Ansätze zur automatisierten Speicherfehlererkennung

FLORIAN DINTER, Universität Siegen, Deutschland

Speicherzugriffsfehler, wie beispielsweise Out-of-Bounds-Zugriffe und die Nutzung bereits freigegebenen Speichers, stellen nach wie vor ein gravierendes Problem für Programmiersprachen wie C und C++ dar. Dieses Paper fasst zwei bedeutende Ansätze zur Erkennung von Speicherfehlern zusammen und vergleicht deren Wirksamkeit und Leistung: den neu entwickelten „mcds“und den etablierten AddressSanitizer.

CCS Concepts: • **Security and privacy** → Software security engineering.

1 EINLEITUNG

Die Erkennung von Speicherzugriffsfehlern ist ein entscheidender Aspekt der Softwareentwicklung, insbesondere für Programme, die in den unsicheren Sprachen C und C++ geschrieben sind. Speicherzugriffsfehler wie Out-of-Bounds-Zugriffe und die Nutzung bereits freigegebenen Speichers (Use-After-Free) können schwerwiegende Folgen haben, darunter Programmabstürze, Datenkorruption und Sicherheitslücken, die von Angreifenden ausgenutzt werden können. Die Motivation zur Entwicklung effektiver Werkzeuge zur Fehlererkennung ergibt sich aus der Notwendigkeit, diese Probleme frühzeitig zu identifizieren und zu beheben, um die Zuverlässigkeit und Sicherheit von Software zu gewährleisten.

Die Bedeutung der Fehlererkennung von Speicherzugriffen ist weitreichend. Speicherzugriffsfehler gehören zu den häufigsten und zugleich gefährlichsten Fehlern in Software. Sie können zu unvorhersehbarem Verhalten führen, das schwer zu diagnostizieren und zu beheben ist. Insbesondere in sicherheitskritischen Anwendungen, wie z.B. Betriebssystemen, Webbrowsern und Server-Software, ist die präzise Erkennung und Beseitigung solcher Fehler unerlässlich, um Angriffe zu verhindern und die Integrität der Systeme zu bewahren.

Die Relevanz dieser Thematik ist besonders für die Programmiersprachen C und C++ von hoher Bedeutung. Diese Sprachen bieten aufgrund ihrer Leistungsfähigkeit und Flexibilität direkten Zugriff auf Speicher, was für viele Anwendungen vorteilhaft ist. Gleichzeitig erhöht dieser direkte Zugriff jedoch das Risiko von Speicherfehlern, da die Programmierenden selbst für die Speicherverwaltung verantwortlich sind. Im Gegensatz zu Sprachen mit automatischer Speicherverwaltung, wie Java oder Python, bieten C und C++ keine eingebauten Mechanismen zur Vermeidung von Speicherzugriffsfehlern. Daher ist die Entwicklung leistungsfähiger Werkzeuge zur Erkennung und Behebung solcher Fehler in C und C++ besonders wichtig.

Die vorliegende Arbeit untersucht zwei bedeutende Werkzeuge zur Erkennung von Speicherzugriffsfehlern: den neu entwickelten „mcds“ und den etablierten AddressSanitizer. Beide Ansätze zielen darauf ab, Speicherzugriffsfehler in Programmen zu identifizieren und zu beheben, um die Zuverlässigkeit und Sicherheit der Software zu erhöhen.

Author's address: Florian Dinter, florian.dinter@student.uni-siegen.de, Universität Siegen, Siegen, Nordrhein Westfalen, Deutschland.

2 GRUNDLAGEN

Speicherfehler sind eine der häufigsten und schwerwiegendsten Klassen von Fehlern in Software, die in den Programmiersprachen C und C++ entwickelt wird. Diese Fehler entstehen oft durch unsachgemäßen Umgang mit Speicher, insbesondere aufgrund der direkten Kontrolle, die diese Sprachen über Speicherzuweisungen und -freigaben bieten. Solche Fehler können zu unvorhersehbarem Verhalten, Programmabstürzen, Datenkorruption und Sicherheitslücken führen, die potenziell von Angreifenden ausgenutzt werden können.

2.1 Arten von Speicherfehlern

In diesem Abschnitt erläutern wir verschiedene Arten von Speicherzugriffen, die häufig auftreten. Grundsätzlich lässt sich zwischen „Spatial Memory Corruption“ und „Temporal Memory Corruption“ unterscheiden. Spatial Memory Corruption tritt auf, wenn versucht wird, auf eine Speicheradresse zuzugreifen, die außerhalb der zugewiesenen Speichergrenzen liegt. Temporal Memory Corruption tritt auf, wenn versucht wird, auf eine Speicheradresse zuzugreifen, die bereits freigegeben wurde oder deren Speicherobjekt nicht mehr gültig ist [7]. Ein häufiges Problem ist der Out-of-Bounds Access, bei dem ein Programm auf Speicher zugreift, der außerhalb des zugewiesenen Bereichs eines Arrays oder eines anderen Speicherobjekts liegt. Ein Out-Of-Bounds Zugriff kann zu unerwartetem Verhalten führen, da der Speicherbereich außerhalb der Grenzen möglicherweise Daten enthält, die für andere Teile des Programms oder des Betriebssystems relevant sind. Listing 1 zeigt einen solchen Out-Of-Bounds Access. Beim Zugriff auf den siebten Index des Arrays wird Speicher ausgelesen, der nicht im Wertebereich des Arrays liegt. Solche Speicherzugiffe bringen somit ein unvorhersehbares Verhalten mit sich.

```
1 int arr[5] = {1, 2, 3, 4, 5};
2 std::cout << arr[7] << std::endl;
```

Listing 1. Out-Of-Bounds Access

Ein weiteres häufiges Problem ist der Use-After-Free-Fehler, der in Listing 2 veranschaulicht wird. In Listing 2 wird ein Pointer erstellt, der auf einen Integer im Speicher zeigt. Dieser Integer wird auf der Konsole ausgegeben und anschließend gelöscht. Bei der wiederholten Ausgabe auf der Konsole sind die Folgen unvorhersehbar, da der Speicher direkt von anderen Teilen des Programmes oder des Betriebssystems verwendet werden kann. Use-After-Free-Fehler entstehen, wenn ein Programm auf einen Speicherbereich zugreift, der bereits freigegeben wurde. Nachdem der Speicher freigegeben wurde, kann er vom Betriebssystem oder von anderen Teilen des Programms für andere Zwecke wiederverwendet werden, was zu Datenkorruption oder Programmabstürzen führen kann.

```
1 int *ptr = new int(10);
2 std::cout << *ptr << std::endl;
3 delete ptr;
```

```
4 std::cout << *ptr << std::endl;
```

Listing 2. Use-After-Free

Analog zum Use-After-Free-Fehler gibt es den Write-After-Free-Fehler. Listing 3 zeigt, wie der Speicherbereich für einen Integer allokiert wird. Anschließend wird der Speicherbereich durch die `free`-Anweisung wieder freigegeben. Nach der Speicherfreigabe wird versucht, einen Wert zu schreiben. Write-After-Free-Fehler entstehen beim Versuch, einer Variable einen Wert zuzuweisen, deren Speicherbereich schon freiegeben wurde. Write-After-Free-Fehler können zu undefiniertem Verhalten führen, da der Speicher möglicherweise anderweitig durch das Programm oder Betriebssystem verwendet wird.

```
1 int* ptr = (int*)std::malloc(sizeof(int));
2 std::free(ptr);
3 *ptr = 42;
```

Listing 3. Write-After-Free

Ein weiteres Problem sind Speicherlecks, die entstehen, wenn ein Programm Speicher zuweist, diesen jedoch nicht wieder freigibt, nachdem er nicht mehr benötigt wird. Dies kann im Laufe der Zeit zu einem übermäßigen Speicherverbrauch führen und letztendlich dazu, dass dem System der Speicher ausgeht.

Die Auswirkungen von Speicherfehlern können vielfältig und schwerwiegend sein. Zu den häufigsten Folgen gehören Programmabstürze, die oft mit einem `Segmentation Fault` (Speicherzugriffsverletzung) einhergehen, sowie Datenkorruption, die die Datenintegrität beeinträchtigen kann, indem sie unbeabsichtigt Daten überschreibt, was zu unvorhersehbarem Verhalten und falschen Ergebnissen führt. Besonders gravierend sind die Sicherheitslücken, die durch Speicherfehler entstehen können. Angreifende können diese Fehler ausnutzen, um bösartigen Code auszuführen oder vertrauliche Daten zu stehlen. Buffer Overflows und Use-After-Free-Fehler sind häufige Angriffsvektoren für Exploits.

2.2 Techniken zur Speicherfehlererkennung

Aus der Notwendigkeit heraus, Speicherfehler zu erkennen und verhindern, wurden Techniken entwickelt die diese Aufgabe übernehmen und die Programmierenden bei ihrer Arbeit unterstützen. In diesem Kapitel erläutern wir Techniken, die auch in AddressSanitizer und „mcds“ Anwendung finden.

2.2.1 Shadow Memory. Beim Shadow Memory werden Metadaten für jede Speichereinheit der Anwendung gespeichert. Dabei wird eine Speicheradresse der Anwendung auf eine Adresse im *Shadow Memory* abgebildet. Die Abbildung passiert je nach Implementierung entweder durch eine direkte Skalierung und Offset oder durch zusätzliche Übersetzungsebenen, die Gebrauch von einer Lookup-Tabelle machen.

Die Verwendung von *Shadow Memory* resultiert in einen höheren Speicherverbrauch der Anwendung beziehungsweise einer Verkleinerung des nutzbaren Speichers innerhalb der Anwendung. TainTrace verwendet eine direkte Abbildung des *Shadow Memories* und benötigt ein *Shadow Memory* mit derselben Größe wie der des Anwendungsspeichers. Dies kann zu Problemen bei Anwendungen führen, die nicht mit dem halben Speicher auskommen können [3]. Im Gegensatz dazu ist der Speicherbedarf des *Shadow Memories* von LIFT nur ein Achtel des Anwendungspeicherraums und ist damit effizienter als TainTrace [13].

Bei der Nutzung einer mehrstufigen Übersetzung des Anwendungsspeichers in den *Shadow Memory* werden zusätzliche Speicherzugriffe benötigt. Durch die Verwendung von Lookup-Tabellen wird eine höhere Flexibilität des Addresslayouts der Anwendung erreicht.

2.2.2 Binäre Instrumentierung. Binäre Instrumentierung ist eine Technik, die verwendet wird, um zusätzliche Anweisungen in bereits kompilierte Programme (*Binaries*) einzufügen [9]. Das Laufzeitverhalten eines Programms kann so analysiert und modifiziert werden, ohne dass der Quellcode des Programms geändert werden muss. So kann auch proprietäre und closed-source Software instrumentiert werden. Dennoch kann binäre Instrumentierung zu einem Laufzeit-Overhead führen, da zusätzliche Anweisung ausgeführt werden müssen. Zusätzlich kann binäre Instrumentierung aufgrund der hohen Komplexität fehleranfällig sein.

Man unterscheidet zwischen zwei Arten der binären Instrumentierung. Bei der statischen binären Instrumentierung wird der Binärcode eines Programms vor seiner Ausführung modifiziert. Das Werkzeug analysiert also den binären Programmcode, fügt Instrumentierungscode ein und speichert das modifizierte Programm ab. Bei der dynamischen Instrumentierung fügt das Werkzeug den Instrumentierungscode zur Laufzeit ein. Das Einfügen zur Laufzeit ermöglicht es, den Instrumentierungscode abhängig vom aktuellen Zustand und Verhalten des Programms einzufügen.

Tools zu Speicherfehlererkennung nutzen binäre Instrumentierung, um eine Vielfalt an Fehlern zu erkennen. Valgrind [10], Dr. Memory [2], Purify [4], BoundsChecker [8], Intel Parallel Inspector [6] und Discover [12] sind bekannte Tools, die binäre Instrumentierung nutzen und Out-Of-Bounds, sowie Use After Free Fehler ohne Fehlalarme (falsch positive) erkennen. Während diese Tools keine Out-Of-Bounds-Fehler im Stack erkennen können, erkennen sie aber uninitialisierte Lesezugriffe.

2.2.3 Erweiterte Speicherallokatoren. Debug Allocators sind spezialisierte Speicherallokatoren, die verwendet werden, um Speicherfehler wie Out-of-Bounds-Zugriffe und Use-After-Free Fehler zu erkennen. Diese Techniken ändern nicht die Programmausführung selbst, sondern arbeiten durch modifizierte Speicherzuweisungs- und Freigaberoutinen.

Werkzeuge, wie zum Beispiel GuardMalloc [5] nutzen einen Speicherseiten basierten Schutz, um Speicherfehler zu erkennen. Eine Speicherseite ist eine festgelegte Größe eines zusammenhängenden Blocks im virtuellen Speicher, der sowohl im physischen RAM als auch im virtuellen Adressraum des Prozesses verwendet wird. Speicherseiten sind ein zentraler Bestandteil der Speicherverwaltung und des Paging-Mechanismus, der in modernen Betriebssystemen verwendet wird [15].

Beim Speicherseitenschutz wird jeder Allokation von Speicher eine eigene Seite im virtuellen Speicher zugewiesen. Zusätzliche Seiten rechts und links werden als nicht-zugreifbar markiert. Wenn ein Programm versucht, auf diese geschützten Seiten zuzugreifen – zum Beispiel bei einer Out-Of-Bounds Operation –, löst dies einen Out-Of-Bounds Fehler aus. Der Mechanismus zum Speicherseitenschutz

kann allerdings zu einem hohen Speicherverbrauch führen und dazu noch langsam sein, was bei `malloc`-intensiven Programmen auffällt, da jeder `malloc`-Aufruf mindestens einen Systemaufruf erfordert.

Eine weitere Möglichkeit des Speicherseitenschutzes ist es, Redzones um die Speicherregionen hinzuzufügen und mit „magischen" Werten zu füllen. Magische Werte bestehen aus Konstanten, die leicht wiedererkannt werden können, aber im normalen Programmfluss nicht vorkommen. Der Wiedererkennungswert der magischen Werte dient dazu, verschiedene Bereiche mit magischen Werten markieren zu können. Die `free` Anweisung schreibt ebenfalls magische Werte in die betreffende Speicherregion. Wenn ein „magischer" Wert gelesen wird, hat das Programm einen Out-Of-Bounds Fehler erzeugt oder auf nicht initialisierte Werte zugegriffen. Wird ein magischer Wert in einer Redzone überschrieben, wird dies erst später erkannt, wenn die Redzone bei der Speicherfreigabe überprüft wird. Das Werkzeug weiß jedoch nicht genau, wann der Out-Of-Bounds Schreibvorgang oder Write-After-Free Fehler aufgetreten ist. Werkzeuge wie DieHarder [11] machen sich diese Technik zu Nutze.

2.2.4 Memory Delay. Memory Delay ist eine Technik, die die Wiederverwendung von freigegebenem Speicher verzögert, um temporal memory corruption wie Use-After-Free zu erkennen. Memory Delay funktioniert, indem es Speicher, der durch eine free-Operation freigegeben wurde, für eine gewisse Zeit als unzugänglich markiert und erst nach einer Verzögerung wieder zur Allokation verfügbar macht. Dies erhöht die Wahrscheinlichkeit, dass ein Programm, das auf bereits freigegebenen Speicher zugreift, einen Fehler erzeugt, der vom Detektionstool erkannt werden kann. Weiterführende Informationen und die Funktionsweise in AddressSanitizer können in Abschnitt 3.3 gefunden werden.

3 TECHNISCHE ANALYSE: ADDRESSSANITIZER

Konstantin Serebryani, Derek Bruening, Alexander Potapenko und Dmitry Vyukov stellen in ihrem Paper *AddressSanitizer: A Fast Address Sanity Checker* [14] den AddressSanitizer vor. Der AddressSanitizer ist ein Tool zur Speicherfehlererkennung. Dabei arbeitet AddressSanitizer mit einem *Shadow Memory*, um zu prüfen, ob der Speicherzugriff sicher ist und binärer Instrumentierung, um die Prüfung des Status des *Shadow Memory* zu ermöglichen. Dabei benutzt AddressSanitizer effizientere Algorithmen, die ein kompakteres Design des *Shadow Memory* ermöglichen, sowie das Erkennen von Fehlern in Stack und globalen Variablen unterstützt. Zusätzlich kommt eine Laufzeitbibliothek zum Einsatz, die in Abschnitt 3.3 weiterführend beleuchtet wird.

3.1 Shadow Memory

AddressSanitizer verwendet eine direkte Skalierung mit Offset und widmet ein Achtel des virtuellen Adressraums dem *Shadow Memory*. Das bedeutet, dass jeder 8-Byte-Block des Anwendungsspeichers durch ein einzelnes Byte im *Shadow Memory* überwacht werden kann. Das Offset muss je nach Plattform ausgewählt werden.

Im *Shadow Memory* wird der Zustand jedes 8-Byte-Blocks des Anwendungsspeichers mit einem einzelnen Byte kodiert:

- Der Wert 0 bedeutet, dass alle 8 Bytes des Anwendungsspeicherblocks adressierbar sind
- Ein Wert k($1 \leq k \leq 7$) gibt an, dass die ersten k Bytes adressierbar sind, wobei k der Wert des Bytes ist
- Negative Werte zeigen an, dass der gesamte 8-Byte-Block nicht adressierbar ist. Verschiedene negative Werte werden verwendet, um verschiedene Arten von nicht adressierbarem Speicher zu unterscheiden.

Die Adresse des *Shadow Memory* (ShadowAddr) kann durch eine einfache Transformation der Anwendungsspeicheradresse (Addr) berechnet werden. Gleichung (1) zeigt die Berechnung der Adresse im Shadow Memory.

$$\text{ShadowAddr} = (\text{Addr} >> 3) + \text{Offset} \tag{1}$$

Der Anwendungsspeicher wird – wie in Abb. 1 zu sehen – in zwei Parts mit korrespondierenden *Shadow Memories* aufgeteilt. Die Adressen im *Shadow Memory*-Bereich liefern Adressen in der „Bad Region". Diese Bad Region ist ein spezieller Bereich des Speichers, der als nicht zugänglich markiert ist, indem er durch Seitenschutzmechanismen unzugänglich gemacht wird.

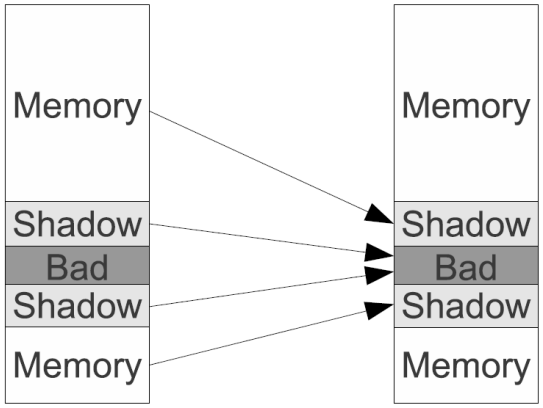

Abb. 1. AddressSanitizer Speicherabbildung

3.2 Instrumentierung

Die Instrumentierung geschieht am Ende der Optimierungspipeline des Compilers. Somit ist gewährleistet, dass nur diejenigen Speicherzugriffe instrumentiert werden, die die Optimierung des Compilers überlebt haben. Die Funktion `ReportAndCrash(Addr)` wird an vielen Stellen im Code eingefügt, um potentielle Speicherfehler zu melden.

Wenn ein 8-Byte-Speicherzugriff passieren soll, berechnet AddressSanitizer die Adresse im *Shadow Memory* nach der Formel, die in Abschnitt 3.1 erläutert wurde. Wie in Listing 4 zu erkennen ist, wird geprüft, ob der Inhalt von `ShadowAddr` ungleich 0 ist. Ist das Shadow Byte ungleich null, deutet dies darauf hin, dass der Speicherzugriff fehlerhaft ist.

```
1 if (*ShadowAddr != 0)
2     ReportAndCrash(Addr);
```

Listing 4. Check ShadowAddr

Tabelle 1. Laufzeitbibliothek Speicherallokation

rz1	mem1	rz2	mem2	rz3

Für Speicherzugriffe mit weniger als 8 Bytes ist die Instrumentierung etwas komplexer. Wenn das Shadow Byte positiv ist, wird überprüft, ob der Zugriff innerhalb der erlaubten Grenzen liegt. `AccessSize` in Listing 5 ist die Größe des Speicherzugriffs in Bytes.

```
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize > k))
    ReportAndCrash(Addr);
```

Listing 5. Check ShadowAddr < 8 Byte

3.3 Laufzeitbibliothek

Die Hauptaufgabe der Laufzeitbibliothek ist es, den *Shadow Memory* zu verwalten. Beim Start der Anwendung wird der gesamte Bereich des *Shadow Memory* reserviert, sodass kein anderer Teil der Anwendung ihn nutzen kann. Somit wird gewährleistet, dass der *Shadow Memory* einzig für die Überwachung des Anwendungsspeichers zur Verfügung steht.

Die Standardfunktionen `malloc` und `free` werden durch die Laufzeitbibliothek überschrieben, um die Funktionalität des Konzepts hinter AddressSanitizer sicherzustellen. Bei einer Speicherallokation wird eine Speicherregion mit ihren Redzones vom Betriebssystem allokiert. Für n Speicherregionen werden immer $n + 1$ Redzones allokiert, sodass die rechte Redzone einer Speicherregion die linke Redzone einer anderen Speicherregion ist. Redzone „rz1“ in Tabelle 1 beinhaltet zusätzlich Informationen des Allokators, wie zum Beispiel die Allokationsgröße, Thread-Id usw.

Die Speicherfreigbar mittels `free` „vergiftet“ den gesamten Speicherbereich und legt ihn in Quarantäne, sodass er von `malloc` nicht erneut allokiert werden kann. Die Quarantäne wird als FIFO-Warteschlange realisiert, die eine feste Menge an Speicher hält. Beide Speicherbefehle (`malloc` und `free`) zeichnen den Callstack auf, um informative Fehlerberichte liefern zu können.

4 TECHNISCHE ANALYSE: MEMORY CORRUPTION DETECTOR SANITIZER

Im Paper *Enhanced Memory Corruption Detection in C/C++ Programs* [7] stellen Ching-Yi Lin und Wuu Yang das neue Werkzeug „mcds“ vor und vergleichen es mit etablierten Tools wie zum Beispiel AddressSanitizer. Die Speicherfehlererkennung ist Zeit- und Speicheraufwändig und kann die Programmausführung erheblich verlangsamen. Hardwareunterstützung könnte die Programmausführung beschleunigen, jedoch mangelt es an Implementierungen, die diese Hardwareunterstützung nutzen [1].

Mit dieser Motivation wurde mcds entwickelt. Mcds nutzt die Intel SGX-Erweiterungsunterstützung für hardwarebasiertes Shadow Memory. Mcds besteht aus drei Teilen: Instrumentierung, einer Laufzeitbibliothek und Erweiterungen mit Intel SGX.

Die Besonderheit von mcds ist, dass es Speicherfehler erkennt, die Redzones fester Größe verursachen. Redzones fester Größe werden von vielen etablierten Speicherfehlererkennungswerkzeugen wie auch AddressSanitizer verwendet. In Abb. 2 kann man einen solchen Speicherfehler sehen, der bei AddressSanitizer nicht erkannt werden würde. Die Größe des Character Arrays a übersteigt die Größe von a und seine Redzone. Mcds erkennt Speicherfehler, die aus Redzones fester Größe resultieren dadurch, dass mehrere virtuelle Speicherseiten in eine physikalische Speicherseite zusammengefasst werden. Dadurch verursacht mcds keinen signifikanten Speicherverbrauch und nutzt die Hardwarefähigkeiten für eine schnellere Ausführung.

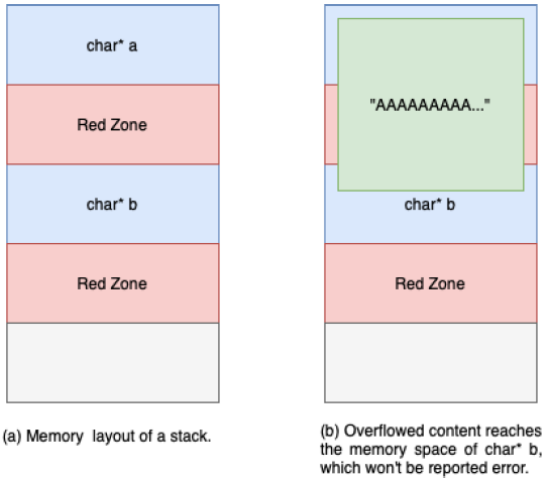

Abb. 2. Problem: Redzones fester Größe

4.1 Instrumentierung

Auch mcds macht von binärer Instrumentierung Gebrauch. Während der Kompilierung wird ein „Sanitizer Pass“ angewendet. Der Sanitizer Pass ist eine spezielle Phase im Kompilierungsprozess, bei der der Code analysiert und modifiziert wird. Der mcds-Sanitizer-Pass ersetzt dabei Aufrufe zur Speicherallokation und Speicherfreigabe mit Aufrufen, die in der Laufzeitbibliothek von mcds definiert sind. Zusätzlich wird eine Report-Funktion in die instrumentierte Binärdatei eingefügt. Diese Funktion wird aufgerufen, wenn ein ungültiger Speicherzugriff oder ein Prozessabsturz erkannt wird. Die Report-Funktion ist dafür verantwortlich, detaillierte Fehlerberichte zu erstellen, die helfen, die Quelle des Speicherfehlers zu identifizieren.

4.2 Laufzeitbibliothek

Die Laufzeitbibliothek von mcds beinhaltet den „debug allocator“ und den „page fault analyzer“. Der debug allocator verwaltet den Speicher, sodass Speicherfehler gefunden werden können. Der page fault analyzer interpretiert die Speicherseitenausnahmen.

Beim der Allokation von Speicher wird jedes Speicherobjekt auf eine separate virtuelle Speicherseite abgebildet. So ist der Abstand zwischen einzelnen (kleinen) Objekten im Speicher fast eine ganze Speicherseite. Um den Speicherbedarf gering zu halten werden mehrere dieser virtuellen Speicherseiten auf eine physikalische Speicherseite abgebildet. Dieser Vorgang wird *page aliasing* genannt [7].

Die Freigabe von Speicher wird die betroffene Speicherseite mit der PROT_NONE Flag ausgestattet. Die Seite wird zunächst unzugänglich für andere Speicherzugriffe und wird geleert. Danach kommt die Seite wieder in den Speicherpool zurück.

4.3 Verwendung von Shadow Memory und Intel SGX

Während Werkzeuge wie zum Beispiel AddressSanitizer softwarebasiertes *Shadow Memory* einsetzen, nutzt mcds Intel SGX (Software Guard Extensions), um *Shadow Memory* hardwareseitig zu implementieren. Dies reduziert den Overhead, den es benötigt die Speicheradresse im *Shadow Memory* zu berechnen. Auch wird Overhead minimiert, da die Redzones bei mcds Seitenbasiert sind, während die Redzones bei AddressSanitizer nicht zwangsläufig seitenbasiert ausgerichtet sind. Anstatt softwarebasiertes Shadow Memory zu nutzen, um die Gültigkeit von Speicheradressen zu überprüfen, verfolgt mcds die Signale des EPCM Registers. Diese Signale zeigen ungültige Zugriffe an. Bei einem Verstoß gegen das EPCM Register wird ein Signal ausgelöst, das vom Signal-Handler abgefangen wird, was eine effizientere Fehlererkennung ermöglicht.

5 DISKUSSION

In diesem Kapitel sollen die beiden Speicherfehlererkennungswerkzeuge AddressSanitizer und mcds miteinander verglichen werden. Dabei konzentrieren wir uns auf Aspekte wie die Effizienz der Speicherverwaltung und die Genauigkeit der Fehlererkennung.

AddressSanitizer, ein weit verbreitetes und anerkanntes Tool, verwendet eine softwarebasierte Implementierung von Shadow Memory und Red Zones, um Speicherfehler zu erkennen. Es nutzt Bit-Operationen, um die Zuordnung von Speicheradressen zu überwachen und hat sich als effektiv erwiesen, bringt jedoch einen signifikanten Zeit- und Speicheraufwand mit sich.

Mcds hingegen stellt eine neuere Lösung dar, die auf der LLVM-Toolchain basiert und sich durch die Integration von Intel SGX (Software Guard Extensions) auszeichnet. Diese Hardware-Unterstützung ermöglicht eine effizientere Verwaltung des Shadow Memory, wodurch der Overhead reduziert und die Leistung verbessert wird. Mcds verwendet seitenbasiert ausgerichtete Red Zones und nutzt das Enclave Page Cache Map (EPCM) Register als Hardware-Shadow-Memory, was die Fehlererkennung beschleunigt und die Robustheit erhöht, insbesondere in Mehrprozess- und Multithread-Umgebungen.

Zuerst betrachten wir die Geschwindigkeit, bis ein Stack-Overflow-Fehler und ein Use-After-Free-Fehler im Firefox-Projekt durch die beiden Werkzeuge gefunden wurde. Dabei wurden dieselben Eingabemuster verwendet, um dieselben Speicherfehler zu provozieren. Aus den Ergebnissen aus Tabelle 2 lässt sich schließen, dass mcds ungefähr 6 Mal schneller als AddressSanitizer ist. Die Spalte „Core Dump“ gibt die Ergebnisse für das Originalprogramm ohne Sanitizer an. Anhand der Core-Dump-Ergebnisse und der mcds-Ergebnisse kann ein 2-facher Verlangsamung durch die mcds-Erkennung angenommen werden. Ähnliche Ergebnisse lassen sich auch aus Tabelle 3 ableiten. Dort wurden ebenfalls AddressSanitizer und mcds verglichen, wie sie beim Chrome Browser Projekt abschneiden. Die Core Dump Spalte zeigt ebenfalls die Zeit zum Absturz ohne Speicherfehlererkennungswerkzeug.

Tabelle 2. Geschwindigkeitsvergleich beider Werkzeuge Firefox

	AddressSanitizer	mcds	Core Dump
Stack Overflow	532,1 s	97,3 s	42,3 s
Use-After-Free	627,5 s	103,4	52,7 s

Tabelle 3. Geschwindigkeitsvergleich beider Werkzeuge Chrome

	AddressSanitizer	mcds	Core Dump
Stack Overflow	631,85 s	51,64 s	32,14 s
Use-After-Free	985,1 s	86,02 s	47,34 s

Tabelle 4. Kompolierzeiten einzelner Projekte

	AddressSanitizer	mcds	Ohne Werkzeug
Firefox	2h 56min 5s	2h 15min 17s	1h 23min 10s
Chrome	2h 13min 3s	2h 24min 23s	1h 10min 7s
PHP	8h 42min 2s	9h 2min 4s	4h 27min 12s

Durch die Implementierung der binären Instrumentierung verlängern sich auch die Kompilierzeiten der Software, die mit einem Speicherfehlererkennungswerkzeug ausgestattet wird. Aus Tabelle 4 lassen sich die Kompilierzeiten von Firefox, Chrome und PHP ablesen. Die Projekte wurden jeweils mit AddressSanitizer und mcds ausgestattet, sowie ohne Speicherfehlererkennungswerkzeug kompiliert. Die Messergebnisse zeigen, dass sich keine signifikante Zeitunterschiede zwischen den beiden Werkzeugen ergeben. Das Ausstatten der Softwareprojekte mit einem der beiden Werkzeuge resultiert in einer Verdoppelung der Kompilierzeit.

6 VERWANDTE ARBEITEN

Die in den Arbeiten genannten Referenzen sind essentiel, um den erweiterten Kontext der beiden Werkzeuge zu verstehen. Zum einen finden wir dort Dokumentation zu Werkzeugen, die mit AddressSanitizer oder mcds verglichen werden. Auch in dieser Arbeit sind Dokumentation zu anderen Werkzeugen, wie zum Beispiel Apples Malloc Debugging Feature [5], aufgelistet. Die referenzierten Arbeiten stellen nicht nur Begriffserklärungen bereit, sondern helfen auch die wissenschaftlichen Erkenntnisse der Arbeiten in einen zeitlichen und wissenschaftlichen Rahmen einzuordnen.

Begriffserklärungen finden wir in dieser Arbeit zum Beispiel in *Modern operating Systems* von Tanenbaum und Bos [15], welches ein Standardwerk zu modernen Betriebssystemen ist. Bei AddressSanitizer und mcds lässt sich gut erkennen, dass dort viele Quellen zu Referenzimplementierung genannt sind. Somit lässt sich die Lösung mit anderen Lösungen vergleichen und Vor- und Nachteile gegenüberstellen.

7 SCHLUSS

In dieser Arbeit wurden die Werkzeuge AddressSanitizer und mcds detailliert untersucht und verglichen, wobei deren Ansätze zur Erkennung und Vermeidung von Speicherfehlern im Vordergrund standen. Beide Werkzeuge haben sich als leistungsfähige Lösungen zur Verbesserung der Software-Sicherheit erwiesen, jedoch mit unterschiedlichen Stärken und Schwächen.

AddressSanitizer ist ein etablierter Ansatz zur Erkennung von Speicherfehlern, der durch seine umfassende Unterstützung und Integration in Compiler-Frameworks wie LLVM weit verbreitet ist. Im Gegensatz dazu nutzt mcds einen innovativen Ansatz, der hardwaregestützte Techniken wie Intel SGX integriert, um die Effizienz der Speicherüberwachung zu verbessern. Durch die Nutzung von seitenbasierten Red Zones und die Verwendung von Hardwareunterstützung zur Validierung von Speicherzugriffen kann mcds den Laufzeit-Overhead signifikant reduzieren. Dies macht mcds besonders geeignet für Anwendungen, die eine hohe Leistung erfordern und gleichzeitig gegen Speicherfehler geschützt werden müssen. Die Ergebnisse der durchgeführten Experimente zeigen, dass mcds in vielen Szenarien schneller arbeitet als AddressSanitizer, insbesondere bei der Erkennung temporaler Speicherfehler.

Insgesamt bieten beide Werkzeuge wertvolle Beiträge zur Verbesserung der Software-Sicherheit. Zukünftige Arbeiten könnten darauf abzielen, die Vorteile beider Ansätze zu kombinieren und die Hard- und Softwarekombination, die durch mcds eingeführt wurde, zu optimieren oder dessen Komplexität zu reduzieren. Die kontinuierliche Weiterentwicklung und Verfeinerung von Werkzeugen wie AddressSanitizer und mcds wird entscheidend dazu beitragen, die Zuverlässigkeit und Sicherheit moderner Softwareanwendungen zu erhöhen.

LITERATUR

[1] 2024. Hardware-assisted AddressSanitizer Design Documentation.

[2] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*. 213–223. https://doi.org/10.1109/CGO.2011.5764689

[3] W. Cheng, Qin Zhao, Bei Yu, and S. Hiroshige. 2006. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *11th IEEE Symposium on Computers and Communications (ISCC'06)*. 749–754. https://doi.org/10.1109/ISCC.2006.158

[4] Robert O. Hastings and Beverly A. Joyce. 1991. Fast detection of memory leaks and access errors. https://api.semanticscholar.org/CorpusID:1798819

[5] Apple Inc. 2013. Enabling the Malloc Debugging Features. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/MallocDebug.html

[6] Intel. [n. d.]. https://www.intel.com/content/www/us/en/developer/tools/oneapi/inspector.html

[7] Ching-Yi Lin and Wuu Yang. 2023. Enhanced Memory Corruption Detection in C/C++ Programs. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops* (<conf-loc>, <city>Salt Lake City</city>, <state>UT</state>, <country>USA</country>, </conf-loc>) *(ICPP Workshops '23)*. Association for Computing Machinery, New York, NY, USA, 71–78. https://doi.org/10.1145/3605731.3605903

[8] microfocus. [n. d.]. https://www.microfocus.com/documentation/extend-acucobol/925/BKUSUSPROGS047.html

[9] Nicholas Nethercore. 2004. Dynamic binary analysis and instrumentation. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf

[10] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *SIGPLAN Not.* 42, 6 (jun 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[11] Gene Novark and Emery D. Berger. 2010. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. Association for Computing Machinery, New York, NY, USA, 573–584. https://doi.org/10.1145/1866307.1866371

[12] Oracle. [n. d.]. https://docs.oracle.com/cd/E77782_01/html/E77795/gmzsf.html

[13] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 135–148. https://doi.org/10.1109/MICRO.2006.29

[14] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[15] Andrew S. Tanenbaum and Herbert Bos. 2015. *Modern operating systems* (4. ed ed.). Prentice Hall, Boston.