es ist auch möglich den vorherigen Shadow Stack Pointer zu speichern und diesen gespeicherten Shadow Stack Pointer wiederherzustellen. Dies funktioniert mit den beiden Anweisungen *SAVEPREVSSP* und *RSTORSSP*, welche für den Shadow Stack Switch benötigt werden. Mit *WRSS/WRUSS* kann man auf den Shadow Stack schreiben. Es gibt auch eine Busy-Flag, die man mit den Anweisungen *SETSSBSY/CLRSSBSY* entweder setzen oder löschen kann. Mit dieser Busy-Flag hat man die Möglichkeit einen Shadow Stack zu Aktivieren oder zu Deaktivieren.

2) Shadow Stack Switch: Im Folgenden soll nun auf den Shadow Stack Switch genauer eingegangen werden. Man muss nämlich in der Lage sein zwischen unterschiedlichen Shadow Stacks hin- und herspringen zu können, weil unterschiedliche Prozesse und Threads gleichzeitig am laufen sind [8]. Jeder dieser Prozesse und Threads besitzt nämlich einen eigenen Shadow Stack. Wenn der Scheduler nun einen neuen Thread scheduled, dann wird vom Shadow Stack des aktuellen Threads zum Shadow Stack des nächsten Threads geswitched.
Für den Switch werden, wie schon erwähnt, die beiden Anweisungen SAVEPREVSSP und RSTORSSP bereitgestellt, um den Switch auf eine kontrollierte Weise zu vollziehen. Wenn der Scheduler von einem aktiven Shadow Stack wegswitcht und später zu diesem Shadow Stack zurückkehrt, muss der Shadow Stack Pointer (SSP) wieder genau der gleiche sein, wie zu dem Zeitpunkt als von dem Shadow Stack weggeswitcht wurde [11]. Der SSP muss also wieder an dieselbe Adresse auf dem Shadow Stack zeigen, damit die Sicherheit des Shadow Stacks gesichert ist.
Der Switch ist ein zweistufiger Prozess: Zuerst wird *RSTORSSP* ausgeführt, um den neuen Shadow Stack zu verifizieren und auf ihn zu wechseln. Damit ein Wiederherstellungspunkt, ein sogenanntes *shadow stack restore token* auf dem alten Shadow Stack gespeichert wird, wird *SAVEPREVSSP* ausgeführt.

In Abbildung 6 sind zwei Shadow Stacks zu sehen und das Ziel ist es vom linken Shadow Stack (aktiv) zu dem rechten Shadow Stack zu wechseln.

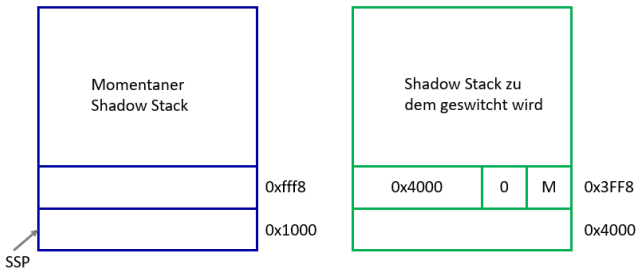

Abbildung 6. Shadow Stacks vor Switch

Der Shadow Stack Pointer zeigt aktuell auf die Adresse 0x1000 des aktiven Shadow Stacks. Auf dem Shadow Stack, zu dem geswicht werden soll, liegt ein ein *shadow stack restore token* an der Adresse 0x3FF8. Das heißt der rechte Shadow Stack war schon einmal aktiv, es wurde jedoch von ihm weggewechselt und nun soll auf ihn zurückgekehrt werden. Das *shadow stack restore token* ist ein 64-Bit Wert und ist folgendermaßen aufgebaut:

- Bit 63:2 - 4-byte aligned SSP, für den der Wiederherstellungspunkt erstellt wurde. Die Adresse gibt an wo sich der SSP befunden hat, als von dem Shadow Stack weggeswitcht wurde. Dieser SSP muss sich an einer Adresse befinden die sich 8 oder 12 bytes über der Adresse des *shadow stack restore tokens* befindet. Diese Eigenschaft wird in der *RSTORSSP* Anweisung überprüft [11].
- Bit 1 - reserviert. Muss 0 sein [11].
- Bit 0 - auch Modus Bit genannt. Wenn das Modus Bit 0 ist, dann kann das *shadow stack restore token* von einer *RSTORSSP* Anweisung im 32 bit Modus verwendet werden. Wenn es 1 ist kann das *shadow stack restore token* von einer *RSTORSSP* Anweisung im 64 bit Modus verwendet werden [11].

Das Token gibt also an, dass der SSP an der Adresse 0x4000 wiederhergestellt werden soll. Die Anweisung *RSTORSSP* wird mit dem Argument 0x3FF8 aufgerufen, da die Anweisung die Adresse eines *shadow stack restore token* benötigt. Als erstes gleicht *RSTORSSP* den Modus der Maschine mit dem gesetzten Modus Bit ab. Danach wird gecheckt, ob das reservierte Bit an Position 1 0 ist und ob die im Token gespeicherte Adresse, hier 0x4000, 8 oder 12 bytes von der Adresse des Tokens entfernt ist. Wenn alle Checks erfolgreich waren, wird der SSP nun auf 0x3FF8 gesetzt.

Zudem wird das *shadow stack restore token* wird durch ein *previous SSP token* ersetzt. Dieses *previous SSP token* hat folgende Form:

- Bit 63:2 - vorheriger SSP, der auf die Spitze des alten Shadow Stacks zeigt. Es handelt sich also um den SSP, der aktiv war, als RSTORSSP aufgerufen wurde [11].
- Bit 1 - ist auf 1 gesetzt und zeigt, dass es sich um ein *previous SSP token* handelt [11].
- Bit 0 - auch Modus Bit genannt. Wenn das Modus Bit 0 ist, dann kann das *previous SSP toke* von einer *SAVEPREVSSP* Anweisung im 32 bit Modus verwendet werden. Wenn es 1 ist kann das *previous SSP token* von einer *SAVEPREVSSP Anweisung* im 64 bit Modus verwendet werden [11].

Nach dem Wechsel zum neuen Shadow Stack kann nun ein Wiederherstellungspunkt mit *SAVEPREVSSP* auf dem alten Shadow Stack erstellt werden. *SAVEPREVSSP* verwendet das *previous SSP token*, das von RSTORSSP erzeugt wurde, um das *shadow stack restore token* zu erstellen. Dazu benötigt *SAVEPREVSSP* keinen Operanden, es konsumiert das *previous SSP token*, das oben auf dem neuen Shadow Stack liegt.
SAVEPREVSSP verifiziert zuerst die Adresse, die im *previous SSP token* gespeichert ist. Danach werden die 8 bytes des *previous SSP token* vom neuen Shadow Stack gepoppt. Anschließend wird wieder überprüft, ob das Bit an Position 1 1 ist und ob das Modus Bit mit dem Modus der Maschine übereinstimmt. Wenn alles gepasst hat wird ein *shadow stack restore token* auf den alten Shadow Stack gepusht. Nach diesen Schritten wurde erfolgreich von einem Shadow Stack zu einem anderen Shadow Stack geswitched [11]. Den Zustand der Shadow Stacks nach diesen Schritten kann man in Abbildung 7 sehen.

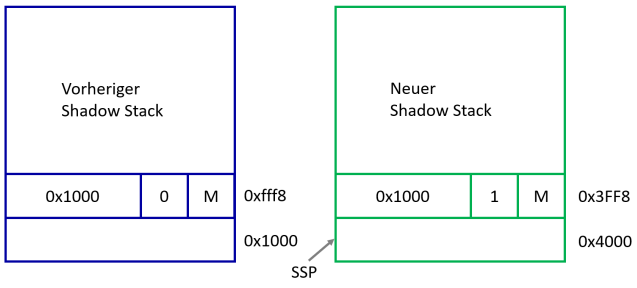

Abbildung 7. Shadow Stacks nach Switch

Falls man keinen Wiederherstellungspunkt auf dem alten Shadow Stack benötigt, kann man auch das *previous SSP token* mit der Anweisung *INCSSP* vom neuen Shadow Stack poppen.

C. Indirect Branch Tracking

Das zweite neue Element ist Indirect Branch Tracking. Es führt die neue Anweisung ENDBR ein. ENDBR markiert valide Codeziele für indirekt calls oder jumps [11]. Das bedeutet, wenn ein indirekter call/jump eine andere Anweisung als ENDBR zum Ziel hat wird eine #CP Exception geworfen. So werden Versuche den Kontrollfluss auf ungewollte Ziele zu lenken verhindert und gemeldet.
Beim Indirect Branch Tracking handelt es sich um eine Coarse Grained CFI Lösung, da der indirekte Zweig nicht die korrekte und sichere Adresse zum Ziel haben muss. Es wird jedoch verhindert, dass in die Mitte von Funktionen gesprungen werden kann. Die Opcodes der neuen Anweisung wurden so gewählt, dass sie sich wie eine NOP Anweisung auf Intel 64 Prozessoren, die CET nicht unterstützen, verhält [11]. Auf Prozessoren die CET unterstützen verhält sich ENDBR fast wie eine NOP Anweisung, weil sie keine zusätzlichen Register belastet, sie verändert auch nicht den Zustand des Programms und sie hat nur minimale Auswirkungen auf die Performance des Programms [11].
Indirect Branch Tracking implementiert zwei identische State Machines. Eine ist für den user mode und die andere für den supervisor mode [11]. Der Aufbau der State Machine ist in Abbildung 8 zu sehen. Der Startzustand der State Machine ist der IDLE Zustand. Wenn eine andere Anweisung, als ein indirekter call/jump auftritt, bleibt die State Machine im IDLE Zustand. Nur bei einer indirekten Verzweigung (call/jump) geht die State Machine in den Zustand WAIT_FOR_ENDBRANCH über.

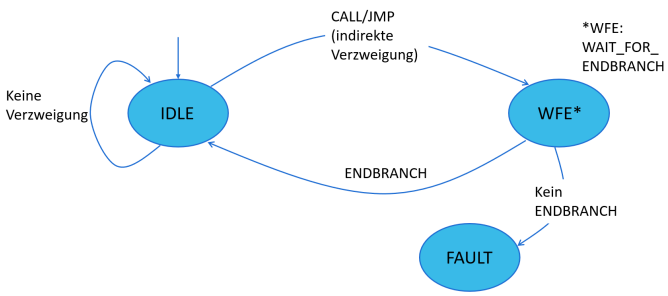

Abbildung 8. Indirect Branch Tracking State Machine

In diesem Zustand wird die State Machine eine #CP Exception mit dem Fehlercode 'ENDBRANCH' werfen, wenn die nächste Anweisung kein ENDBR ist. Für den Fall, dass die nächste Anweisung der richtige ENDBR ist, wird wieder in den IDLE Zustand übergegangen.

D. Ergebnisse

Nun wird zum einen die Sicherheit von CET und zum anderen die Performance von CET evaluiert.

1) Sicherheit: Ein Weg, um die Sicherheit von CET zu evaluieren, ist es die Average indirect target Reduction, kurz AIR, zu berechnen. Dabei handelt es sich um eine Metrik, mit der man die Stärke der Control-flow Integrity berechnen kann [11]. Sie repräsentiert ein Set von Adressen, die über indirekte Kontrollübertragungen erreicht werden können. Die AIR Metrik wird mit folgender Formel beschrieben [11]:

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

Hier ist n die Gesamtzahl der indirekten Verzweigungsstellen im Programm. S stellt die Menge der Programmadressen dar, an die alle indirekten Verzweigungsstellen den Kontrollfluss ohne CFI-Schutz leiten können. Und T_i repräsentiert die Menge der Programmadressen, zu denen die i-te indirekte Verzweigungsstelle den Kontrollfluss mit CFI-Schutzmaßnahmen lenken kann [11].
Ein niedriger AIR Wert bedeutet, dass man ein großes Set an Adressen über indirekte Kontrollübertragungen erreichen kann. Die Kontrollfluss Integrität ist in der Folge niedrig. Ein hoher AIR Wert dagegen steht für eine hohe Kontrollfluss Integrität, da nur ein kleines Set an Adressen mit indirekten Kontrollübertragungen erreicht werden kann.
Auf der x86 Architektur können indirekte Kontrollübertragungen jedes Byte im Programm erreichen. Daraus folgt, dass S die Größe des Programm-Codes ist [11]. Folglich ist der AIR Wert sehr niedrig.
Mit aktivierten CET kann eine return Anweisung genau ein Ziel im Programm erreichen, nämlich die return-Adresse, die oben auf dem Shadow Stack liegt. Indirekte calls/jumps können nur einen ENDBR zum Ziel haben.

Für den SPEC CPU 2006 C/C++ Benchmark wurde eine durchschnittliche AIR-Metrik von 99,8% berechnet [11]. Für einzelne Programme waren Werte zwischen 99,3% und 99,99% vertreten.

Des Weiteren wurde das Linux Kernel Binary nach verfügbaren Gadgets durchsucht [11]. Wenn sich in einem Binary viele Gadgets befinden, hat ein Angreifer es leichter einen ROP-Angriff durchzuführen. Aus diesem Grund gibt die Anzahl der gefundenen Gadgets in einem Binary einen guten Richtwert für die Sicherheit des Binarys.
Der Linux Kernel wurde mit dem ROPGadget Tool, nach den verfügbaren Gadgets durchsucht. Das verwendete Linux Binary war eine default Konfiguration mit einer Größe von 25 MB. Das ROPGadget Tool wurde eingeschränkt, Gadgets bis zu einer Größe von maximal 10 bytes zu suchen. Unter diesen Voraussetzung wurden 197241 Gadgets gefunden - man kann jedoch davon ausgehen, dass sich noch mehr Gadgets in dem Binary befinden, da die Größe der Gadgets limitiert wurde. Diese große Anzahl an Gadgets zeigt die Größe der vorhandenen Angriffsoberfläche des Binarys.
In Abbildung 9 kann man die Verteilung der verschiedenen Gadget Größen nachvollziehen. Es wurden z.B 18000 Gadgets mit einer Größe von 2 bytes, 20000 Gadgets mit einer Größe von 4 bytes und 25000 mit einer Größe von 9 bytes gefunden.

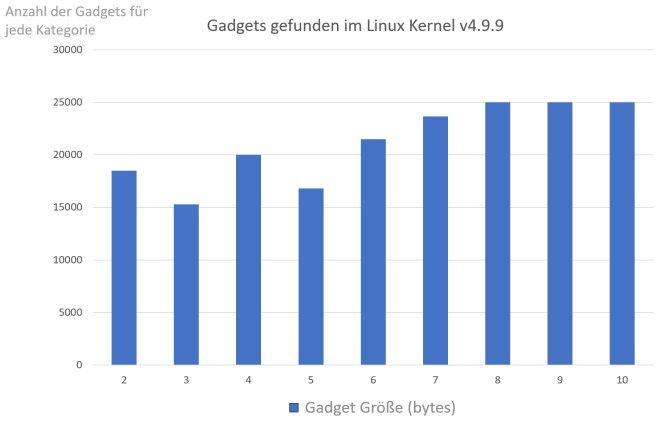

Abbildung 9. Gefunde Gadgets im Linux Kernel v4.9.9 [11]

Im Kontrast dazu ist der Angreifer bei einem Binary, das CET unterstützt, eingeschränkt Funktionen zu verwenden, die einen ENDBRANCH haben und zur letzten Adresse auf dem Shadow Stack zurückkehren [11]. Im analysierten Linux Kernel Binary wurden 18412 solcher Funktionen gefunden. Diese Funktionen können auch nicht mehr mit einem schädlichen return verkettet werden, sondern müssen mit einem indirekten call/jump aneinandergehängt werden. Die Funktionen hatten eine durchschnittliche Größe von 214 bytes. Durch diese Größe entstehen größere Nebeneffekten, was zu einem Anstieg der Komplexität bei der Durchführung von COP-Angriffen führt [11].
In Abbildung 10 wird aufgezeigt, wie sehr die Anzahl der Gadgets reduziert wurde, die für einen code reuse attack verwendet werden können. Zusätzlich hat sich auch die Verkettung der noch vorhandenen Gadgets deutlich erschwert. Durch die Eliminierung des Großteils der unerwünschten Gadgets hat sich die Angriffsoberfläche des Linux Kernel um ein Vielfaches verringert. Es kann für eine code reuse attack nur noch eine kleine Anzahl an Funktionen verwendet werden. Die kleinere Angriffsoberfläche kann systematisch analysiert werden, um nicht benötigte Cases zu elemnieren oder um unsicher Konstrukte durch Redesign oder gezielte Checks zu adressieren.
Die Sicherheit des Linux Kernel Binarys hat sich also durch die Aktivierung von CET erheblich erhöht.

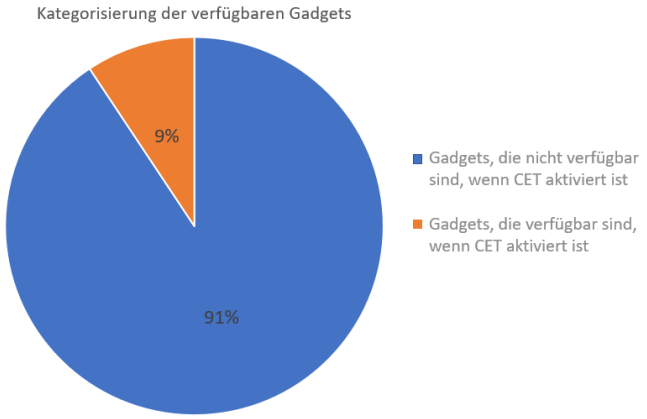

Abbildung 10. Kategorisierung der verfügbaren Gadgets

2) Performance: Besonders interessant ist die Performance von CET und im speziellen die Performance des Shadow Stacks. Der Shadow Stack implementiert eine Fine Grained CFI und diese ist nur mit einem Performance Overhead umsetzbar. Die Performance des Shadow Stacks wurde anhand einer Reihe von Microprozessoren Benchmarks und Anwendungs Traces, die auf einem zyklusgenauen Prozessor Performance-Modell ausgeführt wurden, evaluiert [11].

Die call Anweisung wurde geupdated einen zusätzlichen Push auf den Shadow Stack zu machen und die return Anweisung wurde geupdated die return-Adresse vom Shadow Stack zu poppen und mit der Adresse vom Daten-Stack zu vergleichen. Der geometrische Mittelwert des instruction-per-cycle (IPC) Verlusts lag über die Workload-Traces bei ungefähr 1,65% [11]. Der IPC Verlust lag zwischen Werten von 0,08% bis 2,71%.

Der Einfluss von Indirect Branch Tracking auf die Performance wurde evaluiert durch das Kompilieren von C/C++ Programmen von der SPEC CPU 2006 C/C++ unter Verwendung eines ICC Kompilers, der CET unterstützt. Da sich die ENDBRANCH Anweisung auf Prozessoren - mit oder ohne CET Unterstützung - wie eine NOP Anweisung verhält, wurde kein spürbarer Slowdown im Durchschnitt gemessen [11].

E. Schwachstellen

Jedoch hat CET auch Schwachstellen und kann folglich auch umgangen werden. Darunter Fallen Code Replacement Attacks, Counterfeit Object-Oriented Programming (COOP) und Data-only corruption, um ein paar zu nennen [8]. In diesem Abschnitt wird sich auf Code Replacement Attacks fokusiert, da es sich dabei um eine sehr kreative Lösung handelt den Shadow Stack zu umgehen.
Das Prinzip wird anhand vom nachfolgenden Beispiel erklärt [4]. In der Abbildung 11 ist ein kleiner call-Stack zu sehen. Auf dem call-Stack liegt FOO.dll. Foo.dll hat eine Funktion namens StartWork, welche eine weitere Funktion namens DoWork, die in einem zweiten Dll liegt, aufruft. Es wird angenommen, dass DoWork sich gerade in einer Schleife befindet und etwas berechnet.

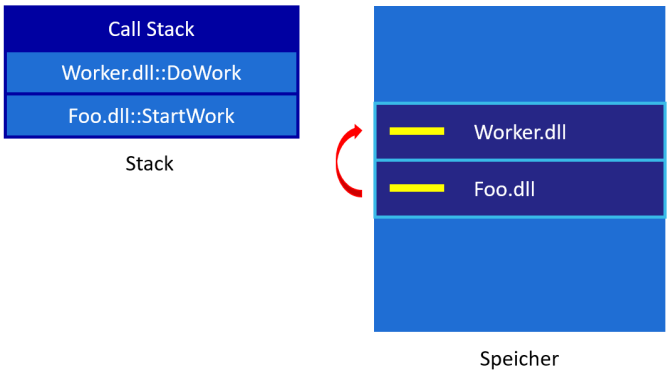

Abbildung 11. Beispiel Code Replacement Attack

In dieser Zeit erzwingt der Angreifer FOO.dll durch z.B. Data Corrpution zu entladen. An genau die Stelle, an der sich FOO.dll befunden hat, lädt der Angreifer nun etwas völlig anderes, z.B. ein weiteres Dll, welches er sich ausgesucht hat. Das was der Angreifer geladen hat befindet sich also an genau der selben virtuellen Adresse, an der FOO.dll gemapped war.
Wenn nun DoWork zurückkehrt, verwendet es die return-Adresse, die auf dem Shadow Stacks gespeichert ist. Der Pointer zeigt immer noch an die virtuelle Adresse, an der FOO.dll geladen war. Jedoch befindet sich nun dort etwas völlig anderes und der return-Pointer ist inzwischen ein dangling Pointer. Es wird also in das vom Angreifer geladen Dll returnt, was der Ausgangspunkt für einen ROP-Angriff sein könnte.

V. Zusammenfassung

Wie gerade gezeigt wurde hat auch CET seine Schwachstellen und ist nicht komplett sicher. CET ist zwar ein sehr wichtiger Schritt, um Kontrollfluss Integrität durchzusetzen und die Sicherheit von Computern zu gewährleisten, jedoch reicht CET alleine nicht aus.

Man sollte sich Kontrollfluss Integrität und auch Sicherheit eines Computerprogramm eher als eine Burg vorstellen, die aus vielen Mauerstücken besteht. Die Mauerstücke arbeiten zusammen und unterstützen sich gegenseitig um das gemeinsame Ziel 'Sicherheit' zu gewährleisten. Eines dieser Mauerstücke ist CET, es ist auch ein sehr großes und wichtiges Mauerstück und sorgt für eine hohe Kontrollfluss Integrität. Aber auch CET ist umgehbar, wenn die anderen Mauerstücke nicht vorhanden wären.

Deswegen ist eine Kombination von verschiedenen Technologien der einzige Weg, wie die Sicherheit eines Computerprogramm maximiert werden kann. Ein anderes Mauerstück wäre beispielsweise Data Execution Prevention (DEP) - was sogar eine Annahme von CET ist - um die Injection von bösartigen Code zu verhindern. Auch PAC (Pointer Authentication Codes), um die Integrität von Pointer zu gewährleisten, ist ein wichtiger Bestandteil der Burg.

Außerdem kann CET seine Wirkung gegen ROP-Angriffe erst richtig entfalten, wenn es großflächig auf dem Markt eingeführt wurde, was noch ein wenig Zeit in Anspruch nehmen wird.

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).