Making Mobile Code both Safe and Efficient Compression und SafeTSA

BSc. Max Arends
arends@gmx.at
937 9835111

Zusammenfassung

In diesem Paper wird besprochen und werde Wege aufgezeigt, wie man mobilen Code besser optimiert und dessen Ausführbarkeit sicherer machen kann. Dafür werden zwei Ansätze vorgestellt: Compression von Bytecode und SafeTSA. Für sich allein und in Kombination ist es mittels der besprochenen Methoden möglich, die Performance von mobilem Code deutlich zu verbessern und die notwendige Evaluierung auf der Zielmaschine weitgehend zu minimieren.

Einleitung

Im der heutigen Zeit wird die Übertragung von mobilen Code immer wichtiger. Mobiler Code muss auf verschiedensten Plattformen, Architekturen und Betriebssystemen sicher laufen. Dies stellt eine große an die Architektur dar und wird im Beispiel von Java mittels der Java Virtual Machine gelöst.
Um sicher zu gehen, dass das Programm auf dem Zielrechner fehlerfrei ausgeführt wird, muss der Code vor dem ausführen gecheckt werden. Durch dieses checken wird der zu übertragende Bytecode allerdings verhältnismäßig gross, da auf Compileroptimierungen verzichtet werden muss, und das Überprüfen auf der Zielmaschine nimmt Prozessorzeit in anspruch, und dies nochdazu in dem Moment, in welchem der Benutzer wartet.
In diesem Paper soll nun darauf eingegangen werden, wie man den Zwischencode besser optimiert, dadurch also die größe des zu übertragenden Bytecodes verkleinert und die Ausführung sicherer machen kann.
Hierfür wird auf zwei verschiedene Ansätze eingegangen: einerseits auf die hocheffektive Compression von Abstract Syntax Trees und andererseits auf auf SafeTSA, welches auf einer referenzsicheren und typsicheren Variante der Static Single Assignment Form basiert.

Line of Defenses

Mobiler Code birgt Risiken in sich und führt kann zum Verlust von Vertraulichkeit, Verlust von Informationsintegrität, Verlust von Information oder eine Kombination aus diesen Eigenschaften führen.
Für dieses Problem gibt es Gegenmaßnahmen, die sogenannten Lines of Defenses (1)
Die erste Line of Defense gegen solche beschädigten Code ist „alle Computersysteme, alle Kommunikation zwischen Ihnen sowie die Information selbst gegen Angreifer sowohl physisch als auch logisch abzusichern“ (1).
Die zweite Line of Defense benutzt kryptographische Authentifizierungsmechanismen um mobilen Code zu entdecken, der nachträglich eingeschleust wurde.
Schließlich gibt es die dritte Line of Defense, welche es möglich macht, Angriffe, die sowohl die erste als auch die zweite Line of Defense erfolgreich durchbrochen haben, zu entdecken, zu korrigieren und im besten Fall ganz zu vermeiden.

1. Compression

Durch den vermehrten Einsatz von mobilem Code, ist es immer wichtiger diesen Code zu komprimieren und dessen Größe so klein wie möglich zu halten, vor allem wenn dieser über Netzwerke mit geringen Bandbreiten, wie beispielsweise Handynetze übertragen wird.
Wenn man sich die Entwicklung der letzten 10 Jahre ansieht, so kann man erkennen, dass die CPU Perforance exponential zu den Speicherzugriffszeiten zugenommen hat. Aus diesem Grund können hohe Kompressionsraten erreicht werden. (1)

1.1 Ansätze für Kompression von mobilem Code:

Es gibt verschiedene Ansätze mobilen Code zu komprimieren. Hierzu gehören (a) Methoden die cCompileroptimierungen durchführen und dadurch die Codegröße verkleinern, während der Code selber ausführbar bleibt, (b) Methoden welche die Codegröße verkleinern, indem statisische Mehtoden angewandt werden und (c) Methoden, welche abstrakte Syntax Trees (ASTs) komprimieren. (1)
Das Paper beschreibt Methode c, welche nach der Extraktion der AST aus dem Sourcecode

mittels einer probalistischen Codierung komprimiert werden.

1.2 Compressionstechniken für ASTs

Der Quellcode von Computerprogrammen sind ansich nichts anderes als Sätze formaler Sprachen, welche sich aus Wörtern von Buchstaben zusammensetzt.(1) Computerprogramme allerdings bestehen nicht aus Buchstaben, sondern aus abstrakten Abhängigkeiten und Grammatiken. AST beschreiben daher logische Abhängigkeiten wie beispielsweise While-Statements oder Zuweisungen.

1.3 Abstrakte Regeln und ASTs

Jeder AST entspricht einer abstrakten Grammatik (AG). Jeder AST Konten entspricht einer Regel. Insgesamt gibt es drei Arten von Regeln, welche AGs spezifizieren.
Aggregate Rules definieren AST Knoten, mit einer bestimmten Anzahl von Kindern (beispielsweise ein While-Statement).
Choice Rules definieren AST Knoten, welche genau ein Kind besitzen.
Die letzte Form sind String Rules, welche einen String definieren.
Figure 1 beschreibt den Aufbau dieser Regeln von ASTs.

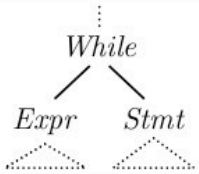
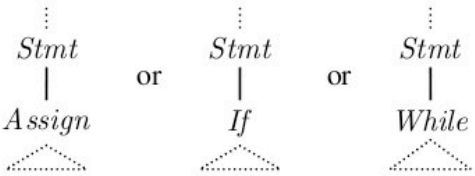
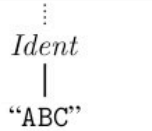
Form of Rule	Example Rule	Example Tree Fragment
Aggregate	$While \triangleq Expr\ \ Stmt$	
Choice	$Stmt \triangleq Assign \mid If \mid While$	
String	$Ident \triangleq$ STRING	

Figure 1. Abstract rules and corresponding abstract tree fragments

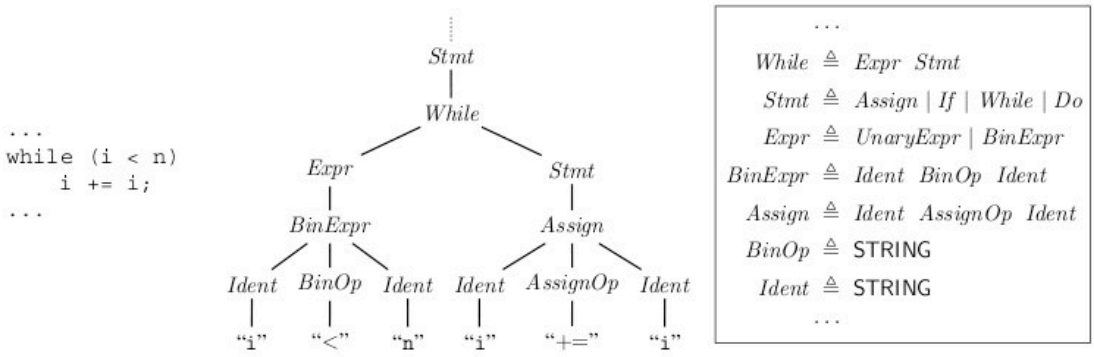


Figure 2. Source code snippet, its AST fragement, and some relevant AG rules

Figure 2 beschreibt den Aufbau von Abstract Syntax trees mittels dieser drei verschiedenen Regeln, anhand des linksstehenden Beispiels.

1.4 Serialisierung und Codierung von ASTs

Um ASTs zu speichern, bzw. zu übertragen, müssen diese serialisiert und codiert werden. Dafür werden diese Knoten von oben nach unten (top-down) abgearbeitet.
Da viele Anweisungen öfter vorkommen, ist die Information der Elternknoten redundant, man kann also bei mehrmaligem Ab- und wieder aufsteigen der einzlenen Ebenen davon ausgehen, dass die übergeordnete Ebene sich nicht verändert hat.
Die Grafik zeigt, wie die ASTs traversiert werden. Praktisch werden die Inhalte also in eine Liste gespeichert, und deren Auftreten mit einer anderen Liste verbunden.

Mittels arithmetischer Codierung und Prediction by Partial Match (PPM)werden die serialisierten ASTs nun codiert und komprimiert. Dadurch werden die Daten extrem komprimiert.

Resultate der Komprimierung:
Um die besprochene Herangehensweise zu bewerten, wird der Pughs Algorighmus verwendet, welcher die beste Compressions Ratio für Java Class Dateien aufweist (1). Die durschnittliche Ratio von mit Pugh komprimierten Dateien zu regulären Java Class Dateien ist 1:6,5. (1).

Bei Betrachtung der Resultate sieht man, dass unser Ansatz (PPM) bei allen getesteten Java Class Dateien besser abschneidet als Pugh, oder andere Komprimierungsansätze wie Gzip oder Bzip2.

Name	Class File	Gzip	Bzip2	Jar	Pugh	PPM	PPM/Pugh
`ErrorMessage`	388	256	270	409	209	84	0.40
`CompilerMember`	1321	637	641	792	396	227	0.58
`BatchParser`	5437	2037	2130	2189	1226	943	0.77
`Main`	13474	5482	5607	5627	3452	2909	0.85
`SourceMember`	15764	5805	5705	5920	3601	2477	0.69
`SourceClass`	37884	13663	13157	13975	8863	6428	0.73
`javac` (package)	92160	32615	30403	36421	18021	13948	0.78

Table 2. File sizes of compressed files for some classes from `javac` (all numbers in bytes).

2. The SafeTSA Representation

Wie zuvor Besprochen muss Code immer vor dem ausführen überprüft werden. Um den Bytecode also allgemein zu halten, muss dieser Code so allgemein wie möglich gehalten werden. SafeTSA zeigt einen ansatz, welcher Typsicherheit mit besserem Code darstellt.

2.1. Static Single Assignment Form (SSA)

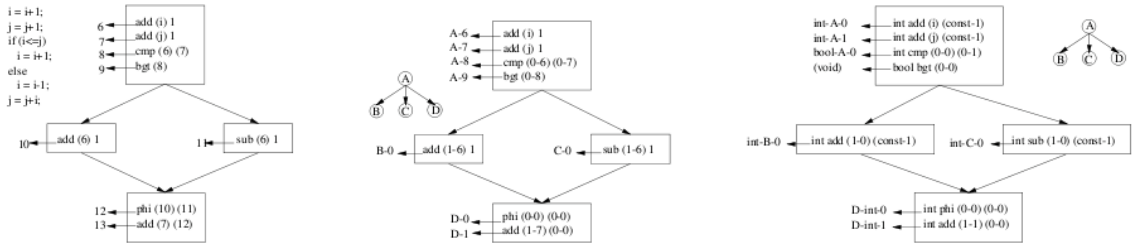
Ein Programm, welches in der Static Single Assignment (SSA) Form verfasst wurde, enthält keine Zuweisungen oder Registerbewegungen., jede Variable wird nur genau einmal statisch zugewiesen (2) Dadurch bietet sich die SSA Form auch so gut für Optimierungen an.

2.2 Type Seperation

Ein wichtiger Teil von SafeTSA ist „Type Seperation“. Während gewöhnliches SSA normalerweise eine unlimitierte Anzahl von Registern (genau eine pro Definition) hat, benutzt SafeTSA ein Model, wo es getrennte Register für jeden Typ gibt (1).

2.3. Referenzielle Integrität

SafeTSA stellt sicher, dass der Code nicht während der Ausführung an anderer Stelle verändert werden kann. Diesen wichtigen Teilaspekt für SafeTSA nennt man referenzielle Integrität. Figure 5 beschreibt wie ein Programm in SSA Form, in Reference-Safe SSA Form und Typed Reference-Safe SSA Form funktioniert. Durch den in Figure 5 c, gezeigten form, ist es nicht möglich, die zuvor zugewiesenen Register, welche zuvor beschrieben wurden, vor der durch MalCode zu überprüfen.



(a) in SSA Form

(b) in Reference-Safe SSA Form

(c) in Typed Reference-Safe SSA Form

Figure 5. An Example Program

2.4 Type und Safety

Es gibt nur 4 Operationen, welche den Speicher ändern dürfen. Diese sind newObject, newArray, setfield und setelt. Die Operationen dürfen dies nur in entsprechend der Typdeklarationen in der Typtabelle.(1) Die ist der Schlüssel für Typsicherheit.

Conclusio

Die in diesem Paper besprochenen Methoden machen es möglich, mobilen Code gleichzeitig sicher und effizient zu machen. Die gezeigten Resultate ist eine einzigartige Verbesserung für Lösungen, die eine Virtulle Maschine benötigen (1).

QUELLEN:
(1) W.Amme, N. Dalton, J. von Ronne: Making Mobile Code Both Safe Efficient
(2) W. Amme, N.Dalton, J von Ronne: SafeTSA: a type and referentially secure mobile-code representation based on static single assignment form.