Seminararbeit

Bearbeitung des wissenschaftlichen Papers: „An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety C Programs“

Gregor Pridun

Matrnr.:9725153

Studienkz.:937

1) Einleitung

In diesem wissenschaftlichen Paper aus dem Jahr 2004 beschäftigten sich die Autoren Wei XU, C. DuVarney und R. Sekar mit einer Strategie zur effizienten Auffindung von Speicherzugriffsfehlern bei der Verwendung von Pointern und Arrays. Die vorgestellten Methoden und Tools sind für die Programmiersprache C ausgelegt.

Die Verwendung von Pointern ist ein wichtiges und in vielen Programmen oft verwendetes Konzept. Doch im Programmieralltag hat sich gezeigt, daß die Verwendung dieses Konzepts Probleme in Form von Speicherzugriffzfehlern und inkorrekten Ergebnissen mit sich bringt. Diese Fehler sind oft schwer durch konventionelle Debuggingmethoden auffindbar und behebbar. Oft führen diese Art von Fehlern zu so genannten „Core-Dumps“, welche ein sofortiges Ende der Ausführung des Programmes zur Folge haben. Aus diesem Grund macht es Sinn sich mit einer Strategie auseinander zu setzen, die es ermöglicht diese Fehler auf effiziente Weise ausfindig zu machen und dadurch in weiterer Folge zu vermeiden.

2) Grundbegriffe und Methodik

Zum besseren Verständnis von möglichen Speicherzugriffsfehlern sind diese prinzipiell in zwei Gruppen zu unterteilen:

1.) örtliche (oder auch „spatial“ genannt) Speicherzugriffsfehler: Diese treten dann auf, wenn man versucht auf Speicher zuzugreifen der nicht vorher allociert wurde. Dies passiert zum Beispiel bei der „Out-of-Bound exception“, wenn fehlerhafterweise auf die Speicherstelle hinter dem letzten gültigen Zeichen eines Array zugegriffen wird.
2.) Zeitliche (oder auch „temporal“ genannt) Speicherzugriffsfehler: Hier wird versucht auf Speicher zuzugreifen, der zwar vorher allociert wurde, aber in der Zwischenzeit wieder freigegeben wurde.

Die in diesem Paper vorgestellte Methode, zielt darauf ab alle Speicherzugriffsfehler zu entdecken und dies auf eine Art, daß möglichst wenig Änderungen am Sourcecode vorgenommen werden müssen. Dazu soll maximale Kompatibilität bei der Zusammenarbeit mit undmodifierzierten Code gewährleistet sein. Die Autoren haben einen Source-to-Source-Compiler geschrieben, welchen Sie mit einigen Programmen getestet haben. In diesem Berreich spielen auch notwendige Optimierungen eine große Rolle, da der entstehende Overhead in Grenzen gehalten werden muss.

Damit diese Methode angewendet werden kann müssen folgende drei Einschränkungen beachtet werden:

1.) Es wird kein eigenes angepasstes Speichermanagement verwendet. Sollte dies der Fall sein so sind manuelle Anpassungen notwendig.
2.) Es dürfen keine Integervariablen zu Pointer gecastet werden
3.) Alle Casts müssen dem Up- und Downcast- Paradigma unterliegen.

Die Grundidee besteht darin zu jedem Pointer Informationen über diesen (sogenannte Metainformationen) mitzuspeichern und zu verwalten. Mit Hilfe dieser Metainformationen ist es möglich festzustellen, ob ein bestimmter Pointer gültig ist oder nicht. Dabei ist anzumerken, daß in C ungültige Pointer durchaus erlaubt sind. Jedoch ist es nicht erlaubt solche Pointer zu dereferenzieren.
Das vorgestellte System speichert diese Metadaten in eine eigene vom eigentlichen Pointer vollkommen getrennte Datenstruktur ab. Die Verlinktung des pointer mit diesen Inforamtionen wird über den Variablennamen erzielt: zu jedem pointer „exampleptr“ wird eine Metadatenstruktur „exampleptr_info“ definiert. Diese struct mit dem Typnamen „ptr_info“ besteht aus folgenden Feldern:

- void *base : Diese Felder enthält die Basisaddresse des Pointers
- signed long size: Darin wird die Speichergröße des Ziels des Pointers gespeichert
- cability *cap_ptr: Dieses Feld wird zur Auffindung von temporalen Speicherzugriffsfehler verwendet. Durch dieses Feld kann ein Pointer auf den Status „valid“ oder „invalid“ gesetzt werden. Die Capabilities aller in einem Programm zu Laufzeit verwendeten Pointer werden hierbei in einer gemeinsamen Datenstruktur dem „Global Capabilities Store“ verwaltet. Der Vorteil dieser Methode besteht darin, dass man nur ein Bit benötigt um den Status zu speichern. Allerdings können Capabilites nicht wiederverwendet werden, weshalb es in pointerintesiven Programmen zu Speicherplatzproblemem kommen kann.
- unsigned long cap_index: Aus den im vorhergehenden Punkt erwähnten möglichen Speicherplatzproblemen, haben die Autoren diesen Index für jede Pointercapability definiert. Dieser wird immer erhöht, wenn ein Speicherpaltz realloziert oder freigeben wurde. Dadurch ist es möglich Capabilities wiederzuverwenden.
- struct ptr_info *link: Dieses Feld wird benötigt, wenn ein Pointer auf eine Struktur zeigt, welche wiederum Pointer beinhaltet. Hier werden die Metainformationen für diesen Referenten mitgespeichert.

Die ersten beiden Felder werden dazu verwendet um örtliche Speicherzugriffsfehler aufzudecken, während .die Variablen „cap_ptr“ und „cap_index“ dazu beötigt werden sogenannte temporale

Speicherzugriffsfehler zu entdecken.

Der „ Global Capabilities Store“ stellt hierbei eine der Kernstücke des Systems dar. Da globale Variablen niemals temporal ungültig sind, können sich diese eine einzige Capability in Form einer globalen Variable teilen. Neben dieser Komponente sind für diese Verwaltungstruktur noch zwei weitere definiert. Eine Komponente ist dabei zuständig für die Heap-Speicherverwaltung. Diese wird im Folgenden „Heap Capability Store“ genannt kurz HCS. Dieser ist als erweiterbarer Array implementiert. Die einzelnen Speicherfelder können entweder eine Capibility enthalten oder sind noch frei. Die freien Felder des Arrays sind untereinander als verkettete Liste (genannt die HCS free list) organisiert. Daß heißt jedes freies Feld enthält einen Pointer zum nächtes unbenutzten Bereich. Die dritte und letzte Komponente ist für lokale Pointervariablen zuständig. Der „Stack Capability Store“ (kurz „SCS“) ist so implementiert, daß sich alle lokalen Variablen eines Frames eine einzige Capability teilen, da diese ja (z.B. beim Aufrauf einer Funktion) alle gleichzeitig alloziert und dannach gemeinsam befreit werden. Diese Capability wird nach dem Last-in First-out -Prinzip im SCS verwaltet.

3) Anwendung der Methodik

Nachdem die Organisation der Metadaten von Pointern und deren einzelne Felder näher beleuchtet worden sind, ist es nun notwendig den konkreten Einsatz dieser Methodik und wichtige dabei zu beachtende Aspekte zu beschreiben. Mit der Transformation der Pointerdefinitionen ist nur ein Teilbereich der vorgegebenen Problematik abgedeckt. Es ist zusätzlich bei der Übersetzung des Sourcecodes notwendig an jeder Stelle, an welcher ein Pointer dereferenziert wird Methoden/Funktionen einzufügen, welche die einzelnen Felder der Metainformationen dahingehen überprüfen, ob der entspechende Pointer „valid“ ist oder nicht. Diese Mechnismen wurden als zwei Makros „CHECK_SPATIAL“ und „CHECK_TEMPORAL“ realisiert, welche automatisch vom Prototypen des entwickelten Source-to-Source-Compilers an den notwendigen Stellen eingefügt werden.

Ein weiter wichtiger Fall ist natürlich die Zuweisung und Initialisierung von den Pointervariablen. In diesem Berreich sind vor allem drei Fälle zu unterscheiden:

1.) Ein Pointerwert wird einem anderen zugewiesen. Bei dieser Operation müssen die entsprechenden Metadaten ebenfalls richtig zugewiesen werden (aus ptr1=ptr2 folgt ptr1_info=ptr2_info)
2.) Ein Pointer wird durch Pointeraritmethik erzeugt: Wenn ein neuer Wert durch zum Beispiel p++ erzeugt wird, dann muß ebenfalls das Feld p_info.link um eins erhöht werden.
3.) Ein Pointerwert wird erzeugt durch den &-Operator oder malloc(). Bei Aufruf von malloc() werden die Werte der Metadatenstruktur initialisiert. Ebenso wird eine entsprechende Capibility angelegt. Bei der Verwendung des Operator „&“ z.B. p=&q ist es notwendig, falls

q eine Datenstruktur sein sollte, welche pointer beinhaltet, daß p_info.link=q_info.link gesetzt wird.

Funktionen und deren Aufruf müssen des weiteren im Sourcecode dahingehend modifiziert werden, daß diese zusätzliche Parameter für die Metadaten der Pointer annehmen und diese auch wieder zurückliefern. Eine Stärke dieses Konzeptes besteht darin, daß die Metadaten gänzlich von den eigentlichen Pointern im Programm getrennt sind. So ist es möglich externe Funktionen, welche vielleicht in Form von kompilierten nicht veränderbaren libraries vorliegen, ohne die entsprechenden Metadaten aufzurufen und deren Ergebnisse weiterzuverarbeiten. Allerdings können dabei Speicherzugriffsfehler nicht ausgeschlossen werden, sofern man keine Wrapper-Funktionen für solche Aufrufe implementiert.

4) Sicheres Typecasting

Die Sprache C unterstützt jegliche Arten von Casts, das heißt die Umwandlung von einem Datentypen in einen anderen. Dabei handelt in den meisten Fällen um Casts zwischen verwandten Datentypen zwischen denen eine Supertype-Subtype-Beziehung besteht. Um solche Umwandlungen weiterhin konsistent zu ermöglichen haben die Autoren die Metadaten um ein weiteres Feld erweitert, das ebenfalls verwaltet werden muss. Dieses heißt „tid“, und gibt an, um welchen Datentypen es sich bei dem Referenten eines Pointers handelt. Mittels dieses Feldes ist es möglich zu entscheiden, welche Casts erlaubt sind und welche nicht. Während sogenannte Upcasts (von einen Subtypen auf einen Supertypen) immer sicher sind und nicht überprüft werden müssen, so gilt dies nicht für Downcasts, welche in jedem Fall vor deren Durchführung überprüft werden müssen. Mit dieser Erweiterung des Konzepts haben die Autoren nicht nur das Problem der „unsicheren“ Casts bearbeitet. Ebenfalls ist es damit möglich sichere „Unions“ in Programmen zu unterstützen, da diese implizite Typecasts darstellen.

Ein weiterer wichtiger Bereich bei der Implementierung dieser Methodik ist natürlich die Performance. Zur Laufzeit entsteht erheblicher Overhead, da vor jeder Derefenzierung eines Pointers und vor jedem Typecast entsprechende Kontrollen durchgeführt werden müssen. Daher ist die Optimierung eines solchen Systems von äußerster Wichtigkeit. Dabei geht es vorallem darum, wie man die Metadaten strukturiert und unnötige Überprüfungen aus dem Sourcecode streichen kann.

5) Die Optimierung der Performance

Hierbei unterscheiden die Autoren zwischen lokalen und globalen Optimierungen. Den größten Optimierungserfolg konnten diese dadurch erreichen, daß die Metadatenstruktur in zwei Unterstrukturen aufgeteilt wird:

1) header:Dies beinhaltet die Headerinforamtionen. Diese Inforamtionen können von mehreren

Pointern eines Blocks gemeinsam verwendet werden. Dadurch wird nicht nur Speicherplatz gespart sondern auch die Performance deutlich gesteigert, weil erheblich weniger Felder verwaltet werden müssen.

2) ptr_info: Dies beinhaltet die restlichen Felder, welche immer zu einem spezifischen Pointer gehören. Diese können nicht gemeinsam verwaltet werden.

Eine weitere Möglichkeit der Optimierung haben die Autoren in den Funktionen des GNU-C-Compilers (kurz „gcc“) gefunden. Dieser besitzt umfangreiche Möglichkeiten im Bereich der Ausdrucksersetzung und der Eleminierung von toten Variablen, kann diese allerdings nicht auf selbstdefinierte Strukturen anwenden. Deswegen ersetzt der entwickelte Prototyp überall, wo es sicher ist, die verwendete Metadatenstruktur durch einzelne Variablen. Diese können dann vom gcc optimiert werden.

Als letzte Möglichkeit der lokalen Optimierung wird die Eleminierung von unnötigen Operationen im Zusammenhang des „Stack Capability Stores“ aufgeführt. Hier können unnötige Überprüfungen in Funktionen weggelassen werden, in welchen keine Pointer zu lokalen Variablen erzeugt werden.

Zusammendfassend verbessern diese beschriebenen Optimierungen die Performance des angewendeten Systems um den Faktor zwei.

Die globalen Opimierungsmöglichkeiten werden in dieser Publikation zwar andiskutiert, allerdings wurde keine einzige im entwickelten Prototypen zur Anwendung gebracht. Hier geht es vor allem auch um Konzepte der statischen Analyse um unnötige Überprüfungen und Datenfelder zu eleminieren.

6) Evaluierung der Performance

Der entwickelte Prototyp welche syntax-basierend die Source-to-Source-Transformationen durchführt ist in der Sprache „objective CAML“ implementiert und verwendet CIL als Frontend. Die Auswirkungen der Transformationen wurden mittels einiger Testprogramme anhand der „Olden“ und „SPECINT“ Benchmarks gemessen. Die getesteten Programme wiesen dabei eine mittlere Größe auf, das heißt die Lines of Code rangierten zwischen 565 und 24.399. Getestet wurde unter Red Hat Linux 9 mit einer Intel Xeon Workstation (3 Ghz) und 3 Gigabyte RAM.

Der Overhead bei den ausgeführten Programmbefehlen lag bei den Experimenten zwischen 112 und 269 Prozent. (Durchschnittlich bei 162%). Beim Speicherverbrauch hingegen, gab es starke Unterschiede zwischen den „Olden“ und „SPECINT“ benchmarks. Der Overhead liegt bei den „Olden“ benchmarks bei nahzu 500%, was daran liegt, argumentieren die Autoren, daß diese Tests fast ausschließlich Strukturen allozieren, welche Pointer beinhalten.

Mit diesen Ergebnissen vorliegend ist es natürlich interessant, diese in Kontext zu verwandet Methoden und Arbeiten zu betrachten. 1994 wurde von den Autoren Todd Austin, Scott Breach und Gurindar Sohi ein wissenschaftliches Paper veröffentlicht mit dem Titel „Efficient Detection of All

Pointer and Array Access Errors“. Damals entwickelten diese dieselbe Grundidee, welche auch dieser hier besprochenen wissenschaftlichen Publikation zu Grunde liegt. Allerdings untscheiden sich die Implementierung und die eingesetzten Möglichkeiten der Optimierung deutlich. Das 1994 entwickelte Konzept des „Safe C“ besaß einen Overhead von bis zu 500% (im Durchsschnitt 300%) bei der Ausführung von Befehlen. Hier zeigt sich, wie abhängig die Nützlichkeit solcher Konzepte von der Art und Weise der Implementierung und Optimierung ist. In der hier vorgestellten Methode ist die Performance signifikant besser und in ihrer Ausprägung in Messwerten deutlich stabiler.

7) Alternative Lösungsansätze und Ausblick

Der hier vorgestellte Ansatz ist natürlich nicht der einzige um dem Problem der Vehinderung von Speicherzugriffsfehlern zu begegnen. Es gibt eine Vielzahl von Methoden und Lösungen die dem Programmierer helfen sollen, solche Fehler zu verhindern. Im letzten Teil der hier besprochenen Arbeit unterscheiden die Autoren hierbei zwischen vier Kategorien:

1) Ansätze, welche die Debugging-Arbeit erleichtern sollen: Zu diesen Ansätzen gehören bereits enwickelte Systeme wie zum Beispiel bcc oder purify. Diese Systeme erkennen „nur“ gewisse Unterklassen von Speicherzugriffsfehlern. So erkennt Purify gewisse örtliche Fehler nicht und ist auch nicht in der Lage temporale Fehler zu erkennen. Da der Fokus bei diesen Systemen auf der Erleichterung des Debuggingprozesses liegt, spielt der Overhead nur eine geringe Rolle.
2) Ansätze, welche Garbage-Collection verwenden: In diesem Berreich sind die Projekte Cyclone und Ccured zu erwähnen. Diese versuchen eine ähnliche Infrastruktur wie sichere Sprachen wie zum Beispiel Java zu bieten. Leider ist die Portierung von bestehenden Programmen oft nicht einfach zu bewerkstelligen und der Programmierer muß durch das Einfügen von Annotationen nachhelfen. Die Performance dieser Systeme ist für manche Einsatzgebiete akkzeptabel, allerdings nicht für Kernelnahe oder zeitkritische Prozesse.
3) Ansätze, welche möglichst große Rückwärts-Kompatibilität gewährleisten. In diese Sparte fällt die in dieser Arbeit vorgestellte Methode ebenso wie das weiter oben erwähnte Konzept von „Safe C“. Der Nachteil von „Safe C“ ist allerdings die deutlich geringere Performance sowie die Verwendung von „fat pointers“ (das heißt die Pointerstruktur wird verändert). „Fat Pointer“ können dazu führen, daß die Kompatibiltät zu vorkompilierten Libraries nicht gegeben ist.
4) Ansätze, welche Security-Aspekte in den Vordergrund rücken. Hier werden die Projekte von CRED und Lam genannt. Dies ist eine Sonderkategorie, da diese nicht diesselbe Problemstellung zu lösen versucht, wie die Projekte in den übrigen Kategorien. Es geht vorallem darum Sicherheit gegen Format-String-Attacken zu gewährleisten und Exploits durch Speicherzugriffsfehler zu verhindern.

Der hier vorgestellte Ansatz entwickelt eine Idee weiter, welche bereits 1994 in Form von „Safe C“ vorhanden war. Es zeigt sich das Potential dieser Weiterentwicklung in der besseren Performance gegenüber „Safe C“.

Die Ziele der zukünftigen Arbeit an diesem Projekt sind die Optimierungen auszubauen und zu verbessern. Desweiteren wollen die Autoren in Zukunft noch mehr Arten von Typecasts unterstützen und Support für Programme realisieren, welche auf eigene Routinen zur Speicherallozierung zurückgreifen.