Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch die Informatik 4 (Verteilte Systeme und Betriebssysteme), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Arbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den

Bachelorarbeit

Umfeld: KESO (http://www4.cs.fau.de/Research/KESO) ist eine Java- Laufzeitumgebung für eingebettete Systeme mit dem Fokus auf software-basierten Speicherschutz. KESO ermöglicht die isolierte Ausführung mehrerer Anwendung auf einem Mikrokontroller. Um eine möglichst ressourcenschonende Laufzeitumgebung bereitzustellen, wird auf viele teure Java-Mechanismen (Reflection, Just-in-Time-Übersetzung, dynamisches Nachladen von Klassen, Exceptions) verzichtet und das komplette System wird vor der Ausführung in C-Code übersetzt.

Vorarbeiten: Der software-basierte Speicherschutz in KESO basiert auf der typsicheren Sprache Java. Da die Gewährleistung der Typsicherheit nicht vollständig zur Übersetzungszeit erfolgen kann, werden in den übersetzen Code Laufzeitprüfungen (Null-Pointer-Checks, Array-Bounds-Checks, etc.) eingefügt, um ggf. Fehler zur Laufzeit zu erkennen. Ein wesentlicher Teil des Overheads entsteht durch diese Laufzeitchecks. Im Rahmen des KESO-Projekts wurde untersucht, inwiefern der software-basierte Speicherschutz graduell angewendet werden kann, indem nur ein Teil der Laufzeitchecks in das System eingebracht wird, um so eine feiner Abwägung zwischen Kosten und Safety-Zugewinn zu ermöglichen. Detaillierte Informationen zum Thema können den Folien (http://www4.cs.fau.de/m̃ike/iids2010.pdf) eines Vortrags zu diesem Thema entnommen werden.

Aufgabenstellung: In dieser Arbeit soll gradueller Speicherschutz semi-automatisch implementiert werden. Hierbei soll z.B. Wissen über das Speicherlayout der Zielplattform, wie dieses zum Beispiel in Linkerscripten des Linkers beschrieben ist, in den Compilerlauf mit einfliessen, um eine sinnvolle Vorauswahl der relevanten Laufzeitchecks durchzuführen. Außerdem ist zu untersuchen, welche anderen Informationen eine solche Auswahl ggf. unterstützen können und wie der Anwendungsentwickler den Compiler in der Auswahl unterstützen kann.

Betreuung: Prof. Dr.-Ing. Wolfgang Schröder-Preikschat, Dipl.-Inf. Michael Stilkerich.

Bearbeiter: Michael Strotz

Abstract

Bei der KESO Multi-JVM handelt es sich um einen statischen Übersetzer, der Java-Code für Mikrocontroller übersetzt. Sie stellt die Isolation einzelner Anwendungen auf Basis der Typsicherheit Javas sicher. Hierzu sind Null- und Bereichsprüfungen nötig, die zusätzliche Kosten verursachen. Im Rahmen der Arbeit werden Nullprüfungen entfernt, falls die Zugriffe, vor denen sie schützen, ohnehin eine Fehlerbehandlung auslösen (Hardware-Trap). Weiter werden bei Bedarf Prüfungen verworfen, die nur der Typsicherheit, nicht aber der Speichersicherheit dienen. Die zusätzlichen Kosten können dadurch größtenteils eingespart werden. Allerdings liegt dies auch an anderen Optimierungstechniken, die bereits in KESO vorhanden sind.

Inhaltsverzeichnis

1 Grundlagen und Motivation

1.1 Hardwarebasierter Speicherschutz

Speicherschutz bedeutet allgemein, Programme so voneinander zu isolieren, dass das fehlerhafte Verhalten eines Programmes keine Auswirkungen auf die Stabilität anderer Programme hat. Hardwarebasierter Speicherschutz setzt eine Speicherschutzeinheit (MPU) oder eine Speicherverwaltungseinheit (MMU) voraus. Während MMUs in heutigen 32-Bit-Mikrocontrollern sehr selten anzutreffen sind, verfügen einige über eine MPU. So besitzt der Infineon Tricore TC1796 [6] eine solche. Auf dem ARMv7-M [2] ist optional eine MPU vorhanden.

Diese MPUs bieten beispielsweise Schutz durch „Eingrenzung“ an. Hierbei wird jedem Programm ein zusammenhängender Speicherbereich zugewiesen, auf den es zugreifen darf. Wird zu einem anderen Programm gewechselt, wird auch der Speicherbereich gewechselt. Wird versucht außerhalb des Bereiches zuzugreifen, wird ein Trap ausgelöst, welcher eine vom Betriebssystem zugewiesene Behandlungsroutine aufruft. Wurde die MPU aktiviert, wird jeder Speicherzugriff automatisch überprüft. Da die Prüfung von der Hardware direkt durchgeführt wird, geschieht sie schneller, als eine entsprechende softwareseitige Prüfung. Leider ist die Anzahl der Speicherbereiche, mit denen eine MPU programmiert werden kann, begrenzt. So sind auf dem TC1796 nur vier solche Bereiche für Daten möglich. Zusätzliche Laufzeitkosten entstehen bei der Umschaltung der Bereiche, zum Beispiel beim Programmwechsel, bei Betriebssystemaufrufen und bei Kommunikation zwischen Programmen.

Abbildung 1.1 zeigt die Isolation zweier Anwendungen durch die MPU. Die erste Zeile stellt den Speicher dar. Die hexadezimalen Zahlen sind Inhalt der Eingrenzungsregister. Darunter befinden sich die Zugriffsrechte zu den jeweiligen Bereichen. Beiden Anwendungen wird ein Daten- und ein Codebereich zugewiesen. Versucht nun Anwendung „App0“ auf die Adresse 0x80 zuzugreifen, löst dies einen Trap aus. Auch können beide Anwendungen nicht auf die Adresse 0 zugreifen, ohne unterbrochen zu werden.

Um Umschaltungen zu vermeiden, können mehrere Programme in einem Bereich ausgeführt werden. Dies hebt natürlich auch die Isolation dieser Anwendungen untereinander auf. Um diese wiederherzustellen, kann softwarebasierter Speicherschutz verwendet werden (siehe Abschnitt 1.2). Gleichwohl verursachen Zugriffe außerhalb dieses Bereiches einen Trap.

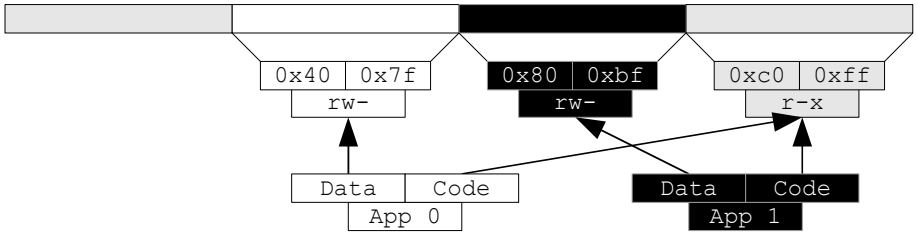

Abbildung 1.1: Isolation durch die MPU

1.2 Speicherschutz durch Typsicherheit

Typsicherheit kann verwendet werden, um Anwendungen voreinander zu schützen, die gemeinsam in einem Hardware-Speicherschutzbereich ausgeführt werden. Dies kann der Fall sein, wenn Schutzbereichwechsel vermieden werden sollen, oder wenn keine Speicherschutzeinheit vorhanden ist (siehe Abschnitt 1.1).

Typsicherheit bedeutet, dass Objekte eines bestimmten Datentyps nur so verwendet werden dürfen, wie es von der Programmiersprache für diesen Datentyp definiert wird. Programmiersprachen wie C, die nicht vollständig typsicher sind, erlauben jedoch Operationen wie Zeigerarithmetik, unsichere Zeigerkonvertierungen oder das manuelle Löschen von Objekten. Durch diese Operationen ist es möglich, jeden Bereich eines Adressraumes als ein Objekt eines beliebigen Typs zu interpretieren. Greift man so auf den Speicher einer anderen Anwendung, die sich im gleichen Speicherschutzbereich befindet, zu, kann dies die Stabilität dieser Anwendung stören.

Bei einer typsicheren Sprache wie Java enthält jede Referenz entweder den Ausnahmewert „null“ oder zeigt auf ein gültiges Objekt passenden Typs. Damit gibt es keine wilden Referenzen, die auf ungültige Objekte oder Objekte eines inkompatiblen Typs zeigen. Um zu verhindern, dass auf die ungültige Adresse „null“ zugegriffen wird, werden Laufzeitprüfungen eingefügt, sogenannte „Nullprüfungen“. Diese stoßen eine Ausnahmebehandlung an, falls eine solche Referenz dereferenziert wird.

Eine weitere Bedrohung der Typsicherheit bilden Vektorzugriffe, bei denen es sich in unsicheren Sprachen nur um Zeigerarithmetik handelt. Greift man auf Indizes zu, die die Größe des Vektors übersteigen, interpretiert die Anwendung unbekannten Speicher als Objekt vom Elementtyp des Vektors. Auch so könnte auf jeden beliebigen Speicherbereich auf beliebige Art zugegriffen werden. Um dies zu verhindern, werden bei sicheren Sprachen wieder Laufzeitprüfungen, sogenannte „Bereichsprüfungen“, eingeführt, die den Index vor dem Zugriff mit der Vektorgröße vergleichen und gegebenenfalls eine Ausnahmebehandlung verursachen.

Durch diese Maßnahmen können Anwendungen nur auf die Speicherbereiche zugreifen, die durch die ihnen bekannten Referenzen erreichbar sind. Objekte anderer Anwendungen sind nicht sichtbar, solange deren Referenzen dieser Anwendung nicht mitgeteilt

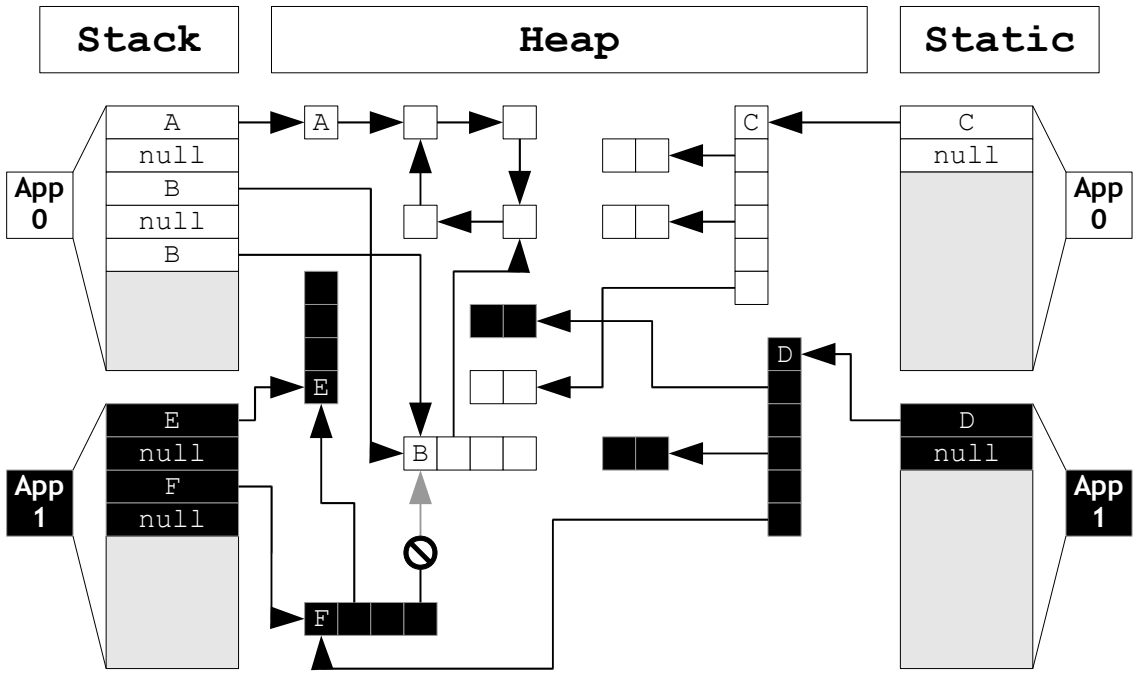

Abbildung 1.2: Sprachbasierte Isolation

werden. Eine solche Mitteilung könnte über statische Variablen geschehen, weshalb jede Anwendung eigene Kopien dieser besitzen muss. Auch bei der Kommunikation zwischen Anwendungen dürfen keine Referenzen ausgetauscht werden. Damit ist der Speicher anderer Anwendungen vor dieser Anwendung geschützt.

Objekte können nun, wie in Abbildung 1.2, ohne räumliche Isolation nebeneinander im Speicher liegen, ohne dass eine Anwendung die Objekte anderer Anwendungen manipulieren kann. Eine Referenz von Objekt F auf Objekt B ist hierbei nicht gestattet, da diese zu unterschiedlichen Anwendungen gehören. Durch diese Referenz könnte Anwendung „App1“ nicht nur das Objekt B manipulieren, sondern auch die Objekte bei A und potentiell alle Objekte von „App0“.

Neben diesem Schutz bietet Typsicherheit noch weitere Vorteile. So können Programmierfehler wie die Dereferenzierung ungültiger Referenzen oder Indizes genau erkannt und im Quelltext wiedergefunden werden. Weiter erlaubt die Typsicherheit eine genaue Analyse des Programms zur Übersetzungszeit.

Der Nachteil dieses softwarebasierten Speicherschutzes ist jedoch der zusätzliche Rechenaufwand, der durch die eingefügten Laufzeitprüfungen entsteht.

1.3 Die KESO Multi-JVM

Bei KESO [11] handelt es sich um eine Multi-JVM für eingebettete Systeme. KESO erlaubt es, mehrere Java-Anwendungen gleichzeitig - isoliert voneinander - auf einem Mikrocontroller auszuführen. Die Isolation basiert auf Typsicherheit.

KESO wurde für statische, eingebettete Systeme entworfen. Das bedeutet, dass ausführbarer Code nicht dynamisch nachgeladen werden kann. Alle Anwendungen sind beim Beschreiben des Mikrocontrollers bekannt. Daher ist es nicht nötig, deren Javacode zu interpretieren oder „Just-in-Time“, also während der Ausführung, zu nativem Code zu übersetzen. Der Javacode wird vor der Ausführung, also „Ahead-of-Time“, verifiziert und zu C-Code übersetzt. Dabei kann eine statische Analyse der gesamten Anwendung vorgenommen werden (sogenannte „whole program analysis“), welche tiefgreifende Optimierungen erlaubt. So sind beispielsweise zu einer Methode alle Positionen bekannt, an denen sie aufgerufen wird, wodurch Rückschlüsse über den Wertebereich ihrer Parameter möglich sind. Außerdem wird durch die statische Analyse versucht, Laufzeitprüfungen einzusparen. So kann sie unter Umständen erkennen, ob Bereichsprüfungen nötig sind, wenn in einer einfachen Schleife durch einen Laufindex auf einen Vektor zugegriffen wird. Auch Nullprüfungen können verworfen werden, wenn sichergestellt werden kann, dass eine Referenz ungleich *null* ist. Dies ist beispielsweise dann der Fall, wenn eine Referenz bereits dereferenziert wurde, da dies bei einer *null*-Referenz zu einer Ausnahme geführt hätte. Der C-Code wird anschließend in nativen Code übersetzt, welcher schließlich auf den Mikrocontroller geladen wird.

Die Kommunikation zwischen den Anwendungen geschieht durch Portale, die den „Java Remote Method Invocations“ oder „Remote Procedure Calls“ ähnelt. Hierbei spielt es keine Rolle, ob sich die Anwendung, mit der kommuniziert wird, auf dem gleichen oder auf einem anderen, verbundenen Mikrocontroller befindet. Schon deshalb dürfen keine Referenzen von einer Anwendung zur anderen weitergegeben werden (siehe Abschnitt 1.2).

Neben dem Verzicht auf das dynamische Nachladen von Klassen, weicht KESO auch in anderen Aspekten von der JVM-Spezifikation ab, um Java-Anwendungen auch auf schwachen Mikrocontrollern ausführen zu können. So werden teure Mechanismen wie Reflexion oder eine komplexe Ausnahmebehandlung nicht unterstützt.

1.4 Motivation und Ziel der Arbeit

Da Laufzeitprüfungen zusätzliche Kosten verursachen, soll durch die Arbeit überprüft werden, ob die Speicherschutzmechanismen der Hardware genutzt werden können, um softwareseitige Prüfungen zu ersetzen.

Zusätzlich zum Speicherschutz bietet Javas Typsicherheit die Möglichkeit Programmierfehler schnell und genau zu erkennen. In der vorliegenden Arbeit soll daher überprüft werden, ob programmseitige Prüfungen abgeschaltet werden können, die nicht notwen-

dig sind, um die Stabilität anderer Anwendungen zu schützen. Dies führt zwar zu einer Verletzung der Java-Spezifikation [7], kann aber dazu beitragen, die Ressourcen des Mikrocontrollers zu schonen und den Rechenaufwand zu reduzieren.

2 Analyse

Die Analyse beschäftigt sich mit folgenden Fragen:

- Welche Stufen des softwarebasierten Schutzes sind für den Anwender interessant?
- Gibt es Unterschiede in der Wichtigkeit der Prüfungen für die Stabilität?
- Welche Prüfungen können durch die Hardware übernommen werden und wie hilft das Speicherlayout noch, um Prüfungen zu sparen?
- Welche zusätzlichen Informationen benötigt KESO, um Prüfungen abschalten zu können?

2.1 Stufen des graduellen Schutzes

Für den Anwender lässt sich der graduelle softwarebasierte Schutz in zwei Stufen unterteilen.

Bei der ersten Stufe werden Prüfungen zwar auf die Hardware ausgelagert, das Verhalten des Programms im Fehlerfall bleibt aber erhalten. Die Ausnahmebehandlung wird nun durch einen Hardware-Trap ausgelöst, statt durch eine Laufzeitprüfung. Die Möglichkeit, Fehler frühzeitig zu erkennen, geht dadurch nicht verloren. Auch die Java-Konformität wird dadurch nicht weiter verletzt. Außerdem kann die statische Analyse zur Übersetzungszeit Referenzen als geprüft markieren, nachdem sie einmal dereferenziert wurden, da eine *null*-Referenz hierbei eine Ausnahme verursacht hätte. Diese Stufe ist immer empfehlenswert. Welche Prüfungen durch die Hardware ausführbar sind, wird in Abschnitt 2.3 untersucht.

Bei der zweiten Stufe zählt nur noch die Speichersicherheit, also die Isolation der Anwendungen untereinander. Die Fehlererkennung wird vernachlässigt. Welche Prüfungen abgeschaltet werden, ohne die Speichersicherheit zu gefährden, wird in Abschnitt 2.2 untersucht. Dadurch wird die Konformität mit der Java-Spezifikation [7] verletzt. Zugriffe jenseits der Vektorgrenzen führen nicht mehr zwangsläufig zu einer „IndexOutOfBoundsException“, sondern bringen jetzt undefiniertes Verhalten mit sich, wie in der Sprache C. Das Dereferenzieren einer Nullreferenz führt ebenfalls zu undefiniertem Verhalten. Diese Stufe empfiehlt sich nicht für alle Anwendungen. Solche, die sich noch nicht in der Praxis bewährt haben, sollten unter Verwendung der ersten Stufe kompiliert werden. Hat sich die Anwendung bewährt und ist eine Optimierung der Laufzeit erforderlich,

beispielsweise um sie auf einem schwächeren Mikrocontroller auszuführen, empfiehlt sich die zweite Stufe.

Eine dritte Stufe gibt es nicht. Die Speichersicherheit der Anwendungen muss immer erhalten bleiben. Das heißt, dass kein Speicherinhalt direkt verändert werden darf, der nicht der entsprechenden Anwendung gehört und dass dies der Anwendung auch nicht anderweitig ermöglicht werden darf. So dürfen keine wilden Referenzen entstehen, die Steuerflussintegrität darf nicht verletzt werden und Laufzeittypinformationen dürfen nicht manipuliert werden (siehe Abschnitt 2.2).

2.2 Auswirkungen ausgeschalteter Laufzeitprüfungen

Um die Isolation der einzelnen Anwendungen zu sichern, sind nicht alle Laufzeitprüfungen erforderlich. Gemessen daran, welche Auswirkungen das Ignorieren einer fehlgeschlagenen Prüfung zur Folge hat, werden sie in zwei Klassen unterteilt: lokale und globale Auswirkungen [9]. Die Unterabschnitte 2.2.1 und 2.2.2 untersuchen, in welche dieser Klassen ungeprüft durchgeführte Lade- beziehungsweise Speicheroperationen fallen.

Lokale Auswirkungen Das Einsparen einer Prüfung mit lokalen Auswirkungen kann ein Fehlverhalten der beinhaltenden Anwendung verursachen. Dieses Fehlverhalten gefährdet jedoch nicht die Stabilität anderer Anwendungen.

Globale Auswirkungen Werden Prüfungen mit globalen Auswirkungen entfernt, kann dies nicht nur Folgen für die Stabilität der beinhaltenden Anwendung haben, sondern auch auf die anderer Anwendungen. Beispielsweise könnten ungewollte Schreibzugriffe auf den Speicher anderer Anwendungen stattfinden. Das Lesen primitiver Daten aus fremdem Speicher wird hier nicht als globale Auswirkung angesehen, da dies keinen Stabilitätsverlust mit sich bringt. Auch die Anwesenheit schädlicher Software, die versucht, andere Anwendungen auszuspähen, wird nicht erwartet, da bei einem statischen System die gesamte Software schon zur Übersetzungszeit bekannt ist und daher vom Hersteller des Gesamtsystems überprüft werden kann.

2.2.1 Ladeoperationen

Wie prüfenswert Leseoperationen sind, hängt vom gelesenen Datentyp ab. Von Instanzfeldern und Vektoren können Referenzen oder primitive Daten gelesen werden. Zusätzlich können von jedem Objekt Laufzeittypinformationen abgefragt werden.

Primitive Daten Das Lesen primitiver Daten, wie *int*, *double* oder *char*, hat immer lokale Auswirkungen, da diese keinen Einfluss darauf haben, welche Speicherbereiche zugreifbar sind. Auch wenn durch einen fehlerhaften Lesevorgang ein beliebiger Wert

```
static void read_primitive_test() {
  int[] array = new int[2];
  int x = array[256]; //ungeprueft.
  array[x] = -1;      //geprueft.
}
```

Abbildung 2.1: Beispielcode

als Vektorindex interpretiert werden würde, würde eine Bereichsprüfung weitere Fehler verhindern. Der Beispielcode aus Abbildung 2.1 soll dies verdeutlichen: Die zweite Anweisung sei ungeprüft. Da durch sie keine Speicherinhalte verändert werden, stellt sie keine Gefahr für andere Anwendungen dar. Auch von dem zufälligen Wert in x, der dadurch entstanden ist, geht keine Gefahr aus. Wird dieser Wert, wie in der letzten Anweisung, als Index eines Vektorzugriffs benutzt, verhindert die zugehörige Bereichsprüfung ungültige Speicherzugriffe.

Referenzen Referenzen bestimmen welche Speicherbereiche zugreifbar sind. Daher hat das Lesen von Referenzen globale Auswirkungen. Wird ein beliebiger Wert als Referenz gelesen, erzeugt dies einen wilden Zeiger.

Hierzu wird der obige Beispielcode etwas verändert:

```
class RefTest {
 int data;
 static void read_reference_test() {
   RefTest[] array = new RefTest[2];
   RefTest x = array[256]; //ungeprueft.
   x.data = -1;            //geprueft.
 }
};
```

In der zweiten Anweisung wird wieder ungeprüft über die Grenzen des Vektors hinaus gelesen. Hierbei entsteht die Referenz x. Welchen Wert diese Referenz trägt ist unbekannt. Zeigt diese Referenz nicht zufällig auf *null* oder ein Objekt vom Typ $RefTest$, handelt es sich um eine wilde Referenz. In der letzten Anweisung wird der Speicher, auf den die Referenz x zeigt, beschrieben. Die Nullprüfung kann dies nicht abfangen, wenn es sich bei x nicht zufällig um eine *null*-Referenz handelt. Beim Schreiben könnten eigene Laufzeittypinformationen, eigene Referenzen oder fremder Speicher manipuliert werden. Dies verletzt die Speichersicherheit.

Laufzeit-Typinformationen Ein besonderes Feld ist „class-id“, welches den dynamischen Typ einer Instanz speichert. Handelt es sich bei der Instanz um einen Vektor,

enthält dieser ein weiteres besonderes Feld, das die Anzahl der Elemente des Vektors speichert. In der Sprache Java wird dieses Feld „length“ genannt.

Bei einem virtuellen Methodenaufruf wird anhand der „class-id“ festgestellt, welche Methode aufgerufen werden muss. Diese Zugriffe müssen daher immer geprüft werden, da durch einen ungültigen Wert beliebiger Code als Methode mit der Signatur des virtuellen Aufrufs interpretiert werden könnte. Dies würde die Steuerflussintegrität des Programms zerstören, wodurch auch Typsicherheit und damit die Isolation verloren gingen, da nun wieder jeder beliebige Wert als Referenz interpretiert werden könnte. Die Auswirkungen sind global.

Dazu ein Beispiel in Form der Klasse „ControlflowExample“:

```
class ControlflowExample {
  int getInt() { return 42; }
  Object[] getArray() { return new Object[1]; }
  static void test() {
    ControlflowExample ref = null;
    Object[] array = ref.getArray(); //ungeprueft
    array[0] = null;
  }
};
```

Würde beim Aufruf von „getArray“ eine falsche Klassen-ID gelesen werden, die zufällig dazu führt, dass statt dessen „getInt“ aufgerufen wird, könnte ein wilder Zeiger entstehen, der es erlaubt, den Speicher um Adresse 42 als Instanz eines Object-Vektors zu interpretieren. Die Typsicherheit wäre damit verloren.

Auch das ungeprüfte Lesen der Vektorgröße „length“ hat globale Auswirkungen. Zwar handelt es sich hierbei um eine Ganzzahl, die Bereichsprüfung benutzt die Vektorlänge jedoch, um festzustellen, ob sich ein Zugriff inner- oder außerhalb eines Vektors befindet. Wird ein unvorhersagbarer Wert als Vektorgröße interpretiert, funktioniert auch die Bereichsprüfung nicht mehr. Somit wäre ein unvorhersagbar großer Bereich des Speichers an Adresse *null* beliebig zugreifbar.

Ein weiteres Problem, das auftreten könnte, wenn „length“ nur lokale Auswirkungen zugeordnet werden, hängt mit der statischen Analyse zusammen. Hierzu folgender Beispielcode:

```
static void fill( int[] array, int value ) {
  for(int i = 0; i < array.length; ++i) {
    array[i] = value;
  }
}
```

Hier könnte die statische Analyse erkennen, dass innerhalb der Schleife keine Bereichsprüfung nötig ist, da i immer kleiner als *length* ist. Wird beim Zugriff auf *length* auf eine

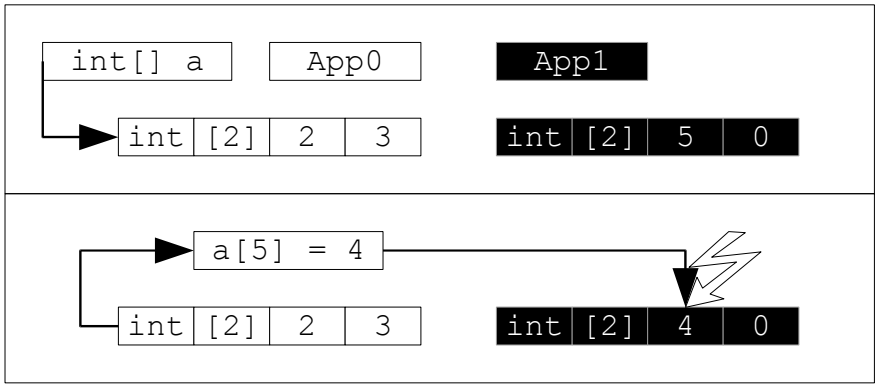

Abbildung 2.2: Beschreiben fremden Speichers

Nullprüfung verzichtet, könnte man mit dem Aufruf „fill(*null*,4711)“ wieder auf einen unvorhersagbar großen Speicherbereich an Adresse *null* zugreifen.

2.2.2 Speicheroperationen

Schreiboperationen haben immer globale Auswirkungen, da sie fremden Speicher und darin abgebildete Register, oder eigene Referenzen und Laufzeittypinformationen überschreiben könnten. Zu Letzterem finden sich Beispiele in Abschnitt 2.3.3.

Abbildung 2.2 zeigt, wie durch einen Vektorzugriff ein Objekt einer anderen Anwendung manipuliert wird. Die beiden vorderen Felder mit der Aufschrift „int [2]“ sollen Laufzeittypinformationen darstellen. Es handelt sich also um zwei Ganzzahlvektoren mit jeweils zwei Einträgen. Wird ungeprüft auf das fünfte Element des ersten Vektors zugegriffen, trifft dies den zweiten Vektor. Hier werden zwar nur primitive Daten verändert, dies reicht jedoch aus, um Fehler in „App1“ hervor zu rufen, deren Ursache nicht in „App1“ zu finden sind. Gleiches gilt auch für das Schreiben von Instanzfeldern, nur dass dabei der betroffene Speicherbereich vorhersagbar ist (siehe Abschnitt 2.3.2).

2.2.3 Nullprüfungen beim direkten Aufruf von Instanzmethoden

In vielen Fällen erlaubt es die statische Analyse, Methoden direkt aufzurufen. Beispielsweise, weil sie mit dem Schlüsselwort *final* versehen wurden, oder weil der dynamische Typ eines Objektes zur Übersetzungszeit ermittelt werden kann. Trotzdem wird vor dem Aufruf ein Nullcheck durchgeführt, um die Java-Konformität zu wahren. Außerdem gehen Methoden davon aus, dass ihre *this*-Referenz ungleich *null* ist, schließlich kann bei einem virtuellen Aufruf kein Laufzeittyp von *null* gelesen werden. Diese Nullprüfung kann nicht von der Hardware übernommen werden, da kein Feld der betroffenen Instanz gelesen wird. Welche Auswirkung es haben kann, diese Prüfung zu ignorieren, hängt

von den Zugriffen auf die *this*-Referenz ab, die in der aufgerufen Methode stattfinden. Die Auswirkungen sind also potentiell global. Spielen Konformität und Fehlererkennung jedoch keine Rolle, kann abgeschaltet werden, dass Methoden annehmen *this* sei „null-geprüft“. Das Ignorieren der Prüfung vor dem direkten Methodenaufruf hat nun keine Auswirkungen mehr, da sie in die Methode verlagert wurde. Somit kann sie abgeschaltet werden. Dies ergibt auf Plattformen Sinn, auf denen praktisch jede andere Nullprüfung durch die Hardware durchgeführt werden kann (siehe Abschnitt 2.3.2). Da die Prüfung der *this*-Referenz in der Methode durch die Hardware durchgeführt wird, entstehen keine Mehrkosten. So ist es möglich alle Nullchecks, die in Java vorkommen, abzuschalten.

2.3 Abschaltung von Laufzeitprüfungen unter Kenntnis des Speicherlayouts

In diesem Abschnitt wird beschrieben, welche Laufzeitprüfungen durch die Hardware übernommen werden können. Ferner wird untersucht, welche Prüfungen dank des Speicherverhaltens ohne Gefährdung der Isolation abgeschaltet werden können, obwohl - entgegen der Java-Spezifikation - keine Ausnahmebehandlung angestoßen wird.

Hierzu wird analysiert, wie der Mikrocontroller auf Speicherzugriffe reagiert und wie die zugegriffenen Bereiche vorhergesagt werden können.

2.3.1 Mögliches Verhalten bei Speicherzugriffen

Lese- und Schreibzugriffe können jeweils drei verschiedene Folgen haben.

Bei einem Lesezugriff kann entweder ein Trap ausgelöst werden, oder es wird ein Wert gelesen, der entweder vorhersagbar ist, oder nicht. Unter einem Trap wird hier jede Möglichkeit verstanden, eine Ausnahmebehandlung anzustoßen. Dies können beispielsweise Busfehler sein. Findet kein Trap statt, wird ein Wert gelesen. Kann sichergestellt werden, dass der betroffene Speicherbereich nicht beschrieben wird, kann der gelesene Wert eventuell vorhergesagt werden. Ein Beispiel hierfür wäre Flashspeicher. Kann dies nicht sichergestellt werden, muss davon ausgegangen werden, dass ein beliebiger Wert gelesen wird.

Bei einem Schreibzugriff kann entweder ein Trap ausgelöst werden, der Vorgang kann ignoriert werden, oder der betroffene Bereich wird beschrieben. Traps wurden bereits oben beschrieben. Wird kein Trap ausgelöst und kein Speicherzustand verändert, wurde der Schreibvorgang ignoriert. Ist dies nicht der Fall, wurde der Speicherinhalt verändert.

Da dieses Wissen dem Compiler nicht zur Verfügung steht, muss ein Weg gefunden werden, ihm mitzuteilen, welche Bereiche des Speichers sich wie verhalten.

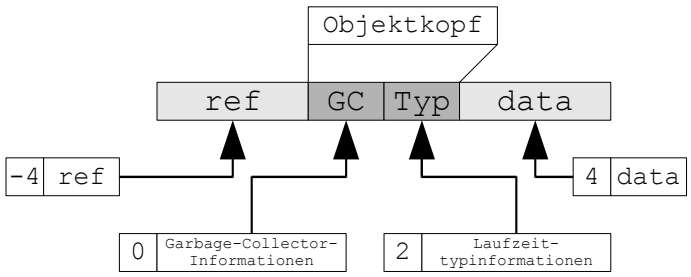

Abbildung 2.3: Objektlayout

2.3.2 Abschaltung von Nullprüfungen

Da es in Java keine wilden Zeiger gibt, zeigen Referenzen entweder auf ein gültiges Objekt, oder enthalten den Ausnahmewert *null*. Da KESO den Wert von *null* kennt, kann der Speicherbereich, der bei einem Zugriff auf *null* angesprochen wird, vorausberechnet werden. Hierbei ist zu beachten, dass nicht immer genau auf *null* zugegriffen wird. Die verschiedenen Instanzfelder haben verschiedene - aber feste - Offsets.

Die *null*-Referenz kann theoretisch beliebig verschoben werden. In den hier gezeigten Beispielen wird jedoch immer angenommen, dass *null* den Wert 0 hat.

Wurde der betroffene Speicherbereich ermittelt, kann der Nullcheck, je nach dem, wie sich der Bereich beim vorliegenden Zugriff verhält, abgeschaltet werden.

Folgende Instruktionen benötigen Nullchecks und würden ohne Prüfung auf den Speicher zugreifen: Zugriffe auf Instanzfelder würden ohne eine Prüfung auf den Speicher an Position *null* plus die relative Position des Feldes im Objekt zugreifen. Die Vektorgröße ist ein spezielles Feld einer Vektorinstanz. Sie wird bei Bereichsprüfungen gelesen, die daher ebenfalls einen Nullcheck benötigen. Virtuelle Methodenaufrufe lesen den Laufzeittyp einer Instanz, welcher ebenfalls ein spezielles Feld ist.

Die verschiedenen Fälle werden am Beispiel der Klasse *Bsp* erläutert. Diese ist definiert als:

```
class Bsp {
  Bsp ref;
  int data;
  void method() { println("Hello Bsp"); }
};
```

Für diese Klasse *Bsp* wird das in Abbildung 2.3 gezeigte Layout angenommen. Dieses Layout ist einem möglichen Objektlayout KESOs grob nachempfunden. Die Zahlen geben die Offsets der Felder an. Dahinter befindet sich deren Beschreibung.

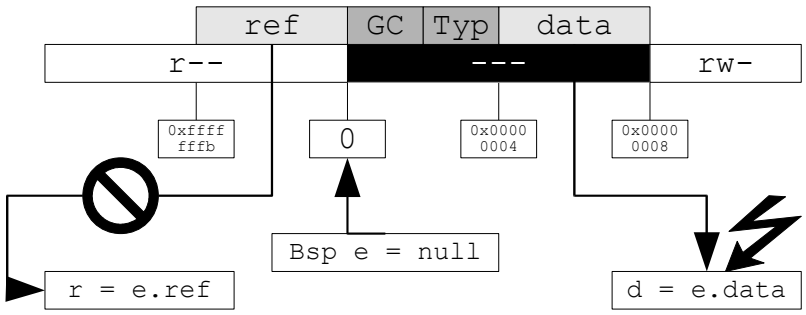

Abbildung 2.4: Traps beim Lesen von Instanzfeldern

Nullchecks beim Lesen von Instanzfeldern

Wird beim lesenden Zugriff auf den Speicher, der sich am Offset des Feldes befindet, ein Trap ausgelöst, kann die Nullprüfung immer entfernt werden. Die Erkennung von *null*-Referenzen bleibt erhalten, da die Ausnahmebehandlung nun durch den Trap angestoßen wird. Da der Zugriff auf das Feld verhindert wird, ist der Typ des Feldes nicht von Bedeutung. Weiter kann die Referenz, nach einem erfolgreichen Zugriff auf das Feld, als „geprüft“ markiert werden. Dies zeigt der statischen Analyse, dass die Referenz nicht den Wert *null* enthalten kann, da dies eine Ausnahmebehandlung verursacht hätte.

In Abbildung 2.4 interpretiert die Anwendung, durch die Anweisung „Bsp e = null“, die Speicheradresse 0 als Objekt vom Typ Bsp. Der Anwendung werden auf dem Bereich 0 bis 8 keine Zugriffsrechte eingeräumt. Das Lesen des Feldes *data* verursacht daher eine Unterbrechung. Anders verhalten sich im Beispiel Zugriffe auf ref. Um diese Nullprüfung zu entfernen, müssen weitere Annahmen getroffen werden.

Verursacht der Zugriff auf ein Feld keinen Trap, kann nur dann weiter optimiert werden, wenn die Java-Spezifikation vernachlässigt werden kann. Ist dies der Fall, muss nach dem Typ des Feldes unterschieden werden. Handelt es sich um primitive Daten, kann die Prüfung entfernt werden, da das Lesen falscher primitiver Daten nur lokale Auswirkungen hat (siehe Abschnitt 2.2.1). Das Lesen von Referenzfeldern hingegen hat globale Auswirkungen, also ist eine Laufzeitprüfung erforderlich.

Eine Ausnahme wäre, wenn von dem betroffenen Speicher ein vorhersagbarer Wert gelesen werden würde, der als Wert einer Referenz zulässig ist. Der einzige vorhersagbare Wert, auf den dies zutrifft, ist *null*. Wird vom betroffenen Speicher konstant *null* gelesen, könnte das Feld ohne Prüfung gelesen werden, ohne die Isolation zu gefährden. Abbildung 2.5 zeigt, wie das Feld ref, das von *null* gelesen wird, selbst wieder auf *null* zeigt.

Probleme treten dann auf, wenn die dadurch entstehenden *null*-Referenzen dort auftauchen, wo sie von der statischen Analyse nicht erwartet werden. Dazu wird die Klasse „NotNullExample“ aus Abbildung 2.6 betrachtet. Die statische Analyse könnte feststellen, dass das Feld $field$ niemals *null* sein kann und deshalb in der Funktion *test* davon

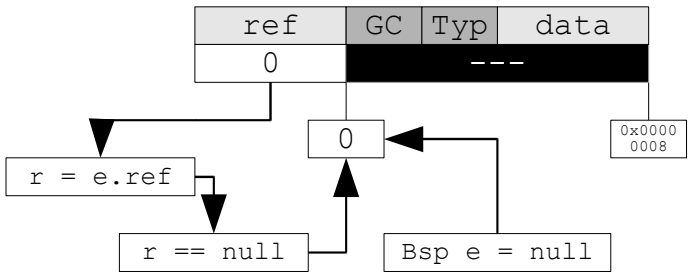

Abbildung 2.5: Lesen der Referenz *null* von Objekten an Adresse *null*

```
final class NotNullExample {
  private final Object[] field = new Object[1];
  public static void test(NotNullExample ex) {
    Object[] objs = ex.field;
    objs[0].wait(); //Moeglicher Verlust der Speichersicherheit
  }
};
```

Abbildung 2.6: NotNullExample

ausgehen, dass *objs* nicht auf *null* geprüft werden muss. Wenn *objs* aber nun den Wert *null* trägt, da *ex* gleich *null* ist, geht dadurch die Speichersicherheit verloren. Momentan schließt die statische Analyse jedoch bei keinem Referenzfeld den Wert *null* aus. Auch existiert bis jetzt keine Zielplattform, bei der der Wert *null* konstant gelesen wird.

Nullchecks beim virtuellen Aufruf von Methoden

Beim Interface- oder virtuellen Aufruf wird zuerst der Laufzeittyp der Instanz gelesen. Verursacht dieser Lesevorgang bei einer *null*-Referenz einen Trap, ist keine Laufzeitprüfung erforderlich und die Referenz kann als „geprüft“ markiert werden. Dies wäre in Beispiel 2.4 der Fall.

Gibt es einen vorhersagbaren Wert, der von *null*-Referenzen als Laufzeittyp gelesen wird, kann dieser verwendet werden, um eine Ausnahmebehandlung anzustoßen. Dies erklärt sich durch die Art, wie virtuelle Methoden aufgerufen werden. Hierzu wird angenommen, dass es zu jeder virtuellen Methode einen Vektor gibt, in dem die Adressen der verschiedenen Überschreibungen der Methoden gespeichert sind. Weiter handelt es sich beim Laufzeittyp um einen Index in diesen Vektor. Soll nun eine Methode aufgerufen werden, wird der Laufzeittyp des Objektes gelesen und die Methode an der entsprechenden Position im Methodenvektor aufgerufen. Als Beispiel dient die Klasse „Derived“, die

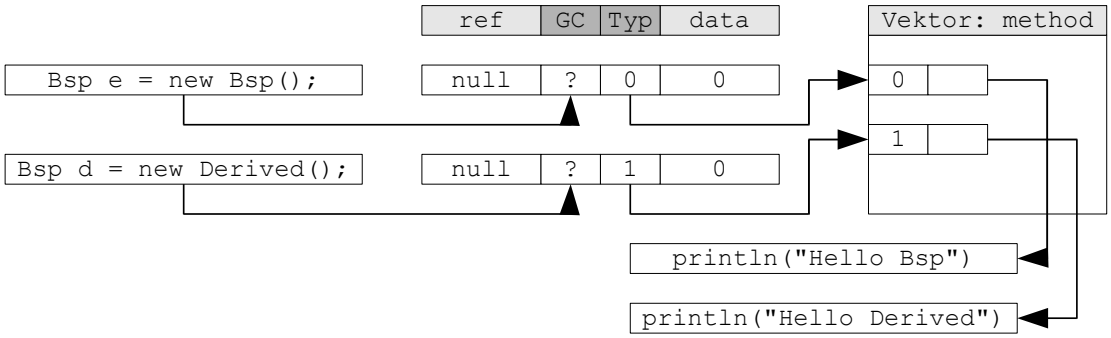

Abbildung 2.7: Aufruf virtueller Methoden

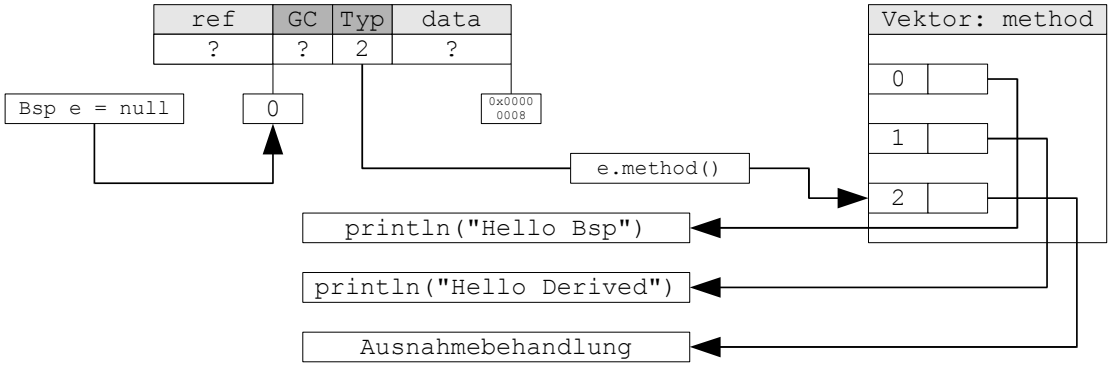

Abbildung 2.8: Ausnahmebehandlung durch vorhersagbare Laufzeittypinformationen

von „Bsp“ abgeleitet wurde und die Methode „method“ überschreibt.

```
class Derived extends Bsp {
  void method() { println("Hello Derived"); }
};
```

Der Klasse „Bsp“ wird als Typindex „0“ zugewiesen; „Derived“ erhält „1“. Abbildung 2.7 soll verdeutlichen, wie der Typindex in den Vektor der jeweiligen Methode verweist. Gibt es nun den oben genannten, vorhersagbaren Wert, kann an dieser Position ein Eintrag im Vektor aller Methoden angelegt werden. Dieser Eintrag zeigt dann auf eine Methode, die eine Ausnahmebehandlung verursacht (siehe Abbildung 2.8).

Im Beispiel wurden die Indizes „0“, „1“ und „2“ verwendet. In der Realität könnte von *null* ein Wert gelesen werden, der wesentlich weiter von den Typindizes der Typen entfernt ist, die diese Methode enthalten. Ist die daraus resultierende Vergrößerung des Methodenvektors nicht tolerierbar, kann dieses Vorgehen nicht angewandt werden.

Vor einem weiteren Problem steht man dann, wenn die Werte, die von der entsprechenden Stelle gelesen werden, zwar über die Laufzeit konstant sind, jedoch erst zur Bindezeit bekannt werden. So könnte sich dort Code befinden, der an eine noch unaufgelöste Marke springt.

Kann der von *null* gelesene Typ nicht vorhergesagt werden, ist eine Laufzeitprüfung erforderlich.

Nullprüfungen beim Schreiben von Instanzfeldern

Beim Beschreiben von Instanzfeldern ist keine Unterscheidung in Referenzen und primitive Daten nötig, da Schreibvorgänge immer globale Auswirkungen haben können.

Verursacht das Schreiben auf den betroffenen Speicherbereich um *null* einen Trap, ist keine Prüfung durch die Software nötig und die Referenz kann für die statische Analyse als „geprüft“ markiert werden. Dies wäre beim Setzen des Wertes von *data* und ref in Beispiel 2.4 der Fall.

Wird eine Verletzung der Java-Spezifikation erlaubt, kann die Prüfung deaktiviert werden, wenn der betroffene Speicher den Schreibvorgang ignoriert. Dies sorgt zwar nicht für den Aufruf eine Ausnahmebehandlung, globale Auswirkungen der Operation werden aber verhindert.

Anderenfalls ist eine Laufzeitprüfung erforderlich, da durch das Beschreiben Register angesprochen werden könnten, die in den Speicher eingeblendet werden.

Nullchecks beim Lesen der Vektorgröße

Beim Lesen der Vektorgröße „length“ gilt ähnliches wie beim Lesen von Referenzen. Wird beim Lesen des entsprechenden Speicherbereichs ein Trap verursacht, kann die Prüfung entfernt werden. Nach einem erfolgreichen Zugriff kann die Referenz auf den Vektor als „geprüft“ markiert werden.

Wird beim Lesen des Bereichs kein Trap verursacht, kann die Prüfung nicht entfernt werden, ohne die Java-Spezifikation zu verletzen. Weiter hat das Lesen einer falschen Vektorlänge globale Auswirkungen. Darf die Java-Spezifikation verletzt werden, gibt es jedoch eine Ausnahme. Die Prüfung kann dann entfernt werden, wenn vorhergesagt werden kann, dass als Vektorlänge von *null* nur der Wert 0 gelesen wird. Der Vektor an Stelle *null* erscheint dann leer, wodurch alle Zugriffe, die die Isolation gefährden, durch die Bereichsprüfung abgefangen werden.

Wird weder ein Trap verursacht noch die Konstante 0 gelesen, ist eine Nullprüfung erforderlich.

Nullchecks beim Lesen von Vektorelementen

Ob ein Nullcheck beim Laden von einem Vektor ausgeführt werden muss hängt davon ab, ob die Bereichsprüfung durchgeführt wird. Wird Konformität zu Java verlangt, muss die Bereichsprüfung immer durchgeführt werden. Wird dies nicht verlangt, muss die

Bereichsprüfung nur dann durchgeführt werden, wenn es sich bei den gelesenen Daten um Referenzen handelt, da das Lesen falscher Referenzen globale Auswirkungen hat.

Wird keine Bereichsprüfung durchgeführt, ist auch keine Überprüfung der Referenz erforderlich. Wichtig ist dann, dass die Referenz auf keinen Fall als „durch die Hardware geprüft“ markiert wird, nur weil ein Zugriff auf die Vektorgröße einen Trap verursachen würde. Findet die Bereichsprüfung statt, gilt für den Nullcheck das Gleiche, wie für den Nullcheck bei Lesen der Vektorgröße.

Nullchecks beim Schreiben von Vektorelementen

Schreibvorgänge können globale Auswirkungen haben. Daher sind Bereichsprüfungen beim Beschreiben von Vektorelementen immer erforderlich. Ob ein Nullcheck erforderlich ist und ob er zur Fehlererkennung beiträgt, hängt davon ab, ob der Nullcheck zum Lesen der Vektorgröße erforderlich ist.

2.3.3 Abschaltung von Bereichsprüfungen

Die maximale Größe des Speicherbereichs, der durch einen Vektorzugriff angesprochen werden kann, ist berechenbar. Sie ergibt sich aus der Bitbreite des Feldes, das die Vektorgröße speichert und aus der Größe des Elementtyps. So kann beispielsweise bei einem Zugriff auf einen 32-Bit-Integer-Vektor mit einem 16-Bit-Index nur ein Speicherbereich von etwa 256 Kibibyte angesprochen werden. Um nun Bereichsprüfungen abschalten zu können, sind folgende Kenntnisse über das Speicherlayout von Nöten.

Um Zugriffe auf *null*-Referenzen zu vermeiden, muss sich an dieser Adresse ein Speicherbereich befinden, der Traps verursacht. Dieser Bereich muss mindestens so groß sein, wie die maximale Größe des durch einen Vektorzugriff erreichbaren Bereiches zuzüglich der Größe des Vektorkopfes. Ein Beispiel hierfür ist der TC1796. Hier verursachen Zugriffe auf die unteren zwei Gibibyte des Speichers einen Trap.

Weiter muss ausgeschlossen werden, dass bei einem Lesezugriff ungültige Referenzen gelesen, oder bei einem Schreibzugriff Referenzen illegal überschrieben werden. Auch Vektor- oder Objektköpfe dürfen nicht überschrieben werden. Dies bedeutet, jeder Vektor muss den maximalen Speicherbereich für seinen Datentyp reservieren, also beispielsweise 256 kiB. Würde sich hinter einem kleineren Vektor ein weiterer Vektor befinden, könnte der Vektorkopf des hinteren Vektors manipuliert werden. Schreibzugriffe könnten Elemente des hinteren Vektors mit Werten vom Datentyp des vorderen Vektors überschrieben. Werden dabei Referenzen durch primitive Daten überschrieben entstehen wilde Referenzen. Werden Referenzen durch Referenzen mit inkompatiblem Typ überschrieben, geht die Typsicherheit ebenfalls verloren. Lesezugriffe könnten Elemente des hinteren Vektors als Werte vom Datentyp des vorderen Vektors interpretieren. Hierbei entstehen ebenfalls wilde Referenzen, wenn primitive Daten als Referenzen interpretiert werden.

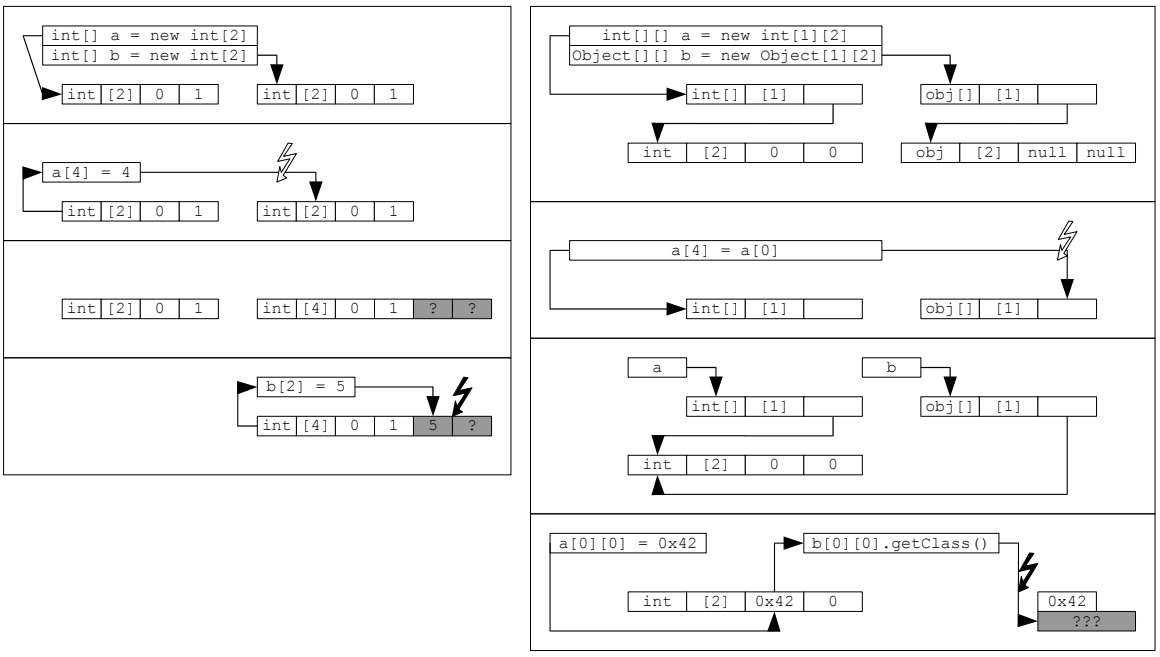

Abbildung 2.9: Überschreiben von Referenzen und RTTI

Beispiel 2.9 (links) zeigt die Manipulation des Vektorkopfes durch einen davor liegenden Vektor. Es werden zwei Ganzzahlvektoren der Größe 2 angelegt. Vektor b befindet sich hier im Speicher etwas hinter Vektor a. Wird der Schreibzugriff $a[4] = 4$ ungeprüft durchgeführt, überschreibt er die Vektorgröße „length“ von Vektor b. Dieser erscheint dann als Vektor der Größe 4. Dadurch können Zugriffe auf b - auch wenn sie überprüft werden - auf Speicherzellen zugreifen, die dieser Anwendung nicht gehören.

In Beispiel 2.9 (rechts) wird eine Referenz auf einen Referenzvektor durch eine Referenz auf einen Ganzzahlvektor überschrieben. Hier werden zwei Matrizen - bei denen es sich in Java um Vektoren von Vektoren handelt - angelegt. So ist a ein Vektor, der eine Referenz auf einen Int-Vektor enthält. Analog ist b ein Vektor, der eine Referenz auf einen Vektor enthält, der wiederum Referenzen auf Instanzen vom Typ *Object* enthält. Wird der Schreibzugriff $a[4] = a[0]$ ungeprüft durchgeführt, wird die in b enthaltene Referenz durch die in a enthaltene Referenz überschrieben. Nun ist ein Vektor sowohl als Referenzvektor als auch als Ganzzahlvektor zugreifbar. Damit liegt ein Vektor von wilden Referenzen vor, auf die durch b zugegriffen werden kann.

Auf Grund dieser Einschränkungen, ist das Abschalten von Bereichsprüfungen durch Kenntnis des Speicherlayouts in der Praxis nur in seltenen Ausnahmefällen sinnvoll. Schließlich rühren die kleinen Vektorindizies, die die Abschätzung des erreichbaren Bereichs ermöglichen, daher, dass Speicherplatz auf solchen eingebetteten Systemen eine knappe Ressource ist. Für jeden Vektor die maximale Größe zu reservieren, würde den

knappen Speicher verschwenden.

3 Implementierung

Um zu entscheiden, welche Prüfungen notwendig sind und welche verworfen werden können, werden folgende Schritte durchlaufen. Beim Lesen der Systembeschreibung wird auch die Speicherbeschreibung gelesen, die für das aktuelle System angegeben wurde. Sobald der Compiler bereit ist, den gelesenen Javacode zu C-Code zu übersetzen, wird einmalig für jedes Feld aller Klassen entschieden, ob bei einem Zugriff eine Laufzeitprüfung erforderlich ist, oder nicht. Hierzu werden die Offsets berechnet und anhand der Speicherbeschreibung geprüft, welcher Speicherbereich bei der Dereferenzierung von *null* betroffen ist. Während der C-Code erzeugt wird, wird jedes mal, wenn eine Null- oder Bereichsprüfung erzeugt werden soll, nachgeschlagen, ob diese notwendig ist. Ist sie nicht notwendig, wird sie verworfen.

Eine Verschiebung der *null*-Referenz wird nicht Implementiert, da dies auf den betrachteten Architekturen nicht notwendig ist, oder kaum Vorteile mit sich bringt. *null* trägt daher immer den Wert 0. Aus dem gleichen Grund wird darauf verzichtet, die vorhersagbaren Typinformationen an *null* dazu zu verwenden, um eine Ausnahmebehandlung bei virtuellen Methodenaufrufen anzustoßen.

3.1 Offsetberechnung

Das Berechnen der Offsets der Instanzfelder erlaubt es, herauszufinden, auf welchen Speicherbereich bei der Dereferenzierung von *null* zugegriffen wird (siehe Abschnitt 2.3). Die Offsets hängen von Anordnung, Größe und Alignment der Felder ab. Treten Fehler bei der Berechnung der Offsets auf, gefährdet das die Speichersicherheit. Dies ist zum Beispiel der Fall, wenn von *null* gelesene Konstanten für den dynamischen Typ, die Vektorgröße oder Referenzen berechnet werden sollen. Um Berechnungsfehler zu erkennen wurden Übersetzungszeitprüfungen eingefügt, die den C-Compiler stoppen, falls Offsets falsch berechnet wurden. Diese Prüfungen sind mit denen vergleichbar, die in „C++11“ durch das Schlüsselwort *static_assert* gesetzt werden.

3.1.1 Anordnung und Größe der Felder

KESO übersetzt Javaklassen zu C-Code. Die Instanzfelder werden in einer Struktur („struct“) angeordnet. Die Reihenfolge der Felder orientiert sich an der SableVM [5]. Jede Struktur erhält einen Objektkopf, der den dynamischen Typ und Informationen für die automatische Speicherbereinigung enthält. Referenzen zeigen immer auf einen

solchen Objektkopf. Felder mit einem primitiven Datentyp werden unter dem Kopf angehängt, haben also positive Offsets. Referenzfelder wachsen nach oben, um die Arbeit des Garbage-Collectors zu erleichtern. Sie werden in umgekehrter Reihenfolge über dem Kopf eingefügt und haben damit negative Offsets. Die Struktur einer abgeleiteten Klasse enthält die Felder der Struktur der Basisklasse. Die neuen Felder werden unter den alten Feldern angehängt, falls es sich um primitive Daten handelt, oder oben eingefügt, falls es sich um Referenzen handelt. Die Reihenfolge der Felder der C-Struktur wird vom C-Compiler nicht verändert: „Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared.“ [1].

Da der Standard der Sprache C die Größe von Ganzzahltypen wie „int“ nicht vorgibt, werden einige Javatypen auf Typen mit fester Größe des Headers „stdint.h“ abgebildet.

3.1.2 Alignment

Der C-Standard [1] schreibt kein Alignment vor. Es ist von der Zielplattform abhängig. Für diese Arbeit wurde angenommen, dass sich die Ausrichtung an der Größe eines Maschinenwortes, also eines Zeigers, orientiert. Ist das Feld kleiner als ein Zeiger, wird es auf eine Adresse ausgerichtet, die durch seine eigene Größe teilbar ist. Ist es größer oder gleich, wird es auf Machinenwortgröße ausgerichtet.

Diese Regeln treffen nicht auf alle Plattformen zu. So gibt das Microsoft Windows ABI auf dem Intel 386 vor, dass 64-Bit-Integer auf durch acht teilbare Adressen ausgerichtet werden müssen [12, durch Tests überprüft]. Auch für den Tricore TC1796, der KESOs Hauptplattform darstellt, heißt es „4. Double-word accesses must be aligned to addresses with address lines $A[2:0] = 000_B$.“ [6, Abschnitt 6.1.3]. Tests mit dem verwendeten Compiler (tricore-gcc 3.4.5) haben jedoch ergeben, dass 64-Bit-Integer auf 4-Byte-Adressen ausgerichtet werden. Zugriffe auf solche Felder werden trotzdem auf 8-Byte-Lade- beziehungsweise Speicherbefehle abgebildet. Microsoft Windows wiederum ist keine Zielplattform für KESO. Auf dem ARMv7-M ist ebenfalls mit einem 4-Byte-Alignment zu rechnen. So heißt es zu Doublewords: „Is a 64-bit data item. Doublewords are normally at least word-aligned in ARM systems.“ [2].

Der wichtigste Grund für obige Annahme ist aber, dass der gesamte Aufbau der C-Strukturen nicht mehr funktionieren würde, wenn es Felder gäbe, die ein größeres Alignment besitzen als Referenzen. Nimmt man beispielsweise die Klassen A und B und ihre zugehörigen C-Strukturen auf einem 32-Bit-System an, wie sie in Abbildung 3.1 gezeigt werden. In $structB$ hätte das Feld a - relativ zum Objektkopf - ein Offset von vier. In $structA$ hätte dieses Feld ein Offset von acht. Wird nun, wie in der Klassenmethode $B :: test$, durch eine Referenz vom statischen Typ A auf ein Objekt mit dynamischem Typ B, auf das Feld a zugegriffen, würde es an Offset acht vermutet werden, obwohl es sich bei dieser Instanz an Offset vier befände. Abbildung 3.2 stellt dies dar.

```
class A {
  long a;
};

class B extends A {
  B b;
  static void test() {
    A x = new B();
    x.a = -5;
  }
};

struct A {
  int32_t OBJECT_HEAD; // offset: 0
  int64_t a;           // offset: 8
};

struct B {
  void *b;             // offset: 0
  int32_t OBJECT_HEAD; // offset: 4
  int64_t a;           // offset: 8
};
```

Abbildung 3.1: Zwei Java-Klassen und ihre entsprechenden C-Strukturen

3.2 Speicherbeschreibung

3.2.1 Erster Ansatz

Um herauszufinden, wie bestimmte Speicherbereiche auf Lese- oder Schreibzugriffe reagieren, wurde anfangs ein Parser für Linkerscripts geschrieben. Diese Scripts beschreiben jedoch nur, an welchen Stellen bestimmte Segmente platziert werden sollen. Ist ein Speicherbereich nicht im Linkerscript beschrieben bedeutet das nicht, dass jeder Zugriff auf diesen Bereich einen Trap verursacht. Beispielsweise könnten sich in einem solchen Bereich Register befinden, die in den Speicher eingeblendet werden. Auch verursachen Schreibzugriffe auf einen Bereich, in dem der Linker nur Readonly-Segmente platzieren soll, nicht zwangsläufig einen Trap.

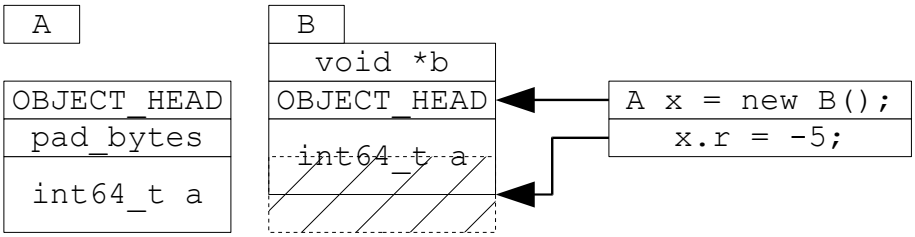

Abbildung 3.2: Doubleword-Alignment

3.2.2 Speicherbeschreibung in der KESO-Anwendungskonfiguration

Um zu einer aussagekräftigen Speicherbeschreibung zu kommen, wurde es ermöglicht, in der KESO-Konfigurationsdatei zu jedem System eine Speicherbeschreibung anzugeben. Die Speicherbeschreibung besteht aus der Beschreibung mehrerer Speicherblöcke. Jeder Block hat die Eigenschaften *Origin*, *Length*, *Read* und *Write*. *Origin* und *Length* geben an, wo der Speicherbereich anfängt und wo er endet.

Read gibt das Verhalten bei Lesezugriffen an. Folgende Werte sind möglich: „trap“, um anzuzeigen, dass ein Zugriff einen Trap auslöst. Eine Folge von Werten zwischen 0 und 255, um festzulegen, dass von diesem Block konstante Werte gelesen werden. Die Folge muss entweder einen Wert enthalten, was bedeutet, dass alle Bytes des Blocks diesen Wert haben, oder so viele Werte, wie Bytes in diesem Block vorhanden sind. Der dritte Wert „unspec“ bedeutet, dass für diesen Bereich nicht bekannt ist, welche Auswirkungen Lesezugriffe haben. Deshalb geht die Implementierung des graduellen softwarebasierten Speicherschutzes (GradMP) davon aus, dass hier zufällige Werte gelesen werden.

Write gibt das Verhalten bei Schreibzugriffen an. Auch hier gibt es wieder den Wert „trap“. In Analogie zum konstanten Lesezugriff, gibt es den Wert „ignore“, der bedeutet, dass Schreibzugriffe ignoriert werden. Es wird kein Speicherinhalt verändert und kein Trap ausgelöst. Wird *Write* der Wert „unspec“ zugewiesen, werden Zugriffe auf diesen Bereich verhindert, weil dabei fremder Speicher überschrieben oder eingeblendete Register angesprochen werden könnten.

3.2.3 Beispiele für Speicherbeschreibungen

Speicherbeschreibungen zu Architekturen, die von KESO unterstützt werden, werden mit KESO ausgeliefert und müssen vom Anwender nur noch in die Systembeschreibung eingebunden werden. Beim Erstellen der Speicherbeschreibung ist vor allem der Bereich um die Adresse 0 von Belang, um Nullchecks zur Laufzeit zu vermeiden.

TC1796 Der TC1796 löst bei Zugriffen auf die untersten acht Bytes einen speziellen „MPN-Trap“ aus, um anzuzeigen, dass ein Nullzeiger dereferenziert wurde. Der Rest der

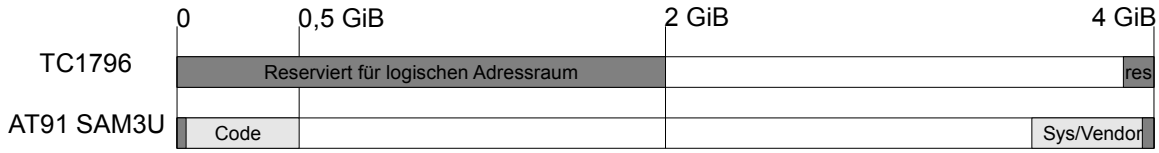

Abbildung 3.3: Wichtige Speicherbereiche

unteren zwei Gibibyte sind für Modelle mit einer MMU, die einen logischen Adressraum bieten, reserviert und verursachen bei einem Zugriff einen Busfehler, also eine Ausnahmebehandlung. Zugriffe auf den Bereich $F8800000_{16}$ bis einschließlich $FFFFFFFF_{16}$ haben ebenfalls einen Busfehler zur Folge [6, Tabelle 9-3]. Damit ist ein genügend großer Bereich um die Adresse 0 bekannt, der Traps auslöst, was in der Praxis ausreicht um alle Nullchecks durch die Hardware durchführen zu lassen. Die sich ergebende Speicherbeschreibung wird in Abbildung 3.4 gezeigt. In Abbildung 3.3 werden die beschriebenen Speicherbereiche veranschaulicht.

```
MemoryDescription {
  reserved_virtual_address_space = {
    origin = 0;
    length = 0x80000000;
    read = "trap";
    write = "trap";
  }
  reserved_space_high = {
    origin = 0xf8800000;
    length = 0x07800000;
    read = "trap";
    write = "trap";
  }
}
```

Abbildung 3.4: Speicherbeschreibung des TC1796

i386-Linux Linux auf einem i386 dient KESO als eine Testplattform. Da man bei C-Programmen auf dieser Plattform davon ausgeht, dass Zugriffe auf die Adresse 0 das Signal „SIGSEGV“ auslösen, wurde für die Seite an dieser Adresse, also den unteren vier Kibibyte des Adressraumes, Trap-Verhalten eingetragen. Für Linux-Anwendungen befinden sich alle Adressen ab dem Wert „PAGE_OFFSET“ im Kernel-Adressraum. Auf i386-Computern liegt PAGE_OFFSET üblicherweise bei drei Gibibyte, ist aber konfigu-

rierbar. Da der Kernel-Adressraum mindestens eine Seite groß sein muss, wird die höchste Seite mit Trap-Verhalten markiert. Dies reicht, wie beim TC1796, aus.

ARMv7-M: AT91 SAM3U Die Informationen über das Speicherverhalten des ARMv7-M [2] sind zu unspezifisch, um für den graduellen Schutz verwendet zu werden. Deshalb wurde im Speziellen der ARMv7-M-basierte AT91 SAM3U [3] betrachtet.

Der Adressraum des ARMv7-M [2] wird in acht Partitionen von jeweils 0,5 GiB unterteilt. Interessant sind wieder nur die Partition an Adresse 0 und die Partition an der höchsten Adresse.

An Adresse 0 befindet sich die Partition „Code“, die typischerweise aus Flash- und ROM-Speicher besteht. Genau an Adresse 0 findet man die Vektortabelle für Boot-Code. Beim AT91 SAM3U erstreckt sich dieser Speicherbereich von Adresse 0 bis 00080000_{16}. Tests lassen vermuten, dass - bei voreingestellter Konfiguration - das Lesen auf diesem Bereich erlaubt ist, das Schreiben jedoch zu einem Trap führt. Die gelesenen Werte können nicht vorhergesagt werden, da die Vektortabelle Sprünge an Marken enthält, die erst zur Bindezeit bekannt sind.

Von $E0000000_{16}$ bis $FFFFFFFF_{16}$ erstreckt sich die Partition „System“. Bis $E00FFFFF_{16}$ findet man den „Private Peripheral Bus“, also beispielsweise Steuerregister für die MPU. Der „Vendor“-Bereich ab $E0100000_{16}$ hängt vom Verkäufer ab. Beim AT91 SAM3U ist dieser Bereich reserviert. Tests zeigen, dass von diesem Bereich konstant 0 gelesen wird; Schreibzugriffe werden ignoriert.

Auch die bisherigen Informationen können nur teilweise genutzt werden. Um es zu ermöglichen, dass Laufzeitprüfungen von der Hardware übernommen werden können, müssen Zugriffe auf diese Speicherbereiche einen Trap verursachen.

Schreibzugriffe auf den „Boot-Code“ verursachen bereits einen Trap. Das ergibt Sinn, da ROM- und Flashspeicher nicht zur Laufzeit überschrieben werden kann. Lesezugriffe auf Code zu verbieten kann problematisch sein. So könnten sich Konstanten wie Strings im Codebereich befinden. Unprivilegierten Anwendungen können die Leserechte entzogen werden, da der Boot-Code nur im privilegierten Modus ausgeführt wird. Tests zeigen, dass keine Komplikationen auftreten, wenn nach dem Bootvorgang Lese- und Schreibrechte auf den Bootcode entzogen werden.

Zugriffsrechte auf den Bereich „Vendor“ können komplett entzogen werden. Da nicht spezifiziert wird, was sich in diesem Bereich befindet, kann auch kein Programm auf diesen Bereich zugreifen wollen. Tests bestätigen dies.

Die Zugriffsrechte auf diese beiden Bereiche zu entziehen, verbraucht zwei der acht MPU-Regionen. Eine Region wird auf Adresse 0 gesetzt. Ihre Größe beträgt 2^{19} Bytes. Das „Execute Never Bit“ wird gelöscht, dass der Code ausführbar bleibt. Die Zugriffsrechte sind „Kein Zugriff“. Bei dieser Region ist zu beachten, dass sie einen höheren Index besitzt als die Code-Region, die sich ebenfalls an Adresse 0 befindet. Die andere Region müsste auf $E0100000_{16}$ gesetzt werden. Dies ist jedoch keine Zweierpotenz. Daher wird sie auf Adresse $FFF80000_{16}$ gesetzt. Die Größe ist wieder 2^{19} Bytes. Dies ist für die

Größe von Java-Klassen völlig ausreichend. Das „Execute Never Bit“ kann gesetzt werden. Die Zugriffsrechte sind wieder „Kein Zugriff“. Abbildung 3.5 zeigt die resultierende Speicherbeschreibung. In Abbildung 3.3 werden die beschriebenen Speicherbereiche veranschaulicht.

```
MemoryDescription {
  boot_code_area = {
    origin = 0;
    length = 0x080000;
    read = "trap";
    write = "trap";
  }
  reserved_vendor_area = {
    origin = 0xfff80000;
    length = 0x00080000;
    read = "trap";
    write = "trap";
  }
}
```

Abbildung 3.5: Speicherbeschreibung des AT91 SAM3U

4 Testergebnisse

4.1 Erfolgskriterien

Ziel dieser Arbeit ist es, den Rechenaufwand, den Laufzeitprüfungen verursachen, zu vermindern. Hierzu wird auf den verschiedenen Stufen des graduellen Schutzes gemessen, wie sich die Laufzeit des Programms ändert und welcher Anteil der Prüfungen eingespart wird.

Weiter ist Speicherplatz auf eingebetteten Systemen eine knappe Ressource. Daher wird gemessen, wieviel Speicherplatz durch das Verwerfen von Prüfungen gespart wird.

4.2 Testumgebung

Zum Test wird auf einem Tricore TC1796 der Benchmark CDx [10] ausgeführt. Dieser Java-Echtzeit-Benchmark führt mehrfach Kollisionserkennungen auf einem simulierten Radar durch. Der Tricore ist derzeit die Hauptzielplattform von KESO. Sein Speicherlayout sollte es erlauben, alle Nullprüfungen zu entfernen, bei denen auf Instanzfelder zugegriffen wird. Als C-Übersetzer wird der tricore-gcc 3.4.5 verwendet. Die voreingestellte Optimierungsstufe für dieses Projekt ist „-O3“.

4.3 Anteil der verworfenen Prüfungen

Die Tabelle in Abbildung 4.1 zeigt welche Prüfungen bei welcher Optimierung verworfen wurden. In den oberen drei Zeilen steht, welche Optimierungen aktiviert wurden. „GradMP“ gibt an auf welcher Stufe der graduelle softwarebasierte Schutz angewendet wurde. „0“ bedeutet GradMP ist inaktiv. „1“ bedeutet GradMP verwirft alle Nullprüfungen, die durch die Hardware übernommen werden können. „2“ bedeutet, dass alle Prüfungen entfernt werden, die nicht vor globalen Auswirkungen schützen. In den folgenden Tabellen werden zusätzlich „3“, „3N“ und „3B“ auftauchen. „3N“ veranlasst GradMP alle Nullprüfungen zu entfernen. „3B“ entfernt alle Bereichsprüfungen. „3“ entfernt beide Prüfungen. Ein „x“ in der Zeile „statische Analyse“ bedeutet, dass es dieser erlaubt wird Prüfungen zu verwerfen. Die Option „no_valid_this_ptr“ sorgt dafür, dass innerhalb von Methoden nicht mehr angenommen wird, dass die *this*-Referenz nullgeprüft sei. Weiter unten wird die Summe über alle Null- beziehungsweise Bereichsprüfungen angezeigt. Bei der Summe der Nullprüfungen ist zu beachten, dass manche

GradMP	0	0	1	1	2	2	2	2
statische Analyse	-	x	-	x	-	-	x	x
no_valid_this_ptr	-	-	-	-	-	x	-	x
Bereichsprüfungen bei Lesezugriffen	72	72	72	72	29	29	29	29
... davon auf Referenzen	29	29	29	29	29	29	29	29
Bereichsprüfungen bei Schreibzugriffen	53	37	53	37	53	53	37	37
... davon auf Referenzen	15	14	15	14	15	15	14	14
Nullprüfungen vor Lesezugriffen auf Felder	546	55	3	3	3	3	3	3
Nullprüfungen vor Schreibzugriffen auf Felder	234	3	0	0	0	0	0	0
Nullprüfungen vor dem Lesen der Vektorgröße	57	17	0	0	0	0	0	0
Nullprüfungen vor virtuellen Methodenaufrufen	101	40	0	0	0	0	0	0
Nullprüfungen vor Schnittstellenaufrufen	86	23	0	0	0	0	0	0
Nullprüfungen vor direkten Methodenaufrufen	483	219	483	219	483	0	219	0
Nullprüfungen (Summe)	1605	395	486	222	486	3	222	3
Bereichsprüfungen (Summe)	125	109	125	109	82	82	66	66
Anteil der entfernten Nullprüfungen (in %)	0	75,39	69,72	86,17	69,72	99,81	86,17	99,81
Anteil der entfernten Bereichsprüfungen (in %)	0	12,8	0	12,8	34,4	34,4	47,2	47,2

Abbildung 4.1: Details: verworfene Prüfungen

Bereichsprüfungen eine Nullprüfung beinhalten. In den letzten Zeilen wird der Anteil der verworfenen Prüfungen angezeigt. Abbildung 4.2 veranschaulicht diese Ergebnisse. Dabei zeigt das jeweilige Tortenstück an, wieviele Prüfungen durch die jeweilige Technik verworfen wurden.

Nullprüfungen Die statische Analyse allein verwirft bereits 75,4% der Nullprüfungen. Sie kann jedoch meistens keine Prüfungen verwerfen, die beim Zugriff auf Referenzen entstehen, die noch nie dereferenziert wurden. GradMP ist dazu in der Lage, solange der Zugriff auf eine Nullreferenz einen Trap verursacht.

Ohne die statische Analyse können durch GradMP 69,7% der Nullprüfungen verworfen werden. Hiervon sind alle Lade- und Schreiboperationen betroffen. Diese Operationen verursachen bei der Dereferenzierung von *null* einen Trap und können daher ohne Verletzung der Java-Spezifikation abgeschaltet werden. Die übrigen Nullprüfungen entstehen durch nicht-virtuelle Methodenaufrufe. Solche Aufrufe greifen nicht auf den Datenspeicher zu und verursachen daher keinen Trap.

Wird die statische Analyse zusätzlich zu GradMP aktiviert, können insgesamt 86,2% der Prüfungen verworfen werden. Der Zuwachs erklärt sich dadurch, dass direkte Methodenaufrufe auf Referenzen, die bereits dereferenziert wurden, nicht mehr geprüft werden.

Nun wird die Option „no_valid_this_ptr“ aktiviert und die Verletzung der Java-Spezifikation erlaubt. Dadurch können auch Nullprüfungen vor direkten Methodenaufrufen deaktiviert werden, wodurch annähernd alle Nullchecks (99,8%) verworfen werden. Die verbleibenden Nullchecks entstehen durch Zugriffe auf Felder besonderer Klassen. Diese besonderen Klassen bieten eine Schnittstelle zum Betriebssystem an. So hat die

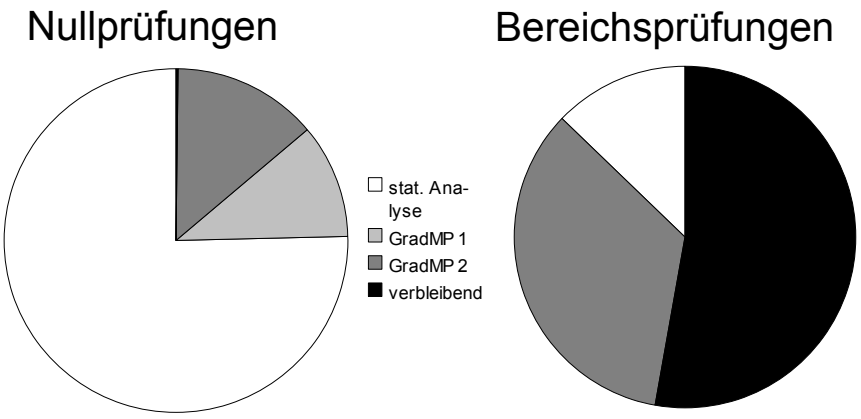

Abbildung 4.2: Verworfene Prüfungen

Klasse „Thread“ ein besonderes Feld namens „_task_id“, welches offenbar die Identifikationsnummer des zugehörigen Fadens speichert. Ähnliches gilt für die Klasse „Alarm“. Diese Felder werden nicht wie normale Java-Felder in KESOs Datenstrukturen eingetragen, sondern beim Generieren des C-Codes als Text angehängt. Folglich sind sie GradMP nicht bekannt. So wird die entsprechende Nullprüfung zwar bearbeitet, kann jedoch nicht abgeschaltet werden, da das betroffene Feld unbekannt ist, daher kein Offset hergeleitet werden kann und dadurch der betroffene Speicherbereich nicht genau bestimmt werden kann.

Bereichsprüfungen Die statische Analyse schafft es allein 12,8% der Prüfungen zu verwerfen. Ohne statische Analyse können keine Bereichsprüfungen abgeschaltet werden, ohne die Java-Spezifikation zu verletzen. Nimmt man eine Verletzung der Java-Spezifikation in Kauf, können 34,4% der Bereichsprüfungen abgeschaltet werden. Dies betrifft jene Vektorzugriffe bei denen primitive Daten gelesen werden. Nimmt man nun die statische Analyse wieder hinzu, können 47,2% der Bereichsprüfungen verworfen werden. Da die statische Analyse hier nur Prüfungen vor Schreibzugriffen und GradMP nur jene vor Lesezugriffen verwirft, ergänzen sich die beiden Optimierungen zufällig genau.

4.4 Laufzeiteinsparungen

Zur Messung der Laufzeiteinsparung werden für verschiedene Optimierungsoptionen jeweils 1000 Durchläufe des CDx gemessen und dann untereinander verglichen. Hierbei treten teilweise unerwartete Ergebnisse auf, auf die am Ende dieses Abschnitts näher eingegangen wird.

Zuerst wird der Fall betrachtet, dass die statische Analyse vollständig aktiv ist - wie es bei der normalen Verwendung KESOs üblich ist. Die Optimierungsstufe des C-

GradMP	0	1	2	2	3	3N	3B
stat. Analyse	x	x	x	x	-	x	x
no_valid_this_ptr	-	-	-	x	-	-	-
O3	0,00	0,11	1,40	3,38	4,44	-0,05	2,12
O1	0,00	2,86	3,11	3,32	6,65	1,24	3,89
Os	0,00	4,35	1,98	1,76	4,23	0,15	2,70

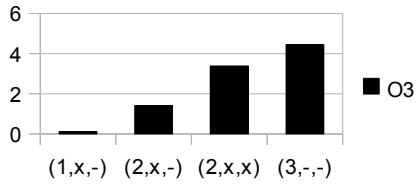

Abbildung 4.3: Verminderung der Laufzeit in %

Übersetzers ist „-O3“. Entfernt man alle verbleibenden Laufzeitprüfungen, vermindert sich die Laufzeit durchschnittlich um 4,4% (Varianz: 0,13%). Entfernt man nur die verbleibenden Bereichsprüfungen sinkt sie um 2,1% (Varianz: 0,14%). Die Laufzeiteinsparung, die das Verwerfen der verbleibenden Nullprüfungen mit sich bringt, ist kaum messbar. Tatsächlich verschlechtert sich die Laufzeit in den Tests durchschnittlich um 0,05%. Die Varianz beträgt 0,03%. Erlaubt man GradMP nun die Entfernung aller Prüfungen, die durch die Speicherschutzeinheit übernommen werden können, verbessert sich die Laufzeit um 0,11% (Varianz: 0,03%). Dies ist ein unerwartetes Ergebnis, schließlich kann die MPU nur Nullprüfungen übernehmen und das Entfernen aller Nullprüfungen führte zu keinem messbaren Ergebnis. Möglicherweise handelt es sich bei solch kleinen Werten um Messungenauigkeiten. Erlaubt man nun die Verletzung der Java-Spezifikation, was das Verwerfen einiger Bereichsprüfungen erlaubt, verbessert sich die Laufzeit insgesamt um 1,4% (Varianz: 0,18%). Werden nun noch die Nullprüfungen entfernt, die sich vor direkten Methodenaufrufen befinden, ist eine Verbesserung von 3,4% (Varianz: 0,07%) zu messen. Auch dieses Ergebnis ist überraschend, da wieder nur Nullprüfungen entfernt wurden. Eventuell lässt eine Verschränkung von Null- und Bereichsprüfungen nur dann eine effektive Optimierung zu, wenn beide Prüfungen entfernt wurden. Bei der stichprobenartigen Inspektion des Maschinencodes wurden jedoch keine Beweise dafür gefunden. Gegenüber dem Ausgangspunkt kann damit bis zu 77,3% der durch Laufzeitprüfungen verursachten Laufzeit eingespart werden. Dazu ist zu sagen, dass der Anteil der Laufzeitprüfungen an der Laufzeit mit 4,4% bereits sehr gering ist. Abbildung 4.3 fasst diese Ergebnisse zusammen. Da die Leistungssteigerung auch von der Optimierungsstufe des C-Übersetzers abhängt, wird zusätzlich „-O1“ und „-Os“ betrachtet. Vor allem bei „-Os“ lassen sich wieder unerwartete Ergebnisse beobachten. So sinkt die Leistung, je stärker optimiert wird.

Bei den obigen Messungen wurden die meisten Prüfungen bereits durch die statische Analyse entfernt. Aus Interesse an der Leistung GradMPs an sich, werden die Messungen wiederholt. Nun wird es der statischen Analyse verboten Laufzeitprüfungen zu entfernen. Eine Entfernung aller Laufzeitprüfungen verringert die Laufzeit jetzt um 19%. Entfernt man alle Bereichsprüfungen, sinkt sie um 2%. Nach dem Entfernen aller Nullprüfungen sinkt sie um 16,4%. Verwirft man nun wieder die Prüfungen, die durch die MPU übernommen werden können, sinkt die Laufzeit um 15,8%. Das Abschalten aller

GradMP	0	0	1	2	2	2	3N	3B	3
stat. Analyse	-	x	-	-	-	x	-	-	-
no_valid_this_ptr	-	-	-	-	x	x	-	-	-
O3	0,00	15,25	15,82	14,45	17,56	18,12	16,36	2,03	19,02
O1	0,00	10,23	13,24	15,04	13,21	13,20	14,93	3,16	16,20
Os	0,00	12,83	13,53	14,05	14,65	14,36	14,25	1,04	16,52

Abbildung 4.4: Verminderung der Laufzeit in % (ohne stat. Analyse)

Bereichsprüfungen, die nur vor lokalen Auswirkungen schützen, lässt die Laufzeiteinsparungen wieder auf 14,5% sinken. Für dieses Ergebnis konnte keine Erklärung gefunden werden. Erlaubt man nun zusätzlich die Entfernung von Nullprüfungen vor direkten Methodenaufrufen, sinkt die Laufzeit insgesamt um 17,6%. Aktiviert man nun wieder die statische Analyse, misst man eine Verbesserung von 18,1%. Die statische Analyse allein schafft 15,3%. Damit kann bis zu 95,3% der durch Laufzeitprüfungen verursachten Laufzeit eingespart werden. Abbildung 4.4 fasst diese Ergebnisse zusammen.

Beim Messen der Laufzeiteinsparung traten teilweise unerwartete Ergebnisse auf, für die im Rahmen dieser Arbeit keine sichere Erklärung gefunden wurde. Wärend im generierten C-Code keine Ursachen für die gemessenen Unregelmäßigkeiten gefunden wurde, könnten sie ihre Ursache in den Entscheidungen haben, die der C-Übersetzer aufgrund des veränderten C-Codes trifft. Doch auch bei der stichprobenartigen Inspektion des generierten Maschinencodes konnte die Ursache der Unregelmäßigkeiten nicht festgestellt werden.

4.5 Programmgröße

Um die Änderungen in der Programmgröße zu vergleichen, werden nur die Größen der Symbole betrachtet, die aus Java-Klassen entstanden sind. Der C-Compiler benutzt die Optimierungsstufe „-O3“. Für jede Messung wurde das Programm neu kompiliert.

Zuerst wird der Fall betrachtet, dass die statische Analyse vollständig aktiv ist - wie es bei der normalen Verwendung KESOs üblich ist. Hier machen die Laufzeitprüfungen noch 16,2% der Programmgröße aus. Schaltet man alle Bereichsprüfungen ab, spart man 5,4%, bei allen Nullprüfungen 11,5% der Größe. Ohne Verletzung der Java-Spezifikation kann GradMP die Größe um 6,3% vermindern. Verletzt man den Standard und entfernt Bereichsprüfungen, die nur vor lokalen Auswirkungen schützen, vermindert sich die Größe um 8,2%. Maximal können 13,3% der Größe eingespart werden. Hierzu muss zusätzlich die Option „no_valid_this_ptr“ aktiviert werden, die es erlaubt annähernd alle Nullprüfungen zu entfernen. Damit wurden 82,1% der, durch Laufzeitprüfungen verursachten, Programmgröße eingespart.

Betrachtet man nun den Fall, dass keine Prüfungen von der statischen Analyse entfernt werden, machen Laufzeitprüfungen 41% der Programmgröße aus. Entfernt man alle

GradMP	0	0	1	1	2	2	2	2	3N	3B	3	3N	3B
stat. Analyse	-	x	-	x	-	-	x	x	-	-	-	x	x
no_valid_this_ptr	-	-	-	-	-	x	-	x	-	-	-	-	-
O3	0,00	29,70	28,57	34,14	29,83	38,74	35,43	39,05	37,44	6,18	41,08	37,76	33,52
Os	0,00	19,70	17,95	23,12	19,29	28,75	24,46	29,13	27,45	3,85	31,60	27,84	23,30

Abbildung 4.5: Verminderung der Codegröße in %

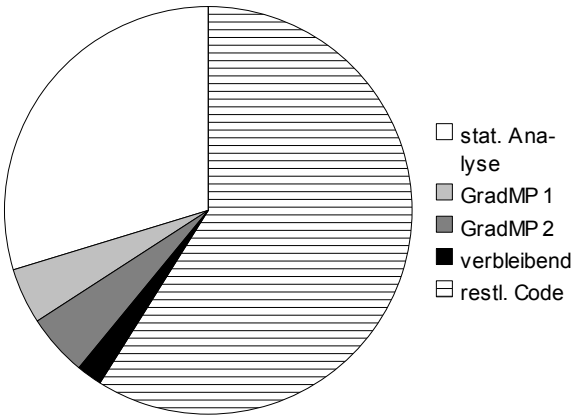

Abbildung 4.6: Verminderung der Codegröße (-O3)

Bereichsprüfungen, sinkt die Größe um 6,2%. Entfernt man alle Nullprüfungen sinkt sie um 37,4%. Dass diese beiden Zahlen addiert mehr als 41% ergeben, liegt daran, dass es sich bei jeder Messung um ein eigenes Kompilat handelt. Ohne Verletzung der Java-Spezifikation kann GradMP die Größe um 28,6% vermindern. Unter Verletzung des Java-Standards kann die Größe um bis zu 38,7% vermindert werden. Wobei sich der Zuwachs hauptsächlich durch das Entfernen von Nullprüfungen vor direkten Methodenaufrufen erklärt. Die statische Analyse allein vermag es die Programmgröße um 29,7% zu senken. Verwendet man beide Optimierungen sinkt die Größe um 39,1%. Damit wurden insgesamt 95,4% der durch Laufzeitprüfungen verursachten Programmgröße eingespart.

Die Einsparungen verhalten sich damit so, wie man es anhand des Anteils der verworfenen Prüfungen erwartet. Die Messungen werden in Tabelle 4.5 zusammengefasst. Da diese Zahlen auch von der Optimierungsstufe des C-Compilers abhängen, wird neben der Stufe „-O3“ auch „-Os“ gezeigt. Die Optimierungsstufen „-O1“ und „-O2“ verhalten sich hier so ähnlich wie „-Os“. „-O3“ und „-Os“ entscheiden sich hauptsächlich dadurch, dass die Laufzeitprüfungen bei „-Os“ einen kleineren Anteil der Programmgröße ausmachen. Abbildung 4.6 veranschaulicht die obigen Ergebnisse. „restl. Code“ bedeutet die Codegröße, die nicht durch Laufzeitprüfungen verursacht wird.

5 Zusammenfassung und Schlussfolgerungen

Zusammenfassung Ziel dieser Arbeit ist es die zusätzlichen Kosten von Laufzeitprüfungen zu vermindern, ohne dabei die Speichersicherheit zu gefährden. Was Laufzeitkosten angeht, leistet die statische Analyse bereits gute Arbeit. So machen die Prüfungen nur noch 4,4% der Laufzeit aus. Betrachtet man die Programmgröße, verbleiben immerhin 16,2% zur Optimierung. Die verbleibenden Kosten können durch GradMP größtenteils eingespart werden. Die Laufzeit kann um 3,4% gesenkt werden, die Programmgröße um 13,3%. Hierzu ist es leider nötig Verletzungen der Java-Spezifikation in Kauf zu nehmen. Die Nullprüfungen, die ohne Verletzung der Spezifikation entfernt werden können, fallen kaum ins Gewicht, da bereits 75% durch die statische Analyse verworfen wurden.

Anwendbar ist GradMP auf allen hier untersuchten Architekturen. Während dies auf dem Tricore (und i386-Linux) ohne Aufwand umsetzbar war, muss der AT91 durch Konfiguration der MPU dazu gebracht werden GradMP im vollen Umfang zu unterstützen.

GradMP als eigenständige Optimierung Der graduelle softwarebasierte Speicherschutz konnte nicht vollständig getrennt von der statischen Analyse gemessen werden. Bei der Entfernung von Laufzeitprüfungen zeigte sich er sich aber ähnlich erfolgreich wie die gegenwärtige statische Analyse.

Der graduelle softwarebasierte Speicherschutz ist vergleichsweise einfach zu implementieren. Er benötigt keine „Static-Single-Assignment“-Darstellung oder Datenflussanalyse. Es wird lediglich auf dem einzelnen Befehl gearbeitet. Auch die Entscheidung, welche Nullprüfungen für ein bestimmtes Feld durchgeführt werden muss, ist einfach durchzuführen. Oft muss nur geprüft werden, ob das Feld in einem bestimmten Bereich um *null* liegt.

Auf Grund der Einfachheit könnte GradMP für dynamische Übersetzer („Just-in-time-compiler“) geeignet sein, die Programme stufenweise optimieren. Diese benutzen anfänglich nur einfache Optimierungen und verstärken diese, falls der betroffene Code oft ausgeführt wird. Beim statischen Übersetzer KESO ist dies kein Vorteil.

Entwurf von Adressräumen Beim Entwurf eines Mikrocontrollers, oder allgemein eines Adressraumes, sollte um die Adresse 0 Platz für einen Trap-Mechanismus eingeplant werden. Dies ist nicht nur für die Umsetzung typsicherer Sprachen vorteilhaft, sondern

auch für Programme die in der Sprache C geschrieben wurden, da auch C ungültige Adressen mit „NULL“, also der Adresse 0 kennzeichnet.

Übertragbarkeit auf andere typsichere Sprachen Der Grund für die Verwendung Javas ist seine Typsicherheit. Dass andere typsichere Sprachen für Mikrocontroller besser geeignet sein könnten, deutet sich dadurch an, dass KESO Java-Programme - auch ohne GradMP - nicht vollständig standardkonform ausführt (siehe Abschnitt 1.3). Die hier analysierten Sachverhalte treffen genauso auf andere typsichere Sprachen zu. Zum Beispiel stellt verifizierbarer CIL-Code [4], der dem JVM-Bytecode sehr ähnlich ist, die Typsicherheit ebenfalls durch Bounds- und Nullchecks sicher. Auch sogenannte „Fatpointer“ und „Never-Null-Pointer“, wie sie beispielsweise im C-Dialekt Cyclone [8] verwendet werden, benutzen ähnliche Prüfungen.

Literaturverzeichnis

[1] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, Dec. 1989.

[2] ARM Limited, 110 Fulbourn Road Cambridge, England CB1 9NJ. *ARMv7-M Architecture Reference Manual*, Feb. 2010.

[3] Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131, USA. *AT91SAM ARM-based Flash MCU, SAM3U Series (6430E-ATARM-29-Aug-11)*, Aug. 2011.

[4] ECMA, Rue du Rhone, CH-1204 Geneva. *Common Language Infrastructure (CLI) Partitions I to VI (ECMA-335)*, Dec. 2010.

[5] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 27–40, Apr. 2001.

[6] Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *TC1796 User's Manual (V2.0)*, July 2007.

[7] G. S. G. B. James Gosling, Bill Joy. In *The Java Language Specification, 3. Auflage*, 2005.

[8] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. pages 275–288, 2002.

[9] M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. Gradual Software-Based Memory Protection. In ACM, editor, *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS '10)*, New York, 2010.

[10] F. P. A. P. B. T. J. V. Tomas Kalibera, Jeff Hagelberg. CDx: A Family of Real-time Java Benchmarks. 2009.

[11] C. Wawersich, M. Stilkerich, and W. Schröder-Preikschat. An OSEK/VDX-based multi-JVM for automotive appliances. In *Embedded System Design: Topics, Techniques and Trends, IFIP International Federation for Information Processing*, pages 85–96, Boston, 2007.

[12] Wikipedia. *Data structure alignment*. `http://en.wikipedia.org/wiki/Data_structure_alignment` (Stand: 30.09.2011).