Ruhr-Universität Bochum

Tools for Embedded Software Reverse Engineering

Kristina Petrow (108015243832)
Leon Mrosewski (108015226296)
Darius Happe (108014233691)
Jan Pieter Van Wickern (108015232675)
Dennis Ogiermann (108013206934)
Nijat Aghayev (108015250476)

Projektarbeit 14 Wintersemester 17/18
Chair for Embedded Security
Horst Görtz Institute for IT-Security,
Ruhr-Universität Bochum

Projektleiter: Mark Fyrbiak, Benjamin Kollenda, Phillip Koppe

hgi
Horst Görtz Institut für IT-Sicherheit

Inhaltsverzeichnis

Einleitung

Das Projekt „Reverse Engineering for Embedded Systems“ behandelt die Methoden, die den Quelltext schwer verständlich oder schwer rückgewinnbar zu machen. Das Projekt ist sehr hilfreich, nicht nur um die Obfuskation sowie Obfuskationsmethoden kennenzulernen, sondern auch den Compilervorgang unter die Lupe zu nehmen. So werden im Projekt die Aufgabenbereiche der drei Phasen des Compilervogangs konkretisiert: Frontend, Transformation und Backend. Die Obfuskation selbst wird in der Transformationsphase durchgeführt.

Was ist überhaupt Obfuskation und wofür wird sie genau verwendet? Grob ausgedrückt, ist Obfuskation dafür da, um denjenigen, der versucht den Code zu analysieren oder zu verändern, die Kopfschmerzen zu bereiten. Drei generelle Hauptziele der Obfuskation sind:

1. Verschleierung des Codes
2. Reduzierung der Größe der Datei, die den Quellcode enthält
3. Verstecken der Unheberrechte

Für die Embedded Systems ist der 1. Punkt unersetzbar. Je schwerer der Quellcode zu nachvollziehen ist, desto höher ist die Sicherheit des Systems. Wenn der Angreifer den Quellcode nachvollziehen kann, kann er entweder Veränderungen durchführen oder Programmteile für seine persönliche Zwecke nutzen. Übersicht der wichtigsten Methoden der Obfuskation:

- Verändern des Kontrollflusses
- Verändern der Namen der Variablen oder Funktionen
- Einfügen von Code
- Verschlüsselung
- Spaltung von Variablen oder Funktionen
- Anti – Debugs

Ein Beispiel zur Obfuskation wäre z.B. aus unserem Projekt, das als Grundlage zur weiteren Zielverfolgung des Lehrstuhls dienen soll. Damit die Angreifer die Reihenfolge der ausführbaren Instruktionen schwer nachvollziehen können, soll die Reihenfolge der Basic Blöcke durcheinandergebracht und die Verweise(Jumps), die am Ende jedes Basic Blocks stehen, in eine Jumptabelle eingetragen werden, da die Angreifer keinen Zugriff auf die Jumptabelle haben. Um die Nachverfolgung des Quellcodes noch mehr zu erschweren, werden die Basic Blöcke auf die gleiche Größe gebracht und die Reihenfolge der Basic Blöcke verändert.

Um den Code zu verschleiern, gibt es je nach Programmiersprache und Plattform verschiedene Vorgehensweisen. Frameworks wie LLVM bieten für solche Zwecke die fertigen Werkzeuge, um für den Entwickler die Verschleierung so bequem wie möglich zu machen. Das war auch der Grund, warum wir uns für LLVM entschieden haben. Die Programmiersprache, die wir verwendet haben, ist C++, da es als Voraussetzung für LLVM-Nutzung ist. Obwohl die Obfuskation eine effektive Methode zur Erhöhung der Sicherheit des Systems bietet, kann sie das Entwickeln des Codes zeitaufwändiger machen und die Anwendungsmöglichkeiten der Reflexion auf verschleierten Code beschränken.

Design und Implementation

Target Architecture: MSP430

Der MSP430 ist ein 16-Bit-RISC-Mircocontroller, hergestellt von Texas Instruments. Programmierbar ist er in C oder in Assembler. Eingesetzt wird der Microcontroller hauptsächlich als Embedded System, z.B. als Wärmezähler, Heizkostenverteilung oder auch im medizinischen Bereich.

Der Prozessor hat 16 Register, dabei sind einige schon Vorbelegt:

1. R0 ist der *Programm Counter*
2. R1 ist der *Stack Pointer*
3. R2 ist das *Status Register*
4. R3 ist der *Constant Generator*

Die restlichen Register, also R4 bis R15, sind frei nutzbar und belegbar.

Der Microcontroller unterstützt 27 Standard Instruktion. Zusätzlich gibt es 24 emulierte Befehle. Diese können durch die Standard Instruktionen dargestellt werden und sind nur für eine bessere Code Übersicht sorgen bzw. das Programmieren leichter machen. Die Nutzung von emulierten Befehlen verschlechtert die Performance des Programms nicht.

Eine Instruktion beginnt immer mit einem Opcode. Dabei kann man die Opcodes bzw. die Befehle in drei Gruppen aufgeteilt werden:

1. Dual-Operand
2. Single-Operand
3. Jump

Zu unterscheiden ist noch zwischen *Word* und *Byte* Befehlen. *Word* steht für einen 16 Bit Befehl. *Byte* steht für einen 8 Bit Befehl. Um zwischen den Größen zu unterscheiden wird entweder ein *.w,* für *Word,* oder ein *.b,* für Byte, an den Opcode angehängt. Falls nichts angehängt wird handelt es sich um einen *Word* Befehl.

Bei den Dual-Operanden folgt auf den Opcode ein Source-Register und ein Destination-Register. Letztendlich wird das Ergebnis der Operation in das Destination-Register geschrieben. Der Inhalt des Source-Registers wird nicht verändert. Das ganze hat dann z.B. die Form:

add.w r4, r5

Dabei ist *r4* Das Source-Register und *r5* das Destination-Register.

Bei den Single-Operanden folgt nur ein Register auf den Opcode nur ein Register das entweder ein Destination-Register ist, also dessen Inhalt manipuliert wird, oder ein Source-Register ist. Wenn es sich um ein Source-Register handelt wird der Inhalt dieses Registers nicht verändert. Allerdings benutzt nur der „PUSH“ Befehl ein Source-Register.

Bei den Jump Befehlen geht es Hauptsächlich darum zu einem bestimmten Label zu springen. Dabei gibt es bei den verschiedenen Jump Befehlen unterschiedliche Voraussetzungen wann ein Sprung ausgeführt wird. Meistens hängt es mit den Status-Bits zusammen. Es gibt jedoch auch einen unbestimmten Sprung der immer ausgeführt wird.

Des weiteren unterstützt die Architektur 7 verschiedene Adressing Modes:

Adressing Mode	**Syntax**	**Beispiel**
Register Mode	Rn	add r5, r6
Indexed Mode	X(Rn)	add 5(r5), 6(r6)
Symbolic Mode	ADDR	add EDE, TONI
Absolute Mode	&ADDR	add &EDE, &TONI
Indirect Register Mode	@Rn	add @r5, @r6
Indirect Autoincrement Register Mode	@Rn+	add @r5+, @r6+
Immediate Mode	#N	add #5, r6

Dabei wird nur bei dem Register Mode direkt auf ein Register zugegriffen. Bei dem Immediate Mode handelt es sich nur um eine konkrete Zahl, und somit kann der Adressing Mode nur für Source-Register genutzt werden. Alle anderen Adressing Modes arbeiten nur auf Speicheradressen die entweder berechnet werden oder direkt geladen werden basierend auf dem Inhalt des Registers oder des Labels (EDE, TONI).

LLVM (externe Libraries)

LLVM ist eine modulare Compiler-Unterbau-Architektur. Sie hat einen virtuellen Befehlssatz und eine virtuelle Maschine die einen Prozessor virtualisiert.

Im allgemeinen können mit LLVM eigene Frontends für verschiedene Sprachen entwickelt werden die dann die LLVM Zwischensprache nutzen.

Wir haben also ein Frontend für den MSP430 geschrieben das den Assembler Code einliest und in die LLVM Zwischensprache, LLVM IR (Intermediate Representation), übersetzt, indem wir für eine MSP430-Instruktion eine oder mehrere LLVM IR Instruktionen erstellen, die den selben Zweck erfüllen. Außerdem haben wir mit Hilfe von LLVM Metadaten zu den MSP430-Instruktionen für die Transformations- und Backend-Phase erstellt.

Im Anschluss wurden im Transformation Teil auf eben diesem LLVM IR Code entsprechende Analysen und Transformationen ausgeführt, sodass der Programmcode einer Obfuscation unterzogen war. Hier wurden einerseits die Metadaten des Frontends, sowie der im Frontend erstellte LLVM IR Code, mit Hilfe von LLVM ausgelesen und andererseits neue LLVM IR Instruktionen erstellt (um zum Beispiel einen Basicblock aufzufüllen).

Desweiteren haben wir LLVM auch für die Arbeit mit dem LLVM IR Code verwendet, um den Code durchzugehen und die verschiedenen Transformationen durchzuführen (z.B. das Splitten der Basicblöcke oder Einfügen von neuen LLVM IR Instruktionen).

Das Backend war letztendlich dafür verantwortlich, dass der veränderte LLVM IR Code wieder in die Ausgangssprache übersetzt wurde. Dabei wurde ebenfalls das Auslesen der Metadaten und des LLVM IR Codes der Transformation durch LLVM realisiert.

Frontend

Leon Mrosewski, Jan Van Wickern, Darius Happe

Einleitung

Die Hauptaufgabe des Frontends bestand daraus einen gegebenen MSP430 Assembler Code einzulesen und diesen anschließend auf die LLVM-Zwischensprache LLVM IR (LLVM Intermediate Representation) zu “liften“.

Im Allgemeinen heißt das, dass wir jede Instruktion gegen ihren Pendant in der LLVM IR Sprache austauschen.Allerdings gibt es nicht für jede Instruktion für den MSP430 eine passende Gegeninstruktion in der LLVM IR. Somit müssen manche Instruktionen mit mehreren LLVM IR Instruktionen dargestellt werden. Dies ist vor allem bei den verschiedenen Adressing Modes der Fall.

Im folgenden wird der allgemeine Programmverlauf kurz Skizziert. Im Verlauf dieser Dokumentation wird noch genauer auf die einzelnen Funktionen eingegangen.Zunächst wird, nachdem das Inputfile geöffnet wurde, nach bestimmten Header Anweisungen gesucht. Diese sind zwar nicht unbedingt wichtig für llvm aber da wir später wieder zur Ausgangsarchitektur zurück kehren wollen (MSP430) werden diese benötigt.Im zweiten Schritt wird nach Labeln gesucht. Dafür wird einmal das gesamte File durchlaufen. Dies ist wichtig damit später ordentliche Verkettungen (wie Jumps) erstellt werden können.

Daran anschließend werden nun die Instruktionen ausgelesen. Dabei wird der Vorgang unterteilt in das Auslesen des Opcodes (also der Anweisung) und das Auslesen der betroffenen Register und deren Adressing Modes.Hierbei ist zu beachten, dass wir nur eine Instruktion auf einmal einlesen und liften. Ist der lift abgeschlossen wird zur nächsten Instruktion gesprungen.Wenn eine Instruktion geliftet werden soll werden zunächst, abhängig vom Adressing Mode, eventuell erforderliche zusätzliche Instruktionen erstellt.

Im Anschluss wird die vorbereitete Instruktion durch eine andere Funktion letztendlich geliftet. Das heißt das die passende llvm Instruktion erstellt wird und in den aktuellen Basicblock eingetragen wird.

Anschließend werden noch die Metadaten angehängt, die von der Transformation und dem Backend benötigt werden.Damit ist der lift einer Instruktion abgeschlossen und das Programm springt zur nächsten.

Blick in die Funktionen: parse

Verantwortliche: Leon Mrosewski, Jan Van Wickern
Optimierungen: Darius Happe

Die Hauptaufgabe der „*parse(..)*“ Funktion ist das einlesen der Instruktionen und das weitergeben der relevanten Informationen an die Funktionen die das liften umsetzen.

Zunächst müssen die Header des gegebenen Assembler Files eingelesen werden.

Dies geschieht über die „*getHeadInformations(...)*“ Funktion. Derzeit werden nur „*.global*“, welches den Namen des Assembler Files definiert und „*.text*“, welches den Start des Programms angibt unterstützt.
Sobald das erste Label eingelesen wird wird die Schleife beendet, da nun die Instruktionen folgen und keine weiteren Header Informationen folgen können.

Im nächsten Schritt müssen zunächst alle Label die in dem Input File vorhanden sind eingelesen werden. Anschließend wird für jedes Label ein Basicblock erstellt und in einer Map hinterlegt.
Dies ist wichtig damit später, wenn zum Beispiel Jump Instruktionen umgesetzt werden müssen, der folge Basicblock direkt eingesetzt werden kann.
Dabei wird das Komplette Input File einmal komplett durchlaufen. Ist dies geschehen und alle vorhanden Labels wurden eingelesen wird der Inputstream, über welchen die Datei eingelesen wurde, wieder zurück gesetzt.

Nun folgt das einlesen der Instruktionen. Dafür wird zunächst über „*gettok(...)*“ zunächst der Opcode der Instruktion einlesen. Opcodes können zum Beispiel „*add*“, „*mov*“, „*sub*“ etc. sein.

Es können über diese Funktion allerdings auch Labels eingelesen werden. Wenn dies der Fall ist wird aus der zuvor erstellten Basicblock Map der passende Basicblock geladen und in die llvm Mainfunktion eingefügt. Somit wird jede weitere Instruktion die wir erstellen in diesen Basicblock eingefügt.

Wenn jetzt allerdings ein anderer Opcode eingelesen wird müssen, abhängig vom Opcode, die weiteren zugehörigen Variabeln (Register, Immediates,…) eingelesen werden.
Dies wird über die Funktion „*gettok2op(...)*“ realisiert.

Hierbei wird in der *parse* Funktion zwischen verschiedenen Fällen unterschieden.
Wenn es sich um eine Instruktion mit zwei Parametern handelt (also mit einem Source und einem Destination Register) werden diese hintereinander eingelesen und in Variablen gespeichert.
Dabei wird auch der entsprechende Adressing Mode mit ausgelesen.
Ist dies ohne Fehler geschehen werden all diese ermittelten Daten an die „*prepareForLift(...)*“ Funktion übergeben.

Wenn es sich bei dem Opcode um einen der emulierten Befehle handelt muss vorher noch überprüft werden ob der emulierte Befehl einen oder zwei Variablen nutzt und dementsprechend auslesen und die *prepareForLift(…)* Funktion aufrufen.

Wurden nun alle Instruktionen eingelesen wird der Inputstream geschlossen.
Anschließend wird über eine llvm Funktion der erstellte LLVM IR Code auf Korrektheit überprüft. Falls dieser in seiner Form nicht lauffähig ist wird der Nutzer darauf hingewiesen das der Code *„malformed“* ist.

Falls er jedoch alle Tests besteht wird er in die Ausgabedatei, die der Nutzer bei Programmstart festgelegt hat geschrieben. Das Ausgabeformat ist dabei Bitcode.

Blick in die Funktionen: getHeadInformations

Verantwortliche: Leon Mrosewski, Darius Happe

Die „*getHeadInformations(...)*“ Funktion liest im allgemeinen nur die Header einer MSP430 Assembler File ein. Mögliche Header sind in dem Fall „*.global*“ oder „*.text*“.
Die Funktion gibt, falls sie ein passenden bzw. existierenden Header gefunden hat, ein entsprechendes Token für diesen Header zurück. Dieses Token wird dann in der *parse(…)* Funktion verarbeitet. Zusätzlich werden, wenn dem Header noch weitere Informationen folgen (wie im Fall von *.global*), diese in einem übergeben String Pointer gespeichert, so dass *parse* darauf zugriff hat.

Desweiteren werden die zuvor ermittelten Header noch als Metadaten angehängt, damit diese im Backend wieder ausgelesen werden können und in die Ausgabe MSP430 Assembler Datei eingefügt werden können. Dies wird über die *appendGlobalMD(…)* Funktion realisiert.

Blick in die Funktionen: gettok

Verantwortliche: Leon Mrosewski

Die *gettok(…)* Funktion liest den Opcode der aktuellen Instruktion ein. Dabei gibt es zu beginn beim einlesen keine Restriktionen oder Verschachtelungen. Es wird erst mal nur eingelesen.
Wichtig wird es sobald der Opcode an sich abgeschlossen ist. Nun wird nämlich festgelegt ob es sich um einen Word (16 bit) oder einen Byte (8 bit) Befehl handelt.

Ein Word Befehl liegt vor wenn entweder “*.w*“ direkt an den Opcode angehängt ist oder ob einfach gar nichts angehängt wurde.

Beispiel Word Befehl:

1. add.w r5, r6
2. add r5, r6

Um einen Byte Befehl zu erhalten gibt es nur die Möglichkeit ein “*.b*“ an den Opcode anzuhängen.

Das heißt also wenn ein Punkt direkt nach dem Opcode folgt muss entweder ein *w* oder ein *b* folgen. Falls jedoch nur ein Leerzeichen folgt wird die Instruktionsgröße direkt auf Word gesetzt.

Danach wird geprüft zu welcher Instruktionskategorie der Opcode gehört. Der ermittelte Opcode wird nun mithilfe des übergebenen Pointers auf einen String gespeichert und es wird ein passendes Token zurückgegeben, sodass die *parse* Funktion damit weiter verfahren kann.

Desweiteren können auch Label in der *gettok* Funktion erkannt werden. Diese sind erkennbar daran das direkt nach dem ermittelten Opcode (welcher in diesem Fall kein Opcode ist) ein “*:*“ folgt.

Blick in die Funktionen: gettok2op

Verantwortliche: Leon Mrosewski, Jan Van Wickern

Die *gettok2op(…)* Funktion ermittelt die dem Opcode nachfolgenden Variablen die für für die Instruktion genutzt werden.

Dabei wird zwischen 7 Verschiedenen Adressingmodes unterschieden:

Adressing Mode	**Syntax**	**Beispiel**
Register Mode	Rn	add r5, r6
Indexed Mode	X(Rn)	add 5(r5), 6(r6)
Symbolic Mode	ADDR	add EDE, TONI
Absolute Mode	&ADDR	add &EDE, &TONI
Indirect Register Mode	@Rn	add @r5, @r6
Indirect Autoincrement Register Mode	@Rn+	add @r5+, @r6+
Immediate Mode	#N	add #5, r5

Dabei kann jeder Adressing Mode mit jedem anderem Adressing Mode kombiniert werden. Eine Ausnahme bildet da allerdings der Immediate Mode, da dieser nicht auf der Destination Seite stehen kann. Dies hat den Grund das man kein Ergebnis in eine Zahl schreiben kann.

Jetzt liest die *gettok2op* eben die Register mit ihrem Adressing Mode ein. Dabei liest sie jedoch bei jedem Aufruf der Funktion nur ein Register ein. Das heißt um zwei Register für einen Opcode einzulesen muss diese zweimal aufgerufen werden.
In ihrem Verlauf unterscheidet die Funktion zwischen den verschiedenen Darstellungsarten für die verschiedenen Adressing Modes und kann so genau bestimmen um welchen es sich handelt.
Dementsprechend wird wieder ein Token zurückgegeben, sodass die *parse* Funktion dies entsprechend verarbeiten/ weitergeben kann.

Blick in die Funktionen: prepareForLift

Verantwortliche: Leon Mrosewski

Optimierungen: Darius Happe

Die Aufgabe der *prepareForLift(…)* Funktion ist, falls die aktuelle Instruktion nur durch mehrere LLVM IR Instruktionen dargestellt bzw. emuliert werden kann, diese Instruktionen zu erstellen.
Dabei muss zwischen den verschiedenen Adressing Modes unterschieden werden.
Je nach Adressing Mode müssen eventuelle zusätzliche Instruktionen hinzugefügt werden, damit der MSP430 Befehl korrekt emuliert bzw. in LLVM IR übersetzt werden kann.

Bei dem Indexed Mode zum Beispiel muss zunächst der Inhalt des betroffenen Registers ausgelesen werden. Anschließend muss der Offset, welcher mitgegeben wurde, zu dem Wert in dem Register addiert werden. Das Ergebnis aus dieser Addition ergibt die Speicheradresse von der aus geladen bzw. in die geschrieben werden soll. Der Inhalt der Register wird dabei nicht verändert.

Zum Beispiel wird aus der MSP430 Instruktion

ADD 2(R5), 4(R6)

folgende LLVM IR Instruktionsfolge:

%r5_TMP_1 = add i16 %r5, 2	→ addieren des Offsets auf Registerinhalt
%r5_TMP_2 = load i16, i16 %r5_TMP_1	→ Laden von der zuvor berechneten Adresse
%r6_TMP_1 = add i16 %r6, 4	→ addieren des Offsets auf Registerinhalt
%r6_TMP_2 = load i16, i16 %r6_TMP_1	→ Laden von der zuvor berechneten Adresse
%TMP_1 = add i16 %r5_TMP_2, %r6_TMP_2	→ Addieren der geladenen Adressinhalte
store i16 %TMP_1, i16 %r6_TMP_2	→ speichern des Ergebnisses

Also wird für jedes benutze Register zunächst die passende Funktion zum Vorbereiten der Instruktion aufgerufen.

Dabei muss nur für den Register Mode und den Immediate Mode keine extra Funktion durchlaufen werden. Alle anderen Adressing Modes benötigen zusätzliche Instruktionen um diese in LLVM IR darstellen zu können.
Ist dies geschehen wird anschließend die finale Funktion zum liften des eigentlichen Befehls aufgerufen.
Handelt es sich um das normale Instructionset wird *liftBinRegister(…)* aufgerufen. Falls es sich um einen emulierten Befehl handelt wird *liftEmulOp(…)* aufgerufen.

Zusätzlich wird in der *prepareForLift* Funktion noch ein String erstellt der die alte MPS430 Instruktion enthält. Dieser wird letztendlich für die Metadaten benötigt.

Blick in die Funktionen: liftBinRegister

Verantwortliche: Leon Mrosewski

Optimierungen: Darius Happe, Jan Van Wickern

In dieser Funktion wird letztendlich die LLVM IR Instruktion erstellt die die MSP430 Instruktion darstellt.
Allerdings nur für die Befehle die direkt vorhanden sind. Für die Emulierten Befehle gibt es eine weitere Funktion.

Zunächst wird anhand des übergeben Opcodes eine LLVM IR Instruktion mittels *BinaryOps* erstellt.

Dies hat die Form

Instruction::BinaryOps instruction = Instruction::Add

Erstellen einer Add Instruktion

Anschließend werden die benötigen Register (welche in LLVM IR schon existieren und in der *prepareForLift(…)* erstellt wurden) aus der Register Map geladen.

Nun wird, abhängig von Adressing Mode, die eigentliche Instruktion erstellt. Zu unterscheiden ist dabei zwischen dem Speichern des Ergebnisses in einem Register oder an einer berechneten Speicherstelle. Im ersten Fall muss einfach nur die Instruktion erstellt werden und der Vorgang ist abgeschlossen.

Falls jedoch an eine Speicheradresse geschrieben werden muss, wird erst die eigentliche Instruktion erstellt und im Anschluss wird das Ergebnis an die berechnete Speicheradresse mittels einer *Store* Instruktion geschrieben. In diesem Schritt wird die erstelle Instruktion auch letztendlich an den aktuellen Basicblock angehängt.

Nun wird noch die *appendMetadata(…)* Funktion aufgerufen um die benötigten Metadaten anzuhängen.

Ein Sonderfall stellt jedoch die MSP430 *Mov* Instruktion dar.
Es gibt keine LLVM IR Instruktion die diese Funktion genau nachbildet. Daher wird hier die Add Instruktion genutzt. Es wird also der Wert der in das Zielregister geschrieben werden soll mit 0 Addiert und dann im LLVM IR Register gespeichert.

Aus zum Beispiel

MOV #5, R5

wird also in LLVM IR

%r5 = add i16 0, 9

Blick in die Funktionen: liftEmulOp

Verantwortliche: Leon Mrosewski

Diese Funktion hat eigentlich die gleiche Funktionalität wie *liftBinRegister* behandelt jedoch nur emulierte Befehle.
Emulierte Befehle des MSP430 können durch eine Folge von “normalen“ Befehlen aus dem MSP430 Instructionsset dargestellt werden.

Im allgemeinen ist der Ablauf genau der selbe wie in *liftBinRegister*.

Es wird der Opcode überprüft und die passende Instruktion erstellt.
Anschließend wird passend zum Adressing Mode eine LLVM IR Instruktion erstellt die dann im Basicblock angehängt wird.

Blick in die Funktion: appendGlobalMD

Verantwortliche: Darius Happe

Diese Funktion ist dafür da die Header-Informationen einer MSP430 Assembly File an das aktuelle LLVM-Modul anzuhängen. Dazu bekommt appendGlobalMD() von der getHeadInformation()-Funktion, entweder „global“ oder „text“ übergeben, je nachdem was dort ausgelesen wurde.

Bei „global“ wird zusätzlich noch der globale Identifier übergeben.

Diese Strings werden zuerst in llvm:MDString konvertiert, dann in ein Array eingefügt(als einzigen Wert) , um dieses Array in ein llvm:MDTupel einfügen zu können.

```
NamedMDNode→addOperand(MDTuple::get(ctx,(ArrayRef<Metadata*>(MDString::get(...));
```

String als Metadaten an das Modul anhängen(NamedMDNode ist der Container für Metadaten)

An das llvm:Modul können Metadaten angehängt werden, indem man sie in eine llvm:NamedMDNode einfügt, die man aber auch benennen kann. Dort werden aber nur MDTupel akzeptiert.

Blick in die Funktion: registerToInt

Verantwortliche: Darius Happe

registerToInt ist eine sehr simple Funktion, die nur dazu da ist den RegisterString (z.B: R7)

in einen Integer zu überführen in der kein Buchstaben mehr auftauchen. Dieser Integer wird benötigt um alle lesenden und schreibenden Register in den Metadaten zu übergeben(siehe dazu appendMetadata()).

Der String den die Funktion bekommt wird nur von vorne bis hinten durchgegangen, Buchstaben werden eliminiert und es wird überprüft ob die Form des Regster stimmt. Das wird an einen neuen String gepusht und dieser dann am Ende mit stoi() zu einem Int konvertiert.

Transformation

Dennis Ogiermann, Kristina Petrow

Einleitung

Die Hauptaufgabe der Transformation besteht darin, die Transformationen bzw. Verschleierungen (Obfuscation Passes) des Codes, der von Frontend in die LLVM IR (Intermediate Representation) geliftet wurde, durchzuführen.

Die Aufgabe des Transformationsteams war mehrere Analysen und Transformationen (sogenannte Passes) zu implementieren, sowie 2 Test Cases zu schreiben.

Es folgt vorerst kurze Beschreibung des Programmablaufs, die genaueren Angaben zur einzelnen Funktionen sind unten zu finden.

Als Erstes wurden 2 Test Cases ausgesucht, die mehrere Funktionen und somit mehrere Basic Blöcke beinhalten, um möglichst große Testabdeckung zu gewährleisten.

Das erste Test Case behandelt Mean shift, ein nicht parametrisches Verfahren zum iterativen Finden von Maxima in Dichtefunktionen. Es wird ein Startfenster festgelegt, deren Verschiebung mithilfe des Gradienten zum Punkt der größten Verteilungsdichte erfolgt. Das zweite Test Case implementiert das Newton Verfahren, welches zur Annäherung einer Nullstelle einer Gleichung mithilfe der Tangente dient.

Die Transformation selbst beginnt mit dem Einlesen der Inputdatei, welche die Frontendphase bereits durchlaufen hat.

Zuerst müssen die Metadaten eingelesen und analysiert werden. Wenn die Analyse erfolgreich war, wird die Bytegröße jedes Basic Blocks berechnet.

Als nächstes werden die Blöcke so gespaltet, dass die Bytegröße jedes Basic Blocks nicht größer als der Mittelwert ist.

Trotz diesem Schritt gibt es Basic Blöcke, die kleiner sind als der Mittelwert und dementsprechend müssen diese Basic Blöcke an die Bytegröße des Mittelwertes angepasst werden. Das Aufpumpen von Basic Blöcken wird von Backendphase übernommen, jedoch ist die Transformationsphase für die Übermittlung dieser Informationen zuständig.

Abschließend wird die Liveness Analyse durchgeführt, um den Nutzungsstatus einzelner Register zu betrachten, damit das Backend weiß, welche Register es zum Auffüllen der Basic Blöcke nutzen darf. Über die Metadaten kann man sehen, welche Register in der aktuellen Instruktion geschrieben und welche gelesen werden.

Einblick in den Size Analysis Pass:

Verantwortliche: Dennis Ogiermann, Kristina Petrow

Die Aufgabe des „Size Analysis“ Passes ist es die übergebenen Metadaten zu analysieren und den Mittelwert der Bytegröße der Basic Blöcke zu berechnen.

Als Erstes durchläuft jeder Basic Block wird die Funktion „*analyzeBlockSize(…)*“, die für den aktuellen Basic Block die Bytegröße berechnet. Für jede Instruktion des aktuellen Basic Blocks wird die Funktion „*getInstructionSize(…)*“ aufgerufen. Diese Funktion liest die Metadaten, die von Frontend zur Verfügung gestellt werden, ein und summiert alle Instruktionsgrößen, die im aktuellen Basic Block vorkommen.

Sollten die Metadaten keine oder unvollständige Daten (z.B. Instruktionsgröße fehlt) enthalten, dann wird das Programm abgebrochen, da an diesem Punkt davon ausgegangen werden kann, dass die Daten beschädigt sind oder ein vorausgehender Fehler nicht abgefangen wurde.

Nachdem die beiden Funktionen durchlaufen sind, wird der Mittelwert ermittelt.

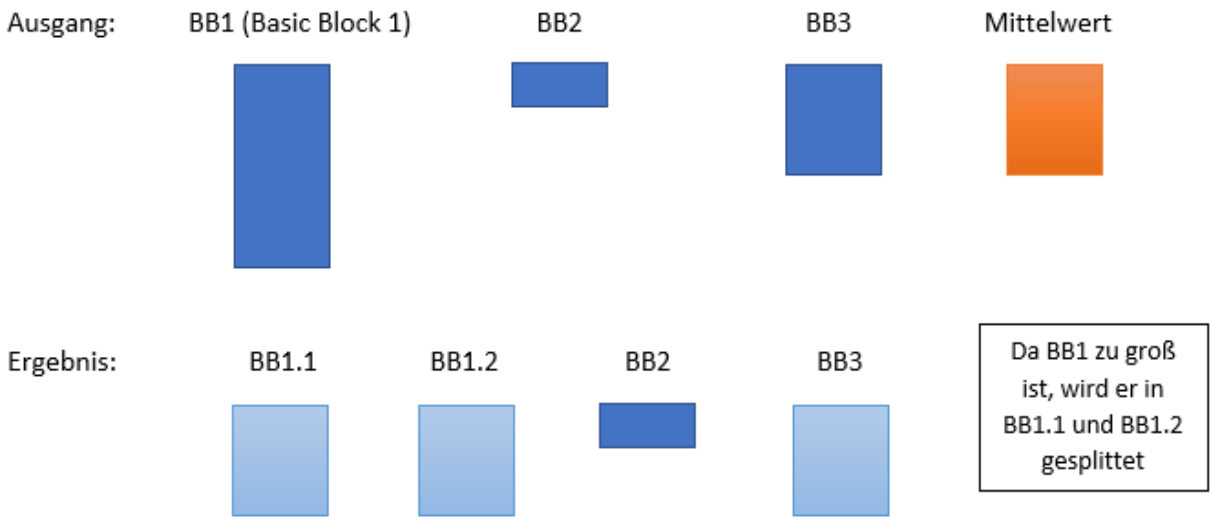
Einblick in den Splitting Pass:

Verantwortliche: Dennis Ogiermann, Kristina Petrow

Die Aufgabe dieses Passes ist es die Basic Blöcke, die größer als der Mittelwert sind, zu splitten, so dass nach dem Durchlauf dieser Funktion die Basic Blöcke entweder exakt die gleiche Größe wie der Mittelwert haben oder kleiner sind.

Für jeden Basic Block wird die Funktion *„continuousSplitting(…)“* gestartet. Zunächst läuft erneut eine Analyse („*analyzeBlockSize(…)*“ und folglich „*getInstructionSize(…)*“) durch. Gleichzeitig werden die Instruktionsgrößen (Info: eine Instruktion kann entweder 1 oder 2 Byte groß sein) aufaddiert, bis die Mittelwertgröße überschritten wird. Basic Blöcke werden mithilfe der Funktion *„splitBasicBlock(…)“*, die vom LLVM Framework zur Verfügung gestellt wird, vor der aktuellen Instruktion gesplittet. Der Rest des Basic Blocks wird zwischengespeichert, damit die Spaltung, falls notwendig, rekursiv fortgesetzt werden kann (rekursiver Aufruf der Funktion *„splitBasicBlock(…)“*). Bei der Spaltung wird eine Branch Instruktion eingefügt, der direkt an dieser Stelle die relevanten Metadaten angehängt werden.

Beispiel (vor und nachdem Splitting Pass):



Einblick in den Padding Pass:

Verantwortliche: Dennis Ogiermann, Kristina Petrow

Die Aufgabe des Padding Passes ist es die Basic Blöcke, deren Größe kleiner als der Mittelwert ist, zu finden und die fehlende Bytegröße zu ermitteln, die später dem Backend über die Metadaten mitgeteilt wird.

Zuerst läuft die Suche nach solchen Basic Blöcken. Wenn so ein Basic Block gefunden wurde, wird die Differenz zwischen der aktuellen Größe des Basic Blocks und die vom Mittelwert ermittelt.

Danach wird am Anfang diesen Basic Blocks eine NOP Instruktion eingefügt und über die Metadaten mittels der Funktion „*appendMetadata(…)*“ unter dem Parameter

„instructionSize“ die fehlende Bytegröße angehängt. So weiß das Backend um wie viele Byte das aktuelle Basic Block aufgepumpt werden soll.

Einblick in den Def-Use Analysis Pass:

Verantwortliche: Dennis Ogiermann, Kristina Petrow

In diesem Pass wird Liveness aller Register zu jedem Zeitpunkt analysiert und über Metadaten dem Backend übermittelt.

Liveness Analysis der Register muss ebenfalls durchgeführt werden, da MSP430 Architektur nur 16 Register unterstützt und LLVM IR dagegen unendliche Anzahl an virtuellen Registern bietet.

Über Metadaten einzelner Instruktionen, die von Fronend übergeben werden, werden noch zusätzlich zwei Vektoren übergeben. Das eine Array sagt, welche Register in der aktuellen Instruktion geschrieben und welche gelesen werden. Diese zwei Vektoren sind identisch aufgebaut. Die Zuordnung der Felder dieser zwei Vektoren zu den Registern erfolgt folgendermaßen:

Z.B.: Vektor „write“ {x, <wert>, x, ...} → Register {R0, R1, R2, ...} („x“→ kein Wert hinterlegt)

- Dieser Vektor sagt, dass in der aktuellen Instruktion ein Wert in Register R1 geschrieben wird

Z.B.: Vektor „read“ {x, x, <wert>, ...} → Register {R0, R1, R2, ...}

- Der Vektor „read“ sagt, dass in der aktuellen Instruktion ein Wert in Register R1 geschrieben wird

In diesem Pass werden 2 Vektoren *„UpExp“* und *„VarKill“* erzeugt (UpExp =Upwards Exposed Reads und VarKill = Written or killed variables). Zuerst folgt die Auslesung der einzelnen Arrayfelder aus den Metadaten (für „write“ und „read“ Variablen getrennt).

Aufbau: UpExp[b][c] und VarKill[b][c]

Diese beiden Vektoren sind zweidimensional, das erste Feld [b] gibt die Nummer des Basic Blocks an und das zweite Feld [c] gibt die Nummer des Registers (s. Bsp. oben) an, wobei b ein Integer und c ein Boolean ist. In der Zeit, wenn die Metadaten ausgelesen werden, wird an allen Werten eines Registers, das in den Instruktionen im aktuellen Basic Block vorkommt, ein ODER-Operator angewendet.

Beispiel*:* UpExp[2][1] ist auf „true“ gesetzt ◊ bedeutet, dass im Basic Block 2 Register 1 gelesen wird

Im Endeffekt erhalten wir zu jeden Basic Block ein Array, das uns sagt, welche Register geschrieben oder gelesen werden.

Backend

Nijat Aghayev

Einleitung

Die Hauptaufgabe der Backend war die vom Team Transformation erstellte LLVM Bitcode Datei (mit .bc Dateiendung) in die MSP430 Assembler Datei zu transformieren (mit .asm Dateiendung).

Programmverlauf

Das Program fordert eine Eingabedatei (als LLVM Bitcode Datei) , liest diese Datei und setzt voraus , dass die LLVM IR Instruktionen in dieser Datei mit entsprechenden Assembler Codes ausgestattet sind. Es iteriert über die Funktionen und Basic Blocks der Funktionen. In jedem Iterationsschritt werden die Metadaten aus LLVM IR Instruktionen gelesen und in die Ausgabedatei geschrieben.Wenn die Metadaten „NOP“ sind , wird eine Folge der zufälligen Assembler Codes generiert, die Anzahl der Zeile dieser Folge entspricht dem zweiten Operand der Metadaten. Wenn die LLVM IR Instruktion „BR“ ist , wird eine „JMP“ Assembler Instruktion erstellt , diese „JMP“ Instruktion spring zum nächsten Basic Block über. Alle „JMP“ Instrutionen in der LLVM Bitcode Datei werden komplett ignoriert.

Details über die Funktionen

Es gibt insgesamt 7 Funktionen in Backend.cpp . In folgenden werden auf die Details dieser Funktionen eingegangen.

bool generateASM(const std::string &FileName);

Eingabeparameter:

FileName als konstante string Referenz

Rückgabetyp:

bool

Die wichtigste Logik des Programms ist in dieser Funktion zu finden. Diese Funktion liest die Eingabedatei (LLVM Bitcode Datei) , iteriert über die Funktionen und die Basic Blocks der Funktionen , liest die Metadaten der Basic Blocks und schreibt sie in die Ausgabedatei. Wenn kein Fehler auftritt , wird true zurückgegeben, sonst false wird zurückgegeben.

void replaceAll(std::string &s , const std::string &search, const std::string &replace);

Eingabeparameter:

s als string Referenz

search als konstante string Referenz

replace als konstante string Referenz

Rückgabetyp:

void

Diese Funktion findet und ersetzt alle Vorkommen des strings „search“ mit dem string „replace“ im string „s“.

int getBBCount(std::unique_ptr<Module> &m);

Eingabeparameters:

m als std::unique_ptr<Module> Referenz

Rückgabetyp:

int

Diese Funktion zählt die Anzahl der Basic Blocks im LLVM Module „m“ durch und gibt das Ergebnis zurück.

void generateRandomCodeForNOP(int numberOfIRForInstr, raw_fd_ostream &asmOutput, std::string &indent);

Eingabeparameters:

numberOfIRForInstr als int

asmOutput als raw_fd_ostream Referenz

indent als string Referenz

Rückgabetyp:

void

Diese Funktion generiert eine Folge der zufälligen Assembler Instruktionen für „NOP“. Die Anzahl der Zeilen der Folge entspricht dem Eingabeparameter „numberOfIRForInstr“. Die erstellte Folge wird in die „asmOutput“ geschrieben.

std::pair<std::string,int> generateRandomRegAndValue();

Eingabeparameters:

Rückgabetyp:

std::pair<std::string,int>

Diese Funktion erstellt ein zufälliges Register von „r0“ durch „r15“ , eine zufällige Integer Zahl zwischen 1 und 255 und packt sie zusammen als std::pair<std::string,int> und gibt dieses pair zurück.

int randNum(int min, int max);

Eingabeparameters:

min als int

max als int

Rückgabetyp:

int

Diese Funktion erstellt eine zufällige Integer Zahl zwischen „min“ und „max“ (beide einschließlich) und gibt das Ergebnis zurück.

int main(int argc, char **argv);

Eingabeparameters:

argc als int

argv als char**

Rückgabetyp:

int

Diese Funktion ist Einsprungpunkt des Programms, sie ruft die Funktion generateASM() auf. Wenn etwas schief geht , wird Rückgabewert -1 zurückgegeben, sonst 0 wird zurückgegeben.

Evaluationen

Evaluation Frontend

Folgende Funktionen haben wir realisiert:

- Parsen und Liften der Adressmodi Registermode und Immediate Mode
- Die Metadaten-Weitergabe: Übergeben von Instruktionen mit Registern und Größe, sowie Schreib- und Lesezugriffe der Funktionen
- Parsen von Labeln und Erstellen der zugehörigen Basicblöcke
- Parsen und Liften des Sprunges ohne Kondition: JMP
- Parsen und Liften der Instruktionen mit zwei Registern (als Byte- oder Word-Instruktion):
  - → Add
  - → Sub
  - → Xor
  - → And
  - → Bis
  - → Mov
- Parsen und Liften der emulierte Instruktionen (als Byte- oder Word-Instruktionen):
  - → Clr
  - → Dec
  - → Decd
  - → Inc
  - → Incd
  - → Ret
- Parsen und Liften der Assembler Directives
  - → .global ...
  - → .text

Folgende Funktionen sind nicht vollständig realisiert worden:

- Das Liften von JMP-Instruktionen, die einen BasicBlock überspringen
- Das Liften der Adressmodi: Indexed Mode, Symbolic Mode, Absolute Mode, Indirect Register Mode, Indirect Autoinkrement Mode

Folgende Funktionen hätten noch realisiert werden müssen, um das Frontend zu vervollständigen:

- Das Liften von Jumpinstruktionen mit Bedingungen (Conditional Jumps)
- Das Realisieren des Carrybits (C-Bit), Overflowbits (V-Bit), Negativebits (N-Bit) und des Zerobits (Z-Bit), sowie das Liften der damit in Verbindung stehenden Funktionen
- Das Realisieren des Stack-Pointers(SP) und des Program-Counters (PC) und das Parsen und Liften der zugehörige Funktionen
- Parsen und Liften restlicher Instruktionen mit einem Register
- Parsen und Liften weiterer Directives

Evaluation Transformation

Die alle implementierten Passes außer Def-Use Analysis Pass erfüllen ihren Zweck. Def-Use Analysis könnte aufgrund des Zeitmangels nicht zur Ende implementiert werden, doch durch die Unvollständigkeit des Passes wird die einwandfreie Ausführung des Programms nicht beeinflusst. Nur im Falle der fehlenden oder unvollständigen Metadaten erfolgt ein kontrollierter Abbruch der Transformationsausführung.

Evaluation Backend

Alle Anforderungen von Projektleitern sind implementiert worden, nur ein Nachteil in dieser Implementation bezieht sich auf die Generierung von zufälligen Assembler Codes für NOP, hier habe ich nur zwei Operationen: ADD und SUB , und (CMP wenn die Anzahl der Zeilen der Folge ungerade ist) verwendet. Man könnte theoretisch diese zufällige Folge besser generieren, z.B. man könnte Eigenschaften des betrachteten Basic Block mitberücksichtigen.

Reflexion

- Hat sich die Aufgabenstellung oder Ihr Verständnis der Aufgabenstellung im Laufe der Zeit verändert? Wenn ja, wie?

Ja, beide haben sich verändert. Im Bezug auf Aufgabenstellung wurde Implementierungssprache von Python zu C++ verwechselt und LLVM wurde hinzugefügt.

Am Anfang des Projektes dachten wir , dass wir ein echtes LLVM Backend für MSP430 Mikrocontroller schreiben müssten, aber später wurde das mit einfacherem Backend ersetzt.

- Wie passend war die Planung des Projektverlaufs, insbesondere der Termine? Wie häufig musste die Planung angepasst werden? Warum? Was war Ihre Rolle dabei?

Die Planung des Projektverlaufs war in Ordnung. Manchmal musste die Termine wegen Zeitgründen abgesagt oder verschoben werden. Manchmal die Projektleiter hatten keine Zeit , manchmal wir Studenten.

- Welche Methoden haben Sie zur Unterstützung der gemeinsamen Planung genutzt und wie haben sie sich bewährt?

Wir haben agile Planungsmethode genutzt, und das war sehr gute Idee, weil während des Projektes einige Sachen hinzugefügt wurden, einige Sachen vereinfacht wurden, und wegen der Flexibilitätseigenschaft der agilen Methode könnten wir uns ganz schnell an neue Veränderungen adaptieren.

- Wie hat die Koordination bei der Projektarbeit funktioniert? Welche Probleme gab es? Wie wurden sie angegangen? Was war Ihre Rolle dabei?

Projektarbeit wurde auf drei unabhängige Unterprojekte aufgeteilt und dementsprechend wurden wir auch auf drei Untergruppen aufgeteilt. Deswegen funktionierte die Koordination reibungslos. Wir hatten die Dokumentation am Ende des Projektes geschrieben, und deswegen jeder Team verstand nur seinen Code im Laufe des Projektes . Wenn bei der Ausführung etwas schief gegangen war , war es schwierig festzustellen woran der Fehler lag, weil niemand mit dem ganzen Code bekannt war.

- In wieweit waren Sie ausreichend kompetent für die Mitarbeit? Welche Kompetenzen mussten Sie sich im Projektverlauf aneignen? Was haben Sie durch das Studienprojekt gelernt?

Wir hatten schon Erfahrungen mit Codierung und C++, aber Embedded System , MSP430 Mikrokontroller, und LLVM waren total neu zu uns. Die wichtigsten Kompetenzen die wir uns im Projektverlauf angeeignet haben sind Lernen unter Zeitdruck, Dokumentation schnell und effizient durchlesen. Wir haben viele Informationen über MSP430 und LLVM gelernt.

- Was haben Sie unternommen, um möglichst effizient und verlässlich im Projekt mitzuarbeiten?

Wir hatten Motivation und Entschlossenheit , und wir hatten unsere Probleme , Ideen in slack Platform miteinander geteilt. Deswegen war es relativ einfach Probleme und Schwierigkeiten zu bewältigen.

- Was würden Sie anders machen, wenn Sie mit der zum Projektende erworbenen Erfahrung die Aufgabe nochmals bearbeiten müssten?

Am Anfang des Projektes hatten wir Angst Fragen zu stellen , weil wir nicht blöd von anderen wahrgenommen werden möchten und zweitens erst in der Mitte des Semesters hatten wir das gesamte Bild über das Projekt. Wenn wir nochmals mit dieser Erfahrung die Aufgabe bearbeiten müssten, würden wir davon ausgehen, dass es nicht schlimm ist , Fragen zu stellen und von Projektleitern anzufordern, die gesamte Anforderungen konkret und schriftlich uns abzugeben.

- Welche Herausforderungen ergaben sich hin sichtlich der Gruppendynamik im Team und wie sind Sie damit umgegangen?

Ein Gruppenmitglied hatte Schwierigkeiten mit der Deutschen Sprache, und als Team müssten wir dies berücksichtigen und manchmal diskutierten wir Sachen in Englisch. Die zweite Herausforderung ergab sich, weil wir uns nicht aus Studium kannten, wir brauchten ein wenig Zeit um diese Herausforderung zu bewältigen.