Sicherheitsanalyse mit JOANA

Jürgen Graf, Martin Hecker, Martin Mohr, Gregor Snelting[1]

Abstract: JOANA ist ein System zur Software-Sicherheitsanalyse, das bis zu 100kLOC volles Java mit Threads analysieren kann. JOANA basiert auf modernen Verfahren zur Programmanalyse und erzeugt deshalb wenig Fehlalarme. JOANA enthält einen "iRLSOD" Algorithmus, der probabilistische Nichtinterferenz garantiert. JOANA ist Open Source, braucht wenige Annotationen und lässt sich leicht bedienen. Der Artikel gibt einen Überblick über JOANA, erklärt das neue iRLSOD Verfahren, und präsentiert eine Fallstudie.[2]

Keywords: Software-Sicherheit, Programmanalyse, Nichtinterferenz

1 Übersicht

Klassische Techniken der Software-Sicherheit, wie etwa Zertifikate, analysieren nicht das tatsächliche Verhalten von Programmen, und können deshalb keine wirklichen Garantien über Vertraulichkeit und/oder Integrität einer (heruntergeladenen) Software machen. *Informationsflusskontrolle* (IFC) ist eine neue Sicherheitstechnik, die international intensiv untersucht wird, um klassische IT-Sicherheit zu ergänzen. IFC analysiert den Quelltext oder Bytecode einer Software, und findet entweder alle Lecks, oder kann eine echte Garantie geben, dass die Software Vertraulichkeit und Integrität respektiert.[3] Dabei bedeutet ein Vertraulichkeits-Leck, dass geheime Daten das öffentlich sichtbare Verhalten der Software beeinflussen, und ein Integritäts-Leck, dass kritische Berechnungen von aussen manipuliert werden können. Inzwischen wurden verschiedene praktisch einsetzbare IFC-Tools entwickelt, die auf verschiedenen Sicherheitskriterien und Analyseverfahren basieren, und sich deshalb in Sprachumfang ("volles Java"), Präzision ("keine Fehlalarme"), Skalierbarkeit ("große Programme"), Benutzbarkeit ("wenig Aufwand"), Korrektheit ("garantiert keine Lecks") und anderen Aspekten unterscheiden.

Das IFC-Tool JOANA wurde am KIT entwickelt und ist öffentlich verfügbar (siehe `joana.ipd.kit.edu`). JOANA analysiert Java Bytecode oder Android (Dalvik) Bytecode. Für alle Ein- und Ausgaben muss der Prüfingenieur angeben, ob sie geheim ("high") oder öffentlich ("low") sind.[4] JOANA kann volles Java mit beliebigen Threads behandeln (nur bei Reflection gibt es gewisse Einschränkungen). JOANA skaliert bis ca. 100kLOC und bietet – dank moderner kontext-, fluss-, und objekt-sensitiver Analysealgorithmen – hohe Präzision und wenig Fehlalarme. In nebenläufigen Programmen werden auch die tückischen

[1] KIT, Fakultät für Informatik, Am Fasanengarten 5, 76131 Karlsruhe, graf@kit.edu

[2] Abschnitt 1-3 des vorliegenden Artikels erschienen in englischer Sprache in: Checking probabilistic noninterference using JOANA. it – Information Technology 56(6), S. 280–287, 2014.

[3] IFC analysiert *nur* den Code einer Applikation. IFC analysiert *nicht* Hardware, Betriebssystem, Internet-Pakete etc. Deshalb muss IFC zusammen mit herkömmlichen Sicherheitstechniken verwendet werden.

[4] JOANA kann beliebig komplexe Verbände von Sicherheitsstufen verwenden, nicht nur *high* und *low*.

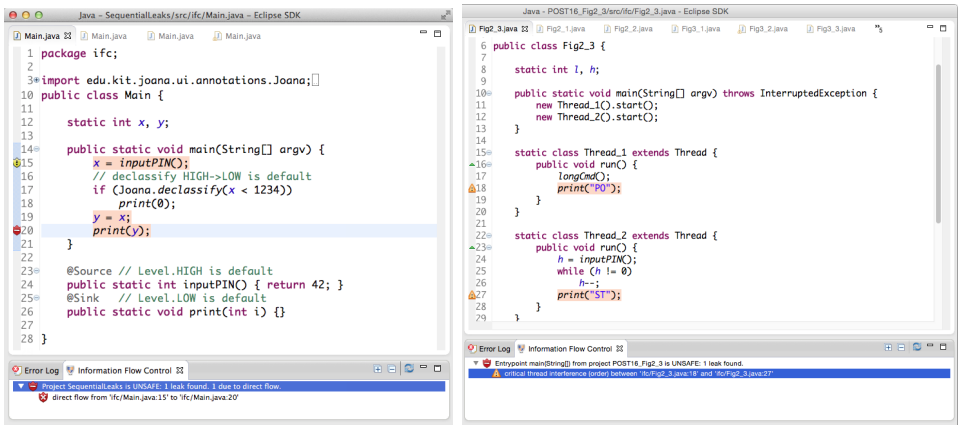

Abb. 1: JOANA GUI. Links: Beispiel mit Klassifikation, Deklassifikation, und illegalem Fluss. Rechts: Beispiel mit probabilistischem Leck.

probabilistischen Lecks entdeckt. Der JOANA Analysealgorithmus für sequentielle Programme wurde mit dem Maschinenbeweiser Isabelle verifiziert. JOANA wurde in diversen Fallstudien, u.a. Analyse eines E-Voting Prototyps angewendet.

Damit ist JOANA eines der wenigen international verfügbaren IFC-Tools, die reale Programme mit beliebigen Threads analysieren können. In diesem Beitrag sollen neue Ergebnisse im JOANA Projekt beschrieben werden, insbesondere das neue iRLSOD-Kriterium nebst Anwendungen. Die technischen Grundlagen sind im Detail in [HS09, GS15, WLS09, BGH+16] beschrieben, spezifische Anwendungen u.a. in [KTG12, KSTG14, MGH15, GHMS15].

2 JOANA Anwendung

Abbildung 1 zeigt das JOANA GUI. Links ist im Quelltextfenster der vollständige Java-Code für Beispiel (1) aus Abbildung 2 zu sehen. Sicherheitsstufen für Ein- und Ausgaben sind als Annotationen hinzugefügt, ebenso wie eine Deklassifikation für `x` im `IF`. Die IFC-Analyse markiert illegale Flüsse im Quelltext. Im Beispiel wird der illegale Fluss von der geheimen `inputPIN` zum `print(y)` markiert, während der illegale Fluss von `inputPIN` zu `print(0)` durch die Deklassifikation "künstlich erlaubt" wird. Weitere Details werden auf Wunsch angezeigt. Abbildung 1 rechts zeigt die Analyse des probabilistischen Lecks analog Beispiel (3) in Abbildung 2, das weiter unten erklärt wird. JOANA markiert die beiden Anweisungen, deren Ausführungsreihenfolge von geheimen Daten abhängen kann und so einem Angreifer einen Ansatzpunkt bieten.

JOANA bietet verschiedene Optionen für die Präzision der Programmanalyse (z.B. objekt-sensitive Points-to Analyse, zeit-sensitives Slicing), und sogar einen Autotuning-Mechanismus zum Finden einer Konfiguration, die keine Fehlalarme auslöst [Gra16]. JOANA verwendet das IBM Analyseframework WALA als Front End. JOANA konnte Sicherheits-

(1)

```
1  void main():
2    x = inputPIN();
3    // inputPIN is
4    // secret
5    if (x < 1234)
6       print(0);
7    y = x;
8    print(y);
9    // public output
```

(2)

```
1  void main():
2    fork thread_1();
3    fork thread_2();
4  void thread_1():
5    x = input();
6    print(x);
7  void thread_2():
8    y = inputPIN();
9    x = y;
```

(3)

```
1  void main():
2    fork thread_1();
3    fork thread_2();
4  void thread_1():
5    doSomething();
6    print('G');
7  void thread_2():
8    y = inputPIN();
9    while (y != 0)
10      y--;
11   print('I');
```

Abb. 2: Kleine, aber typische Beispiele für Lecks. (1) explizite und implizite Lecks. (2) possibilistisches Leck. (3) probabilistisches Leck.

(4)

```
1  void main():
2    h = inputPIN();
3    l = 2;
4    // l is public
5    x = f(h);
6    y = f(l);
7    print(y);
8
9  int f(int x)
10   {return x+42;}
```

(5)

```
1  void main():
2    fork thread_1();
3    fork thread_2();
4  void thread_1():
5    l = 42;
6    h = inputPIN();
7  void thread_2():
8    print(l);
9    l = h;
```

(6)

```
1  void main():
2    L = 0;
3    read(H2);
4    while (H2>0) H2--;
5    fork thread_1();
6    fork thread_2();
7  void thread_1():
8    l = 42;
9    h = inputPIN();
10 void thread_2():
11   print(l);
12   l = h;
```

Abb. 3: Sichere Programme, die durch mangelnde Analyse-Präzision Fehlalarme auslösen. (4) kontext-insensitive Analyse verschmilzt die beiden `f`-Aufrufe. (5) fluss-insensitive Analyse ignoriert die Anweisungsreihenfolge in `thread_2`. (6) klassische "low-deterministic security" verbietet jeden Low-Nichtdeterminismus.

garantien für schwierige Beispiele aus der Literatur herleiten, sowie ein prototypisches E-Voting System – inclusive Kryptographie – als sicher nachweisen [KTG12]. Die vollständige Analyse des bekannten HSQLDB Datenbanksystems dauerte einige Stunden auf einem Standard-PC.

3 Probabilistische Nichtinterferenz

IFC für sequentielle Programme muss explizite und implizite (Vertraulichkeits-)Lecks erkennen, die entstehen, wenn (Teile von) geheimen Daten an öffentliche Variablen kopiert werden; bzw. wenn geheime Daten den Kontrollfluss beinflussen. (siehe Beispiel (1) in Abbildung 2). IFC für nebenläufige Programme mit gemeinsamem Speicher ist eine viel größere Herausforderung, da zusätzliche possibilistische oder probabilistische Lecks entdeckt werden müssen, die durch Interleaving des Schedulers entstehen können. In Beispiel (2) führt die Scheduling-Reihenfolge 5,8,9,6 zum Ausgeben der geheimen PIN. In Beispiel (3) wird die PIN zwar nie ausgegeben, jedoch erhöht eine große PIN die Wahrscheinlichkeit, dass `'GI'` und nicht `'IG'` ausgegeben wird – denn die Laufzeit der Schleife hängt

von der PIN ab. Ein solches probabilistisches Leck kann einem geduldigen Angreifer viel Informationen über geheime Werte liefern; Abschnitt 5 zeigt eine größere Fallstudie.

IFC Algorithmen prüfen *Nichtinterferenz*. Nichtinterferenz bedeutet grob gesagt, dass öffentlich sichtbares Programmverhalten nicht durch geheime Werte beeinflusst werden darf. Alle korrekten Nichtinterferenz-Kriterien sind hinreichend, aber nicht notwendig, so dass zwar garantiert alle Lecks gefunden werden, es aber zu Fehlalarmen kommen kann. Die *Präzision* von IFC-Algorithmen ist deshalb wichtig, um Fehlalarme zu reduzieren (gänzlich eliminieren kann man sie aus Entscheidbarkeitsgründen nicht [Satz von Rice]). Gute IFC nutzt moderne Programmanalyse, um Präzision und Skalierbarkeit zu maximieren. So ist es etwa wichtig, die Reihenfolge von Anweisungen zu beachten ("Fluss-Sensitivität"), ebenso wie verschiedene Aufrufe derselben Funktion zu unterscheiden ("Kontext-Sensitivität") und verschiedene Objekte derselben Klasse zu unterscheiden ("Objekt-Sensitivität"). Beispiele (4) und (5) in Abbildung 3 zeigen Programme, die bei fluss- bzw kontext-insensitiver Analyse zu Fehlalarmen führen. Es ist aber absolut nichttrivial, skalierende Analysen zu entwickeln, die diese (und andere) Sensitivitäten haben; die lange Zeit populären Sicherheits-Typsysteme jedenfalls sind zumeist weder fluss- noch kontextsensitiv.

Probabilistische Nichtinterferenz verlangt nicht nur, dass es keine expliziten oder impliziten Lecks gibt; sondern zusätzlich, dass die *Wahrscheinlichkeit* für jedes öffentliche Verhalten durch geheime Werte nicht beeinflusst wird (vgl. Beispiele (3) und (6)). Gerade die bekannten Kriterien und Algorithmen für probabilistische Nichtinterferenz unterscheiden sich auf subtile Weise, mit jeweiligen Vor- und Nachteilen. So ist das einfachste Kriterium, die "Low-security observable determinism" (LSOD) [RWW94, ZM03] relativ einfach zu prüfen und funktioniert für jeden Scheduler, verbietet aber jeden öffentlichen Nichtdeterminismus – auch wenn dieser nie geheime Information preisgeben kann. Kriterien, die diese Einschränkung nicht haben (u.a. [SS00, Smi06, MS10]), stellen dafür – teils unrealistische – Anforderungen an den Scheduler oder das Programm. Obwohl LSOD seinerzeit nicht populär war, entschieden wir uns, JOANA auf LSOD-Basis zu realisieren – in der später bestätigten Hoffnung, dass sich durch Einsatz moderner Programmanalyse noch wesentliche Verbesserungen erzielen lassen.

JOANA verwendet deshalb sog. Programmabhängigkeitsgraphen (PDGs), die fluss- und kontextsensitiv sind, jedoch deutlich komplexer als Typsysteme. Heutige PDGs, gekoppelt mit Points-to Analyse und Exception Analyse sind sehr präzise, behandeln volles Java, und skalieren bis mehrere hunderttausend LOC. PDGs und ihr Zusammenhang mit IFC wurden oft beschrieben, so dass wir hier darauf verzichten – für Details siehe [HS09, GS15]. Untersuchungen zur Präzision und Skalierbarkeit der PDG-basierten IFC finden sich in [GS15, Ham10].

4 Das iRLSOD Kriterium

Bereits 2003 gab Zdancewicz ein einfach zu handhabendes Kriterium für LSOD an [ZM03]:

1. es darf keine expliziten oder impliziten Lecks geben;
2. keine öffentliche (low) Operation wird durch ein Data Race beeinflusst;

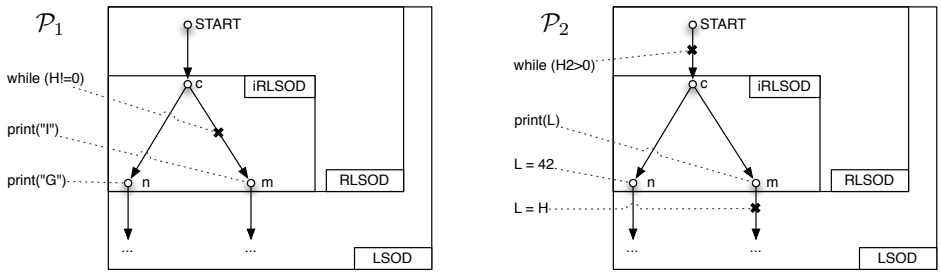

Abb. 4: Visualisierung von LSOD vs. RLSOD vs. iRLSOD. Skizziert sind die Kontrollflussgraphen für Beispiel (3) und Beispiel (6). n und m produzieren Low-Nichtdeterminismus, c ist der gemeinsame Dominator. LSOD verbietet jeglichen Low-Nichtdeterminismus; RLSOD erlaubt Low-Nichtdeterminismus der nicht von High-Anweisungen erreichbar ist; iRLSOD erlaubt Low-Nichtdeterminismus der von High-Anweisungen erreichbar ist, sofern letztere vor dem gemeinsamen Dominator liegen. Die markierten Bereiche zeigen, wo Low-Nichtdeterminimsmus weiterhin verboten ist; man erkennt dass iRLSOD wesentlich mehr Low-Nichtdeterminismus ohne Fehlalarme erlaubt als RLSOD oder gar LSOD.

3. es gibt keinen Low-Nichtdeterminismus.

Der letzte Punkt verbietet, dass jegliche Low-Operationen parallel ausgeführt werden – sogar wenn das nachweisbar sicher wäre. Giffhorn entdeckte in seiner Dissertation [Gif12], dass Fluss-Sensitivität der Schlüssel zu einer präzisen Implementierung des Zdancewicz-Kriteriums ist, und bewies die Korrektheit seines Verfahrens. Insbesondere können alle drei Bedingungen in natürlicher Weise mit PDGs und nebenläufigen Kontrollflussgraphen überprüft werden [GS15]; zusätzlich wird eine präzise Points-to Analyse und eine präzise "May happen in parallel" (MHP) Analyse verwendet. Damit profitiert IFC von den langjährigen, internationalen Anstrengungen zur Konstruktion präziser PDGs für volles Java.

Der Vollständigkeit halber sei Giffhorns LSOD Kriterium und Korrektheitstheorem angegeben, ohne dass wir auf die komplexen Begrifflichkeiten, den Beweis, oder die algorithmischen Herausforderungen eingehen können (siehe [GS15]). Das Kriterium realisiert die 3 Zdancewicz-Bedingungen mittels des PDG und Slicing. $cl(n)$ ist die Klassifikation (Low oder High) des PDG-Knotens bzw der Anweisung n, $BS(n)$ ist der (fluss-, kontext-, objekt-sensitive) Rückwärts-Slice im PDG für n.

Theorem. LSOD und also probabilistische Nichtinterferenz gilt, wenn für alle PDG Knoten n, n', n'' gilt:

1. $cl(n) = L \wedge cl(n') = H \implies n' \notin BS(n)$
2. $MHP(n,n') \wedge \exists v \in def(n) \cap (def(n') \cup use(n')) \wedge cl(n'') = L$
   $\implies n, n' \notin BS(n'')$
3. $MHP(n,n') \implies cl(n) = H \vee cl(n') = H$

Giffhorn entdeckte auch, dass man die Bedingung 3. abschwächen und gewissen Low-Determinismus erlauben kann. Wenn nämlich dieser Low-Nichtdeterminismus nicht im

Kontrollflussgraph von High-Operationen aus erreichbar ist, kann er kein probabilistisches Leck hervorrufen ("RLSOD"-Kriterium: Relaxed LSOD). So ist etwa der Low-Nichtdeterminismus in Beispiel (5) unschädlich, weil zwar die Reihenfolge von Anweisungen 5. und 8. i.a. nichtdeterministisch (nämlich schedulerabhängig) ist, diese Anweisungen aber nicht von High-Anweisungen (insbesondere 6.) erreichbar sind. Andererseits ist in Beispiel (3) der Low-Nichtdeterminismus von Anweisungen 6. und 11. sehr wohl von Anweisungen 8./9. erreichbar, das Programm deshalb unsicher (siehe Abschnitt 3). Beispiel (6) ist hingegen sicher, denn zwar ist der Low-Nichtdeterminismus in Anweisungen 8./11. von den High-Anweisungen 3./4. erreichbar, jedoch beinflussen 3./4. die Anweisungen 8. und 11. in identischer Weise, so dass ein Angreifer trotz des 8./11. Nichtdeterminismus nichts über `H2` erfahren kann!

Das letzte Beispiel führt auf das neue iRLSOD Kriterium. Es reicht zu prüfen, ob der High-Einfluss auf den Low-Nichtdeterminismus vor oder hinter dem *gemeinsamen Dominator* [Aho86] zweier nichtdeterministischer Low-Anweisungen n, m stattfindet. Wie in Abbildung 4 dargestellt, ist etwa im Beispiel (3) eine High-Anweisung auf einem der beiden Kontrollfluss-Zweige vom gemeinsamen Dominator $c = cdom(n,m)$ zu n bzw. m, also *hinter* c. Hingegen gilt für Beispiel (6), dass die High-Anweisung *vor* c liegt. iRLSOD verbietet nur solche High-Anweisungen, die hinter c, aber vor n bzw. m liegen. Da der gemeinsame Dominator in der Praxis weit hinter dem START-Knoten liegt, ergibt sich eine stark reduzierte Zahl von Fehlalarmen durch Low-Nichtdeterminismus; gleichzeitig werden weiterhin alle probabilistischen Lecks garantiert erkannt.

Der Vollständigkeit halber geben wir die modifizierte Bedingung (3) nebst Korrektheitsaussage für iRLSOD an; für Bedingung (2) kann man eine analoge iRLSOD-Version angeben. Details und Beweis finden sich in [BGH+16].

Theorem. Probabilistische Nichtinterferenz gilt, wenn für alle PDG Knoten n, n', n'' LSOD-Bedingung (1) und (2) gilt, und ferner:

3. $MHP(n,n') \wedge cl(n) = cl(n') = L \wedge c = cdom(n,n')$
$\implies \forall n'' \in \left((c \rightarrow^*_{CFG} n) \cup (c \rightarrow^*_{CFG} n')\right) : cl(n'') = L$

5 Fallstudie: E-Voting

In diesem Abschnitt zeigen wir die iRLSOD-Analyse eines experimentellen E-Voting Systems, das in Zusammenarbeit mit R. Küsters et al. entwickelt wurde. Die E-Voting Software soll beweisbar eine fundamentale kryptographische Eigenschaft aufweisen, nämlich die sog. *Computational Indistinguishability*. Um dies durch IFC zu unterstützen, entwickelten Küsters et al. einen Ansatz, bei dem zunächst kryptographische Funktionen durch eine Leerimplementierung ersetzt werden, die lediglich dieselben Schnittstellen wie die echte Krypto-Software aufweist. Kann man dann mit IFC Nichtinterferenz beweisen (d.h. es gibt keine Lecks zwischen Klartext, Schlüssel und Schlüsseltext), so gilt nach einem Theorem von Küsters die "computational indistinguishability" für die echte Implementierung [KTG12, KSTG14]. Für eine rein sequentielle Implementierung konnte JOANA problemlos Nichtinterferenz nachweisen.

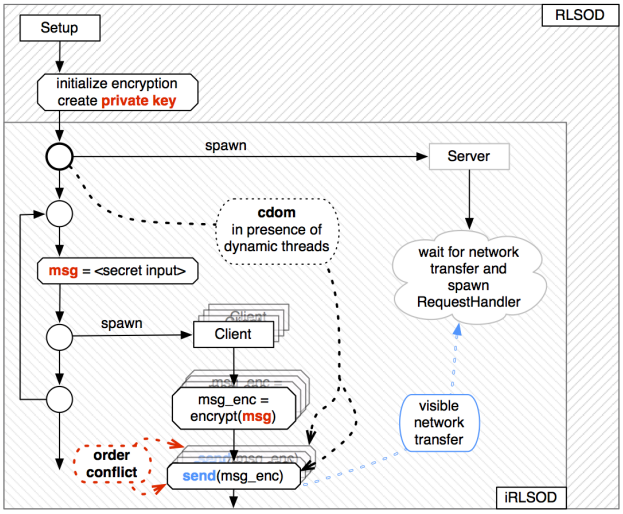

Abb. 5: Struktur des Kontrollflussgraphen für Server-Client basierten Nachrichtenaustausch mit Multithreading.

Eine neuere Implementierung verwendet eine Client-Server Architektur, und besteht aus einigen hundert LOC in 16 Klassen. Der interprozedurale Kontrollfluss ist in Abbildung 5 dargestellt; Abbildung 6 zeigt die relevanten Abschnitte des Quelltextes. Der Hauptthread startet in Klasse `Setup` in Zeile 3ff: Zunächst wird die Verschlüsselung durch Erzeugen von geheimem und öffentlichem Schlüssel initialisiert; dann wird ein einzelner `Server` Thread gestartet, bevor die Hauptschleife beginnt. In der Hauptschleife wird jeweils eine geheime (High) Nachricht gelesen, dann wird jeweils ein `Client` Thread gestartet, der diese verschlüsselt und über das Internet zum Server sendet. In diesem Beispiel gibt es also viele Instanzen des `Client` Threads, je einen pro Nachricht. Die Computational Indistinguishability verlangt, dass der Angreifer nicht zwischen verschiedenen (verschlüsselten) Nachrichteninhalten unterscheiden kann, und nach Küsters Theorem reicht es, Nichtinterferenz zu prüfen.

In diesem Beispiel sind High: (1) der Wert des `secret_bit` (Zeile 3), welches das Wahlverhalten codiert und über den Inhalt der Nachricht entscheidet,[5] und (2) der private Schlüssel (Zeile 33). Beide wurden in JOANA mittels einer `@Source` Annotation als HIGH markiert. Gemäß Angriffsmodell kann der Angreifer die versendeten Nachrichten lesen (aber nicht entschlüsseln), deshalb wurden Aufrufe von `sendMessage` (Zeile 73f) als LOW `@Sink` markiert. JOANA schließt explizite oder implizite Lecks sofort aus, findet im RLSOD-Modus aber zwei probabilistische Lecks – von denen sich eines mittels iRLSOD als Fehlalarm entpuppt.

Das erste potentielle Leck entsteht dadurch, dass die geheime Schlüsselinitialisierung evtl. den `sendMessage` Aufruf beeinflusst. Zur iRLSOD-Analyse dieses Lecks sei angemerkt,

[5] In diesem Beispiel wird angenommen, dass der Wähler nur zwischen 2 Parteien wählen kann, so dass ein Bit für das E-Voting reicht.

```
1  public class Setup {
2
3    public static void setup(@Source boolean secret_bit) { // HIGH input
4      // Public-key encryption functionality for Server
5      Decryptor serverDec = new Decryptor();
6      Encryptor serverEnc = serverDec.getEncryptor();
7      // Creating the server
8      Server server = new Server(serverDec, PORT);
9      new Thread(server).start();
10
11     // The adversary decides how many clients we create
12     while (Environment.untrustedInput() != 0) {
13       // determine the value the client encrypts:
14       // the adversary gives two values
15       byte[] msg1 = Environment.untrustedInputMessage();
16       byte[] msg2 = Environment.untrustedInputMessage();
17       if (msg1.length != msg2.length) { break; }
18
19       byte[] msg = new byte[msg1.length];
20       for(int i = 0; i < msg1.length; ++i)
21         msg[i] = (secret_bit ? msg1[i] : msg2[i]);
22
23       // spawn new client thread
24       Client client = new Client(serverEnc, msg, HOST, PORT);
25       new Thread(client).start();
26     }
27   }
28 }
29
30 public class KeyPair {
31   public byte[] publicKey;
32   @Source
33   public byte[] privateKey; // HIGH value
34 }
35
36 public final class Decryptor {
37
38   private byte[] privKey;
39   private byte[] publKey;
40   private MessagePairList log = new MessagePairList();
41
42   public Decryptor() {
43     // initialize public and secret (HIGH) keys
44     KeyPair keypair = CryptoLib.pke_generateKeyPair();
45     publKey = copyOf(keypair.publicKey);
46     privKey = copyOf(keypair.privateKey);
47   }
48
49   ...
50
51 }
52
53 public class Client implements Runnable {
54
55   private byte[] msg;        private Encryptor enc;
56   private String hostname;   private int port;
57   ...
58
59   @Override
60   public void run() {
61     // encrypt
62     byte[] mgs_enc = enc.encrypt(msg);
63
64     // send
65     long socketID = socketID = Network.openConnection(hostname, port);
66     Network.sendMessage(socketID, mgs_enc);
67     Network.closeConnection(socketID);
68   }
69 }
70
71 public class Network {
72
73   @Sink // LOW output
74   public static void sendMessage(long socketID, byte[] msg) throws NetworkError {
75     ...
76   }
77   ...
78 }
```

Abb. 6: Relevante Sourcecode-Ausschnitte des E-Voting Systems

dass durch die dynamischen Threads die Dominatorberechnung angepasst werden muss: der Dominator aller `sendMessage` Aufrufe (die als potentiell nichtdeterministische Low-Operationen zu betrachten sind) ist der Kopf der While-Schleife. Vor dieser Schleife liegt die Schlüsselinitialisierung, die als High klassifiziert ist (denn sie wird durch den geheimen Schlüssel beeinflusst).[6] Nach iRLSOD liegt sie aber vor dem Dominator und kann kein probabilistisches Leck verursachen – das "alte" RLSOD würde hier einen Fehlalarm erzeugen. Die Schlüsselinitialisierung ist also sicher, und iRLSOD – aber nicht RLSOD – kann das garantieren.

Ein weiteres probabilistisches Leck entsteht aber dadurch, dass der `encrypt` Aufruf vom geheimen Wert `msg` beeinflusst wird. Zwar kann durch die Krypto-Leerimplementierung kein Fehlalarm eines expliziten Lecks erfolgen, und auch eine Deklassifikation ist nicht notwendig. Jedoch kann nicht ausgeschlossen werden, dass die *Laufzeit* des Aufrufs von `msg` abhängt, so dass ein Leck analog Beispiel (3) entstehen könnte. Denn wenn die `encrypt` Laufzeit vom Nachrichteninhalt abhängt, wird das Scheduling statistisch[7] eine bestimmte Reihenfolge der Nachrichten hervorrufen, durch die ein Angreifer evtl. Rückschlüsse auf den Inhalt ziehen kann. JOANA entdeckt dieses probabilistische Leck, weil die iRLSOD-Bedingung verletzt ist: Der `encrypt`-Aufruf ist als High klassifiziert (vgl. Fußnote), liegt aber zwischen Dominator (Schleifenkopf) und Low-nichtdeterministischem `sendMessage`-Aufruf.

JOANA brauchte 5 Sekunden zur Analyse des gesamten Quelltextes. JOANA zeigt im Quelltext die nichtdeterministischen Anweisungen an nebst Hinweis, dass der Nichtdeterminismus sichtbares Verhalten beeinflussen kann. Der Prüfingenieur könnte daraufhin in diesem Beispiel allerdings entscheiden, dass das Leck nicht gefährlich ist! Wenn er z.B. weiss, dass der Nichtdeterminismus nur durch die `encrypt` Laufzeiten entstehen kann, und wenn er garantieren kann, dass die `encrypt` Laufzeit unabhängig vom Nachrichteninhalt ist, darf er das Leck ignorieren. Solche manuellen Entscheidungen (die auch als Deklassifikation im Quelltext vermerkt werden können) erfordern allerdings genaue Programmkenntnis und größte Vorsicht.

6 Verwandte Projekte

JIF [Mye99] ist eins der ältesten IFC Werkzeuge, aber verwendet einen speziellen Java-Dialekt und braucht viele Annotationen. Das kommerziell erfolgreichste IFC Tool ist vermutlich TAJ / Andromeda [TPF+09], das in der IBM AppScan Suite verwendet wird. Das Tool erkennt nur explizite Lecks, skaliert aber in den MLoc Bereich. FlowDroid [ARF+14] kann keine probabilistischen Lecks behandeln, bietet aber guten Support für den Android Life Cycle und Reflection. Für JavaScript wurden Kombinationen von statischer und dynamischer IFC vorgeschlagen [BRGH14].

[6] JOANA klassifiziert Anweisungen, die von High-Eingaben direkt oder indirekt kontroll- oder datenabhängig sind, automatisch ebenfalls als High [HS09].

[7] (i)RLSOD setzt einen echt probabilistischen Scheduler voraus [BGH+16], so dass sich tatsächlich bei verschiedener Laufzeit der `encrypt` Aufrufe statistische Aufruf-Muster bilden, die der Angreifer bobachten kann – auch wenn er den Nachrichteninhalt nicht entschlüsseln kann.

7 Schluss

In diesem Artikel haben wir neue Ergebnisse des JOANA-Projektes skizziert, und das neue iRLSOD-Kriterium zur probabilistischen Nichtinterferenz an Beispielen erklärt. Die tatsächliche Formalisierung nebst Korrektheitsbeweis ist sehr viel komplexer – hier seien nur die Probleme der Terminations-Sensitivität oder der Scheduler-Abhängigkeit erwähnt. Details zu PDG-basierter IFC und (i)RLSOD finden sich in [HS09, GS15, BGH+16].

Heute ist JOANA eines der wenigen international verfügbaren IFC Werkzeuge, das volles Java incl. Threads mit hoher Präzision behandeln kann und an realen Fallstudien erprobt wurde. Wir glauben, dass wir die weitverbreitete Skepsis gegenüber dem "Low-Deterministic Security" Ansatz mit iRLSOD widerlegt haben. Der nächste Schritt wird sein, JOANA an industrieller Software zu erproben.

Danksagung. JOANA wird seit vielen Jahren von der DFG unterstützt (insbesondere durch DFG SPP 1496 "Reliably secure software systems"), sowie vom BMBF im Rahmen des Software-Sicherheits-Kompetenzzentrums KASTEL.

Literaturverzeichnis

[Aho86] Ullman Aho, Sethi. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[ARF+14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Traon, D. Octeau und P. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. PLDI*, Seite 29, 2014.

[BGH+16] Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr und Gregor Snelting. On Improvements Of Low-Deterministic Security. In *Proc. Principles of Security and Trust POST*, 2016. to appear.

[BRGH14] A. Bichhawat, V. Rajani, D. Garg und C. Hammer. Information Flow Control in WebKit's JavaScript Bytecode. In *Proc. Principles of Security and Trust POST*, Seiten 159–178, 2014.

[GHMS15] Jürgen Graf, Martin Hecker, Martin Mohr und Gregor Snelting. Checking Applications using Security APIs with JOANA. Juli 2015. 8th International Workshop on Analysis of Security APIs.

[Gif12] Dennis Giffhorn. *Slicing of Concurrent Programs and its Application to Information Flow Control*. Dissertation, Karlsruher Institut für Technologie, Fakultät für Informatik, Mai 2012.

[Gra16] Jürgen Graf. *Information Flow Control with System Dependence Graphs — Improving Modularity, Scalability and Precision for Object Oriented Languages*. Dissertation, Karlsruher Institut für Technologie, Fakultät für Informatik, 2016. Forthcoming.

[GS15] Dennis Giffhorn und Gregor Snelting. A New Algorithm For Low-Deterministic Security. *International Journal of Information Security*, 14(3):263–287, April 2015.

[Ham10] Christian Hammer. Experiences with PDG-based IFC. In F. Massacci, D. Wallach und N. Zannone, Hrsg., *Proc. ESSoS'10*, Jgg. 5965 of *LNCS*, Seiten 44–60. Springer-Verlag, Februar 2010.

[HS09] Christian Hammer und Gregor Snelting. Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *International Journal of Information Security*, 8(6):399–422, December 2009.

[KSTG14] Ralf Küsters, Enrico Scapin, Tomasz Truderung und Jürgen Graf. Extending and Applying a Framework for the Cryptographic Verification of Java Programs. In *Proc. POST 2014*, LNCS 8424, Seiten 220–239. Springer, 2014.

[KTG12] Ralf Küsters, Tomasz Truderung und Jürgen Graf. A Framework for the Cryptographic Verification of Java-like Programs. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE Computer Society, Juni 2012.

[MGH15] Martin Mohr, Jürgen Graf und Martin Hecker. JoDroid: Adding Android Support to a Static Information Flow Control Tool. In *Proceedings of the 8th Working Conference on Programming Languages (ATPS'15)*, Lecture Notes in Informatics (LNI) 215. Springer Berlin / Heidelberg, Februar 2015. to appear.

[MS10] Heiko Mantel und Henning Sudbrock. Flexible Scheduler-Independent Security. In *Proc. ESORICS*, Jgg. 6345 of *LNCS*, Seiten 116–133, 2010.

[Mye99] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Seiten 228–241, New York, NY, USA, 1999. ACM Press.

[RWW94] A. W. Roscoe, Jim Woodcock und L. Wulf. Non-Interference Through Determinism. In *ESORICS*, Jgg. 875 of *LNCS*, Seiten 33–53, 1994.

[Smi06] Geoffrey Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 14(6):591–623, 2006.

[SS00] Andrei Sabelfeld und David Sands. Probabilistic Noninterference for Multi-Threaded Programs. In *Proc. CSFW '00*, Seite 200, Washington, DC, USA, 2000. IEEE Computer Society.

[TPF+09] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan und Omri Weisman. TAJ: effective taint analysis of web applications. In *Proc. PLDI*, Seiten 87–97, 2009.

[WLS09] Daniel Wasserrab, Denis Lohner und Gregor Snelting. On PDG-Based Noninterference and its Modular Proof. In *Proc. PLAS '09*. ACM, June 2009.

[ZM03] Steve Zdancewic und Andrew C. Myers. Observational Determinism for Concurrent Program Security. In *Proc. CSFW*, Seiten 29–43. IEEE, 2003.