Checking Applications using Security APIs with JOANA

Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting

Karlsruhe Institute of Technology
`[graf,hecker,martin.mohr,gregor.snelting]@kit.edu`

Abstract

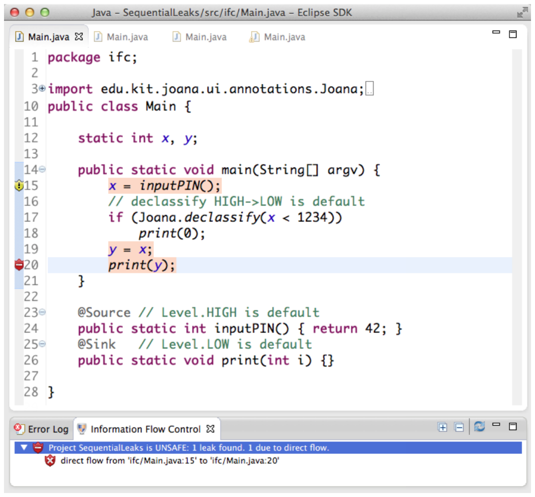
JOANA is a tool for software security analysis, checking up to 100kLOC of full multi-threaded Java. JOANA is based on sophisticated program analysis techniques and very precise. JOANA includes a new algorithm guaranteeing probabilistic noninterference, named RLSOD. JOANA needs few annotations, is open source, and was applied in several case studies.

The current extended abstract discusses the analysis of security APIs using JOANA. In particular, we practically demonstrate a method which guarantees that code using a cryptographic API does not contain confidentiality leaks. The method is backed by a theorem from Küsters [6].

1 JOANA

Information flow control (IFC) is a fine-grained analysis of software source or machine code, which uncovers all security leaks *within* a program, or provides a true guarantee about integrity resp. confidentiality. IFC is typically based on some notion of noninterference, which demands that public behaviour is not influenced by secret data and thus guarantees confidentiality.

The IFC tool JOANA (`joana.ipd.kit.edu`) can handle full Java bytecode with arbitrary threads, scales to ca. 100kLOC, and empirically demonstrated high precision [3, 2]. JOANA is based on a stack of sophisticated program analysis algorithms (pointer analysis, exception analysis, program dependence graphs). JOANA minimizes false alarms through flow-, context-, object-, field-, time-, and lock-sensitive analysis techniques. JOANA allows declassification along sequential information flows. In concurrent programs, all possibilistic and probabilistic leaks are discovered by a new algorithm for probabilistic noninterference, called RLSOD [1]. Soundness of JOANA's sequential IFC was machine-checked in Isabelle [7].



```
package ifc;

import edu.kit.joana.ui.annotations.Joana;
public class Main {

    static int x, y;

    public static void main(String[] argv) {
        x = inputPIN();
        // declassify HIGH->LOW is default
        if (Joana.declassify(x < 1234))
            print(0);
        y = x;
        print(y);
    }

    @Source // Level.HIGH is default
    public static int inputPIN() { return 42; }
    @Sink   // Level.LOW is default
    public static void print(int i) {}

}
```

```
Project SequentialLeaks is UNSAFE: 1 leak found. 1 due to direct flow.
    direct flow from 'ifc/Main.java:15' to 'ifc/Main.java:20'
```

Figure 1: JOANA screenshot demonstrating classification, declassification, and illegal flow.

1.1 Application of JOANA

Figure 1 shows the JOANA plugin for Eclipse. In the source code window, the full source for example (1) from Figure 2 can be seen. Security level annotations for input and output are added, as well as a declassification of `x` in the IF condition. Once the analysis is activated, illegal flows are highlighted in the source code. In the example, the illegal flow from the

(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   x = 0;
6   print(x);
7 void thread_2():
8   y = inputPIN();
9   while (y != 0)
10      y--;
11  x = 1;
12  print(2);
```

(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;
```

Figure 2: Small but typical leaks and IFC precision problems. Programs 1 – 3 leak secret data to public output. (1) Explicit and implicit leaks, (2) possibilistic leak, (3) probabilistic leak. Programs 4 and 5 are secure, but only a precise analysis will see this. (4) context-insensitive analysis will generate a false alarm because calls to `f` are merged, (5) flow-insensitive analysis will generate false alarm because statement order in `thread_2` is ignored.

secret `inputPIN` via `x` via `y` to the public `print(y)` can be seen; due to the declassification, the flow to `print(0)` has been suppressed. Full details on an illegal flow are available on demand. JOANA offers various options for analysis precision (e.g. object-sensitive points-to analysis, time-sensitive backward slicing). JOANA analyses Java bytecode and uses IBM's WALA analysis frontend; recently, a frontend for Android bytecode was added.

JOANA was able to provide security guarantees for difficult examples from the literature, and could analyse e.g. the complete source of the HSQLDB database. More interesting is perhaps the successful analysis of an experimental e-voting system developed by Küsters et al [6, 5].

1.2 Probabilistic Noninterference

In this short contribution, we will not explain technical details of our IFC analysis (for details, see [3, 1]). But we will at least present a few examples, illustrating the power and precision of JOANA's machinery.

IFC for sequential programs must discover explicit and implicit leaks, which arise if (parts of) secret values are copied to public variables, resp. if secret values influence control flow (see example (1) in Figure 2). IFC for multi-threaded programs must additionally prevent leaks which arise from the interleaving of concurrent threads: possibilistic leaks may or may not occur depending on a specific interleaving, while probabilistic leaks exploit the probability distribution of interleaving orders. Example (2) in Figure 2 has a possibilistic leak, e.g., for interleaving order $5, 8, 9, 6$, which causes the secret PIN to be printed on public output. Example (3) has no possibilistic, but a probabilistic leak, because the PIN's value influences the running time of the loop, which may influence the interleaving order of the two assignments to x. Thus the probabilities of public outputs "0" resp. "1" depend on the secret PIN.

Most IFC approaches check some form of noninterference, and to this end classify program variables, input and output as high (secret) or low (public). Noninterference in its simplest form then demands that variations in secret input data do not cause variations in public output data. For concurrent programs with threads, *Probabilistic Noninterference* (PN) is the established security criterion. PN explicitly allows nondeterminism in programs and demands that the probability of any observable behaviour is not influenced by secret values. It is difficult to guarantee PN, as IFC must in principle check all possible interleavings and their impact on execution probabilities.

In JOANA, a new "RLSOD" algorithm for PN is used, which avoids soundness problems or unrealistic restrictions of previous approaches [1].

2 Checking Security APIs

In the following, we will demonstrate how JOANA can be used to guarantee integrity and/or confidentiality for programs relying on a security API. JOANA does not check functional properties (e.g. cryptographic properties), nor does it verify API usage protocols. But it can easily provide two important IFC properties, namely that the security API plus the code using it do not contain integrity or confidentiality leaks.

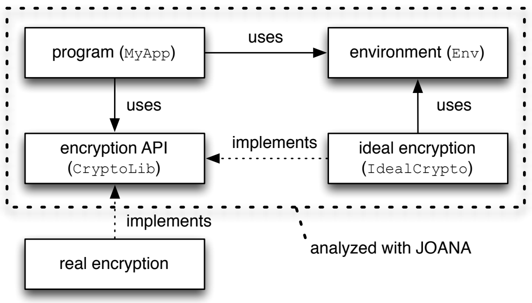


Figure 3: Setup for analyzing a program using a crypto API with JOANA.

We will illustrate our approach using a specific example, namely a program using public key encryption via a cryptographic library. This example is a stripped down variant of our previous work on verified secure client server communication [6, 4]. Traditionally, handling of cryptography in IFC was awkward. Declassification had to be used, because IFC does not know about crypto properties such as IND-CCA, and thus complains about a leak along any crypted data flow. Declassifications manually surpress such false alarms – an error-prone method, which completely relies on the engineer's competence and trustworthiness.

Küsters [6] was the first to demonstrate how cryptography and IFC can be soundly combined without declassification. Roughly speaking, all encryption operations in the source code are replaced by so-called *ideal* implementations that do not generate any flow. Then confidentiality is checked using an IFC tool such as JOANA. If no additional leaks occur, Küster's theorem states that the original program using the non-ideal cryptographic libary is also secure – provided the crypto implementation is "correct" (i.e. fulfills IND-CCA2, see [4] for details).

In the following, we will demonstrate this approach from a practical viewpoint. Figure 3 provides a general overview of our setup. We plan to verify a program (Figure 5) that uses an API for public key encryption as shown in Figure 4a. Our goal is to verify that a program using this API does not leak any sensitive information about the content of the unencrypted message as well as the private key. Therefore we implement an ideal variant of the crypto API that contains no information flow between the original and the encrypted messages (Figure 4c) and an environment class (Figure 4b) —containing security annotations for JOANA— that acts as source of secret (*high*) or random (*low*) input to the program.

In the ideal variant we use a randomly generated byte sequence to pose as the encrypted message. That way the encrypted message contains no information of the content of the original message. This allows our IFC analysis to detect the absence of illicit flows in the main program as long as sensitive operations only depend on the content of the encrypted message. However we still want to capture that once the encrypted message is decrypted, the result again contains sensitive information. Therefore we use a dictionary that maps the encrypted message to the original one. This way a call to `decrypt` can return the original message.

The environment class `Env` contains helper methods that emulate the input of secret and random information. The method bodies are left out for brevity, the actual implementations of `secretBytes` and `randomBytes` ensure that the content of their return values depends on

```
interface CryptoLib {
  byte[] encrypt(byte[] in, byte[] publKey);
  byte[] decrypt(byte[] in, byte[] privKey);
  KeyPair generateKeyPair();
}

class KeyPair {
  byte[] publ;
  byte[] priv;
}
```

(a) Interface of the public key encryption API.

```
class Env {
  @Source(label = Level.HIGH)
  static byte[] secretBytes(...) { ... }
  static byte[] randomBytes(...) { ... }
}

class Network {
  @Sink(label = Level.LOW)
  static void send(byte[] data) { ... }
}
```

(b) Part of the environment and networking class with IFC annotations.

```
class IdealCrypto implements CryptoLib {
  private Map dict; // maps random bytes to message

  byte[] encrypt(byte[] in, byte[] publKey) {
    byte[] random = Env.randomBytes(in.len, publKey);
    dict.addEntry(random, in);
    return random;
  }

  byte[] decrypt(byte[] in, byte[] privKey) {
    if (dict.contains(in)) {
      return dict.find(in);
    }
    return Env.randomBytes(in.len, privKey);
  }

  KeyPair genKeyPair() {
    KeyPair keys = new KeyPair();
    keys.publ = Env.randomBytes();
    keys.priv = Env.secretBytes(publ);
    return keys;
  }
}
```

(c) Ideal variant of the public key encryption API of Figure 4a.

Figure 4: Setup with *ideal* implementation ready for analysis with JOANA.

```
1  class MyApp {
2    public void main() {
3      // initial setup
4      CryptoLib crypto = new IdealCrypto();
5      byte[] message = Env.secretBytes();
6      // start of normal program to check for illegal flow
7      KeyPair keys = crypto.genKeyPair();
8      byte[] encrypted = crypto.encrypt(message, keys.publ);
9      Network.send(message);              // explicit leak
10     Network.send(encrypted);            // no leak
11     if (message[0] > 23) {
12       Network.send(encrypted);          // implicit leak
13     }
14     byte[] decrypted = crypto.decrypt(encrypted, keys.priv);
15     Network.send(decrypted);            // explicit leak
16   }
17 }
```

Figure 5: Program using an ideal variant of the crypto API with the setup provided by Figure 4.

the data passed to them as parameters. The annotation at `secretBytes` tells JOANA that the return value is considered *high* information. A similar annotation is made in the networking class to mark any data sent over the network as *low* observable.

With this setup in place we can use JOANA to check if no *high* information is leaked in our application shown in Figure 5. JOANA automatically detects several leaks in the example: 2 explicit and 1 implicit leak. It also detects that line 10 is not critical as it sends the encrypted message. After removing lines 9, 12 and 15, the application uses the crypto API correctly as JOANA is able to verify the absence of unwanted information leaks.

3 Conclusion

Today, JOANA is one of the very few IFC tools worldwide which can handle full Java with unlimited threads, and – thanks to the underlying stack of sophisticated program analysis – offers good precision and scalability. As we have seen, JOANA can also be used to analyse security APIs. In particular, usage of cryptographic APIs can be checked for integrity or confidentiality leaks.

Acknowledgments. This work was funded by the DFG under the project in the priority program RS3 (SPP 1496) and by the BMBF under the KASTEL competence center for applied IT security technology.

References

[1] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, April 2014. To appear. Electronic preprint at `http://link.springer.com/article/ 10.1007%2Fs10207-014-0257-6`.

[2] Christian Hammer. Experiences with PDG-based IFC. In F. Massacci, D. Wallach, and N. Zannone, editors, *Proc. ESSoS'10*, volume 5965 of *LNCS*, pages 44–60. Springer-Verlag, February 2010.

[3] Christian Hammer and 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.

[4] Ralf Küsters, Enrico Scapin, Tomasz Truderung, and Jürgen Graf. Extending and applying a framework for the cryptographic verification of java programs. In *Proc. POST 2014*, LNCS 8424, pages 220–239. Springer, 2014.

[5] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. A Hybrid Approach for Proving Noninterference of Java Programs. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, 2015. to appear.

[6] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A framework for the cryptographic verification of Java-like programs. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE Computer Society, June 2012.

[7] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On PDG-based noninterference and its modular proof. In *Proc. PLAS '09*. ACM, June 2009.