* Begrüßung
* Vorstellung: Thomas Vogel
* Vergangenes Semester Teamprojekt
* Betreuung durch Lu
* Thema: „Implementing a Security Type System for Java“



* Motivation:
* Peseudocode-Beispiel 1:
  + erklären was „^H“ und „^L“ ist
  + kurz beschreiben
  + manche Funktionen nur Werte akzeptieren, die ein schwächeres Sicherheitslevel haben
* Überprüfung eigentlich simple, Wunsch nach nicht veröffentlichen eines Geheimnisses.



* Pseudocode-Beispiel 2:
  + kurz beschreiben
  + in diesem Fall nicht mehr simple, da man sich in einem Kontext befindet, der von einem starken Sicherheitslevel abhängt



* Übersicht:
  + Ziele des Teamprojekts
  + Worauf hat man bei Java-ähnlichen Sprachen zu achten, bzw. was sind die Konzepte um ein Security Type System umzusetzen.
  + Framework, das für die Analyse verwendet wird
  + Die konkrete Umsetzung der Analyse
  + Zusammenfassung



* Beginnen wir:
  + Was waren die Vorgaben, bzw. Ziele?



* Ziele des Teamprojekts
* Werkzeug, das Entwickler unterstützt sicher Anwendungen zu erstellen.
* Entwickler kann entsprechen den Quellcode annotieren und das Werkezeug überprüft auf Fehler
* Das Tool überprüft statisch, und ist somit die Grundlage für Gradual Java (wahlweise statische Überprüfung oder zur Laufzeit)
* Idee war vorhanden, dies mit einem bestimmten Framework umzusetzen



* Was muss man für ein Security Type System bei Java, bzw. Java-ähnlichen Sprachen beachten?
* Insbesondere bei den einzelnen Bestandteilen?



* Im folgenden zur Veranschaulichung: zwei Sicherheitslevel {Low, High}
* Low ist ein schwaches (kein Geheimnis) und High ein starkes Level (Geheimnis)
* Pseudocode: Felder, Rückgabewerte von Methoden und auch lokale Variablen sollen mit einem Sicherheitslevel versehen werden können



* Gleiches mit den Parametern, definierbar: Übergebene Argument muss aber schwächer sein als der Parameter definiert ist.
* Zu überprüfen ist jetzt der Programmfluss: Ist der Rückgabewert schwächer oder gleich stark wie das definierte Level?



* Konstanten haben das schwächste definierte Level; das Ergebnis einer Expression das stärkste Operanden Sicherheitslevel.



* bei Objekten schlägt das Sicherheitslevel der Instanz, das des Feldes, bzw. der Methode sofern es stärker ist.
* Wie in der Motivation gesehen spielt der Kontext ein große Rolle:  
  So muss dieser bei Rückgabe Statements berücksichtigt werden und verstärkt gegebenenfalls das Sicherheitslevel des zurückgegebenen Werts. Und darüberhinaus auch bei Zuweisungen.



* Zuweisungen:
  + Lokale Variablen: Sicherheitslevel richtet sich nach dem Level  
    des zugewiesenen Wertes.



* + Felder hingegen haben ein festes Sicherheitslevel und eine Zuweisung nur valide, wenn zugewiesener Wert ein schwächeres   
    oder gleich starkes Sicherheitslevel besitzt.



* + Array werden besonders behandelt: Grund hierfür ist, das es notwendig ist für das Array selbst aber auch für die beinhalteten Werte ein Sicherheitslevel zu definieren. Ober man definiert, dass Arrays nur Werte mit dem schwächsten Sicherheitslevel beinhalten.
  + Bei allen Zuweisung: Wenn Kontext stärker, dann schlägt es.



* Seiteneffekte müssen ebenfalls berücksichtigt werden:
* Insbesondere Schreibeffekte: sie haben einen Effekt auf ein spezifisches Sicherheitslevel --> je schwächer dieses ist, umso gravierender ist der Effekt!
* Ein Schreibeffekt entsteht durch Zuweisungen bzw. durch den Aufrufen von Methoden
* Methoden Schreibeffekte vs. Klassen Schreibeffekt (oftmals in Theorie übersehen)
* Auch hier muss der Kontext berücksichtigt werden: es dürfen innerhalb eines Kontextes nur Schreibeffekte auftreten, deren beeinflusstes Sicherheitslevel stärker oder gleich stark wie das des Kontextes ist.   
  z.B.: if (high) { lowField = value; }



* Soweit das Konzept hinter der Implementierung
* Für Analyse wurde das Soot Framework verwendet...



* Das Soot Framework wird durch Sable Research Group entwickelt und dient der Optimierung von Java Programmen.
* Die Quelle kann hierbei neben byte-code auch Java-Sourcecode sein.
* Durch das Framework wird aus dem Ausgangsformat Jimple Code generiert:
  + 3 address code: kommt mit drei Adresszugriffen pro Statement aus
  + Schleifen-Konstrukte werden in GOTO Anweisungen übersetzt.
* Die Entwicklung eigener Analysen mit Soot wird durch das Bereitstellen von abstrakten Analyse-Klassen ermöglich.



* Soweit der Background:
* Die entwickelte Security Type Analyse...



* Es handelt sich um eine statische intra-prozedurale Vorwärtsdatenflussanalyse. Gründe:
  + Jede Methode isoliert: Eingabe -> Rückgabe prüfen
  + Flexible Ausprägung der lokalen Variablen
* Um Analyse durchzuführen:
  + Annotationen: Schreibeffekte (Klasse und Methoden), Rückgabesicherheitslevel sowie Parametersicherheitslevel
  + Für lokale Variablen keine Annotationen möglich, deshalb muss Entwickler ein spezifische Klasse implementieren:
    - Stellt die definierten Sicherheitslevel in einer totalen Ordnung bereit
    - Id-Funktionen: die einen Wert nehmen (<= Sicherheitslevel) und den Wert entsprechend mit dem Sicherheitslevel der Id-Funktion zurückgibt.
* Library Methoden und Felder: schwächstes Level, bzw. stärkstes der Argumente



* Demo...



* Was die Analyse noch bewältig:
  + Security Polimorphismus, also variable Sicherheitslevel für die Parameter und den Rückgabewert
  + Generalisierte Sicherheitslevel Hierarchie --> momentan totale Ordnung...
  + Berücksichtigung von Exceptions
  + Vererbung ist momentan auf dem Prüfstand
  + Nebenläufigkeit
  + Array, die ein frei-definierbares Sicherheitslevel des Inhaltes haben
  + Einbindung der Library
  + Switch case --> jedoch wie if



* last but not least
* Die Zusammenfassung...



* Tool zum Entwickeln sicherer Anwendung mit Java
* Überprüft statisch auf Verletzung des Security Type Systems
* Was ein gute Grundlage für Gradual Type ist
* Und somit gibt es noch viel zu tun...