Noname manuscript No.
(will be inserted by the editor)

A Tool for Static Detection of Timing Channels in Java

Alexander Lux · Artem Starostin

Received: date / Accepted: date

Abstract A timing attack exploits the variance in the running time of a crypto-algorithm's implementation in order to infer confidential information. Such a dependence between confidential information and the running time, called a timing channel, is often caused by branching of the control flow in the implementation's source code with branching conditions depending on the attacked secrets. We present the *Side Channel Finder*, a static analysis tool for detection of such timing channels in Java implementations of cryptographic algorithms.

Keywords Timing channels · Timing attacks · Information flow security · Static analysis · IDEA

1 Introduction

A cryptographic mechanism based on algorithms which are even proved to be secure may become vulnerable after it is implemented in some programming language and run on an actual computer system. Side channel attacks are based on the fact that by observing the implementation's behavior which is not modeled by the underlying cryptographic algorithm the attacker can infer confidential data, e.g., a secret key. Therefore, when developing a cryptographic mechanism it is desirable to check whether its actual implementation opens up side channels.

One possibility to launch a side channel attack is to exploit the variance in the running time of a crypto-algorithm's implementation, called a *timing channel*. First studies of timing attacks on cryptographic schemes, including Diffie-Hellman and RSA, date back to mid 1990s [19]. Since then, they have been practically demonstrated [11], optimized [28], and evaluated [30]. A significant part [19,18,13,27,11,6,9,17,32,29,31] of timing attacks reported in the literature exploits the difference in the running time of crypto-algorithms' implementations which is caused by conditional branches or loops where conditions depend on the attacked secrets.

A timing channel constitutes *information flow* [21] from confidential inputs of a cryptographic algorithm to observations about the running time of its implementation. Information flow policies are means to specify that such information transfer is undesirable. The research area of static information flow control (see Section 2) focuses, among other, on using static program analysis and transformations to enforce information flow policies.

We present the *Side Channel Finder* in the version 1.0 (short SCF 1.0), a static analysis tool for detection of timing channels in Java implementations of cryptographic algorithms. The main purpose of SCF 1.0 is to support a programmer of a crypto-algorithm's implementation in assessing his code for that it is not vulnerable to a class of timing channel attacks. This class consists of attacks caused by branching of the control flow on data which depends on the confidential inputs. The branching may occur due to conditional statements, loops, or polymorphic method calls. The tool lets the programmer specify which input of an implemented algorithm constitutes a secret that must not be leaked, especially not through timing channels. These specifications are a part of an information flow policy which assigns security levels *high* and *low*, representing confidential and public data, respectively, to object fields,

A. Lux · A. Starostin
Modeling and Analysis of Information Systems (MAIS)
Computer Science Department, TU Darmstadt
Hochschulstraße 10, 64289 Darmstadt, Germany
E-mail: {lux, starostin}@mais.informatik.tu-darmstadt.de

Appeared in:
Journal of Cryptographic Engineering 1(4):303-313

method parameters, and return values. SCF 1.0 then analyzes the given program code by checking whether the control flow potentially depends on the confidential inputs. When this is the case, the ultimate goal of the Side Channel Finder is to perform automatic program transformations for elimination of the disclosed timing channels. This last step is currently a work in progress.

The Side Channel Finder is implemented as a plugin for Eclipse [5], a popular IDE for Java developers.

We applied SCF 1.0 to analyze several existing implementations of cryptographic algorithms. Our studies include the open-source libraries *FlexiProvider* (version 1.7p0) [3] and *GNU Classpath* (version 0.98) [2]. In the paper we illustrate how SCF 1.0 finds a timing channel in the implementation of the IDEA algorithm in FlexiProvider. We describe an experimental setup in which an attack exploiting this timing channel could be constructed. Then, we show how a part of the secret key could be revealed by an attacker through the timing channel discovered by SCF 1.0. The details of applying SCF 1.0 to the AES implementation of FlexiProvider and to the DES implementation in GNU Classpath can be found in a technical report [22].

Related Work

To the best of our knowledge, the Side Channel Finder 1.0 is the first tool for static detection of timing channels in Java. The related work for this effort could be grouped into three categories: (i) experimental timing channel analysis, (ii) static information flow security with respect to timing channels, and (iii) tools for static information flow control, in general. Since Section 3 discusses experimental timing attacks in detail, here we provide references only for the two remaining categories. From our point of view, the connection between them has room for improvement: the theory for static detection and transformation of timing channels did not lead to implementation in tools for real programming languages, whereas the mainstream tools for information flow control do not consider timing channels. We believe, that the Side Channel Finder project will finally fill this gap.

Agat [7] presents a simple approach for detection and elimination of timing leaks in C-like programs by using a security type system and program transformation, respectively. The approach was only exemplarily implemented [8] for a subset of Java bytecode without objects. Molnar et al. [24] suggest and realize transformation of timing leaks in C programs (without function calls and pointers) by encoding conditional branches into assignments of expressions, thus, making the control flow independent of branching conditions. Barthe, Rezk, and Warnier [10] introduce a transaction-based program transformation method for elimination of timing leaks in sequential object-oriented programs with exceptions. The implementation of the technique is not reported.

We are aware of four static information flow control tools: Jif [4] (introduced in [25]), FlowCaml [1], SPARK Examiner [12], and Joana [16] for Java, Caml, Ada, and Java bytecode, respectively. None of them considers information flow with respect to timing channels.

Outline

In Section 2 we recap the research area of static information flow control which comprises the methodology for the analysis performed by the Side Channel Finder. In Section 3 we discuss timing attacks from the literature and identify a class of attacks which exploit the control flow branching. Section 4 presents the features and design of SCF 1.0. In Section 5 we consider how SCF 1.0 finds a timing channel in the FlexiProvider's IDEA implementation. In Section 6 we conclude and identify a vision for future versions of the Side Channel Finder.

2 Static Information Flow Control

Information flow security [21] is concerned with the information transfer during program executions and its impact on confidentiality and integrity of information in information sources or information sinks of program executions. For the purpose of security with respect to timing channels we consider only confidentiality. The main question that information flow security aims to answer is *whether a given program is trustworthy enough to receive confidential data as input*, i.e., whether an attacker cannot infer information about the confidential input to the program by his observations about program executions.

The objective of research about static information flow control is to provide security properties that capture what secure information flow means, and to provide mechanisms that analyze given programs for whether they have secure information flow. The analysis is static and does not require program execution. Security properties for information flow are often formulated as *lack-of-dependency* properties, like the famous *noninterference* [15], which states that observations of an attacker are the same for all executions that have the same non-confidential inputs but possibly different confidential inputs.

In order to statically analyze programs, *security type systems* have been introduced [33] and widely adopted [26]. Security type systems have similarities with data type systems. A typical data type system [23] is supposed to ensure that data is used as intended during program execution: the data types have to match when the data is transferred and operations are checked to be applied only to the correct instances of data types. Similarly, a security type system shall ensure that information represented by the language constructs does not flow such that confidential information becomes public. First of all, this includes checking the transfer of data, for instance, when assigning the value of one variable to another. Second, this comprises checks to avoid *implicit* information flow, for instance, in the following exemplary code fragment. Consider

```
keypart = secretkey[i];
if (keypart==0) {
  output = 1;
} else {
  output = 0;
}
```

where the array secretkey initially holds the secret key of a cryptographic algorithm and output will be revealed to an attacker at some later point. The first assignment copies confidential data to keypart. The implicit flow occurs in the conditional: by observing whether output gets assigned 1 or 0 the attacker learns whether secretkey[i] contains 0. A security type system can detect such vulnerabilities by determining whether the condition of the branching depends on confidential information, and, in case it does, checking that in the branches no variables are assigned whose content might be revealed to an attacker later in the program.

3 Causes of Timing Channels

We conducted a careful study of the timing attacks reported in the literature which *do not* involve caches or any other micro-architectural peculiarities. The purpose of this study was to identify a common vulnerability which is exploited in this class of attacks.

The attack against IDEA [18] is based on a conditional in the implementation of the multiplication modulo $2^{16}+1$. For the attacks against modular exponentiation with Montgomery multiplication [13,27,11,6], the cause of the timing channel is the extra modular reduction step that is necessary if an intermediate result is bigger than the modulus, and which is a subtraction in one branch of a conditional. The attacks against RSA in OpenSSL [11,6] as additional cause for the timing channel have the choice of the multiplication-algorithm, namely, Karatsuba in the case of equally-sized multiplicands. The attack against modular multiplication by the Blakley's algorithm [9] is also caused by an extra modular reduction in the case of an intermediate value bigger than the modulus. The investigation of DES implementation in [17] reports the exploited timing channel to be caused by conditionals. The timing attacks against McEliece [32,29,31] exploit loops with a condition on confidential information. For the Kocher's attack [19] against fast exponentiation the causes are differences in the execution time of multiplication of integers. More details are not explored, however, since the relevant integers are usually much bigger than integers on processors, the multiplication would be implemented in some library, where the cause of the timing differences might be conditionals, for instance, extra Montgomery reductions.

When considering the timing channels that are exploited in the attacks against software implementations described with sufficient details, the causes are conditionals or loops, whose conditions depend on information about the attacked secrets. As mentioned in Section 2, determining whether the values of some expressions potentially depend on confidential information is one of the goals of information flow control. Thus, information flow control can detect the causes of all aforementioned attacks by determining whether the conditions of conditional and loop statements depend on the secret key.

4 Side Channel Finder (Version 1.0)

The Side Channel Finder was built with the idea in mind to apply the methods of static information flow control to the problem of timing channel detection. We believe that the tool might be helpful for the implementors of cryptographic algorithms in the Java programming language in checking whether their actual implementations open up timing channels. The programmer is expected to specify secret inputs for the algorithms as *high-classified* in an information flow policy. SCF 1.0 then checks whether the program code conforms to the information flow policy. Most important aspects of that are (i) to keep track where the high-classified data is moved to low-classified locations, and (ii) to verify that the execution would not branch on a value with the high security level.

4.1 Language Coverage

Since SCF 1.0 detects whether the control flow depends on information that is specified as confidential, the aspects of how information is propagated and how the control flow is data-dependent are relevant for the analysis. The static analysis of SCF 1.0 is executed on the syntactic structure of the analyzed programs. However, since secure information flow is a semantic property, the design of the analysis captures semantic aspects.

The Side Channel Finder takes into account the following semantic features regarding the propagation of information. It handles assignments to local variables, for instance, if the value of some expression *exp* is assigned to a local variable v (i.e., v = *exp*) and *exp* contains confidential information then SCF 1.0 treats v also as a container of confidential information. SCF 1.0 also respects assignments to fields of objects on the heap. For instance, let us consider an assignment v.f = *exp*. Firstly, it moves the value of *exp* into the field f. Secondly, this assignment also might reveal the information to which object the variable v is an alias, namely, the object of which the value of the field changed. Similarly, SCF 1.0 respects assignments to elements of arrays on the heap with the additional aspect that the index, which might be confidential, influences at which position the value changes. The tool also respects parameter passing, for instance, in a method call v.m(*exp*). Note that the passing can be by value, if the value of *exp* is a of primitive type like an integer, or by reference, if the value of *exp* is a reference to an object or an array. Thereby, SCF 1.0 does not only consider one method, but all methods that possibly could be the target of the call v.m(*exp*), taking into account inherited methods and polymorphism. If a method returns a value, i.e., its body contains a statement return *exp*, and the value of *exp* contains confidential information, then SCF 1.0 respects that calling this method and using the returned value means using confidential information.

The Side Channel Finder respects the following semantic features about the control flow. A conditional branching with a condition on a confidential value, that is a value that SCF 1.0 has determined to depend on confidential input, is considered a potential timing channel. For example consider

```
if (v==0) {
  // do something
} else {
  // do something different
}
```

where the content of v in confidential. Similarly, SCF 1.0 considers conditions on confidential values in for-loops and while-loops. A further cause of branching in the control-flow which the SCF 1.0 respects are polymorphic method calls. Consider again a method call v.m(*exp*) where now the reference in v is confidential and might point to objects of different classes, each of them having its own implementation of the method. The method which is actually executed depends on the class of the object that v points to.

This coverage of Java language features is sufficiently extensive to analyze existing implementations of cryptographic algorithms, as we show in Section 5.

4.2 Information Flow Policy

The main purpose of an information flow policy is to specify which input of an implementation of a cryptographic algorithm constitutes the secret to be protected, e.g., the secret key of an encryption or decryption algorithm. Moreover, we use policies to provide a guidance for the automatic analysis by specifying further program entities as holding confidential information if the contents of these entities potentially depend on secrets when executing the program.

Let us consider how an input for cryptographic algorithms is realized in Java implementations. In Java libraries, cryptographic algorithms are implemented in certain methods. Hence, particular parameters of such methods may be used to pass secret input to cryptographic algorithms and, therefore, must be specified as secrets. For instance, consider a decryption routine of some encryption scheme that receives the ciphertext to decrypt and the secret key in the form of arrays of bytes, and returns the decrypted result in the form of an array of bytes. This routine can be implemented in a method that has the signature byte[] decrypt(byte[] input, byte[] key). The input parameter key must not be learned by the attacker and hence needs to be specified as secret. Furthermore, some objects that are passed as parameters to such methods may have fields that hold secret input for cryptographic algorithms. Hence, these fields need to be specified as secrets too. Consider, for example, RSA where the decryption method can be realized by a method that has the signature byte[] decrypt(byte[] ciphertext, RSAPrivateKey key), and where the class RSAPrivateKey has a field for the public modulus and a second field with the private decryption exponent. Here, the second field needs to be specified as secret. Note, that a field is only considered not to be secret if the reference to the object of the field is also not secret. That is, in this example the parameter key is specified not to be secret in order to leave the public modulus actually public.

To provide guidance for the analysis, a policy contains one or more specifications for each method that is called and one specification for each field that is accessed. Each field may be specified to contain confidential information. Specifications of methods represent which of the parameters are used to pass confidential information and whether the value they return need to be kept secret. Multiple specifications for a method may be provided in order to make the analysis *context sensitive*, that is, SCF 1.0 can deal with different security levels for arguments at different calls to the same method.

The policy assigns security levels *low* or *high* to fields, method parameters, and method return values. Level *high* represents confidential information and level *low* represents non-confidential information. Information is only considered to be non-confidential if program entities necessary to access it all have level *low*. For instance, the content of a field with level *low* where the field is a part of an object where the parameter with the reference has level *high* is considered to be confidential. Letting level *high* represent confidential information means that information stored in program entities with level *high* must not influence the running time of the analyzed program. On the other hand, this also means it is safe to let information in such program entities depend on other information that is confidential.

The security levels of fields are specified class-wise in contrast to object-wise, hence SCF 1.0 can determine the levels statically without having to know the object at runtime. A whole object can be specified as confidential by specifying the information elements that contain a reference to it as confidential. The same is true for arrays, where also the content of the array is considered confidential as long as the references are confidential. SCF 1.0 does not require to specify the security levels of local variables of methods because SCF 1.0 infers them automatically from the levels that are specified in the policy.

SCF 1.0 reads a policy in the form of an XML document. The graphical interface of the SCF Eclipse plugin provides, as depicted in Figure 1, convenient means for editing the XML code of the policy. The specifications for each class are organized in packages similarly to actual classes of Java. Concerning the location of policies, an alternative to storing them in a separate file could have been to integrate them into the source code, for instance, as annotations. The advantage of our approach is that we do not need to change the source code and can process it as it is.

4.3 Security Type System

In order to check security of a given program against a given policy with respect to timing channels, the Side Channel Finder implements a carefully crafted security type system which we describe below. The reader can find the complete formal definition (3 pages) of this type system in a technical report [22].

The security type system checks method declarations against security signatures for methods and fields. The method security signatures and the field security signatures represent the information that is provided by the security policy: they map method parameter names, return values, and field names to respective security levels. That a method declaration conforms to given method and field security signatures is expressed by *type judgments* which are formal assertions about the (secure) typing of the program. The type check of a specified method declaration is considered successful if the type judgment for this method declaration can be derived by *type rules* which are implications between judgments.

Derivable judgments of our type system are intended to ensure two aspects of statements and expressions. Firstly, for assignment statements they guarantee that the security levels of source and destination memory locations are such that no high data will be moved to low-typed locations during program execution. Secondly, they ensure that execution would not branch on a value with the high security level. In order to provide these main results, derivable judgments determine two further aspects. Firstly, they determine the security levels of local variables that are relevant for the statements and expressions in their respective context. That is, the security type system is *flow sensitive* for local variables. Secondly, for expressions they determine the security levels of the expression evaluation.

The type inference algorithm implemented in SCF 1.0 is rather standard. The source code of the given program is converted into an abstract syntax tree representation by the *javaparser* [14] in the version 1.0.8. The methods and their respective signatures from the accompanying XML file with the information flow policy are coupled. The type rules are defined inductively over the program structure, i.e., a type check of a statement depends on a type check of its sub-statements. The algorithm, therefore, processes the tree structure of the program and searches for an applicable rule. If no such rule is found the program is not typable within the type system. By design of our type system that means a potential timing channel is found. If the given program code is completely processed, i.e., typed, the analysis succeeds meaning that no timing channels are found.

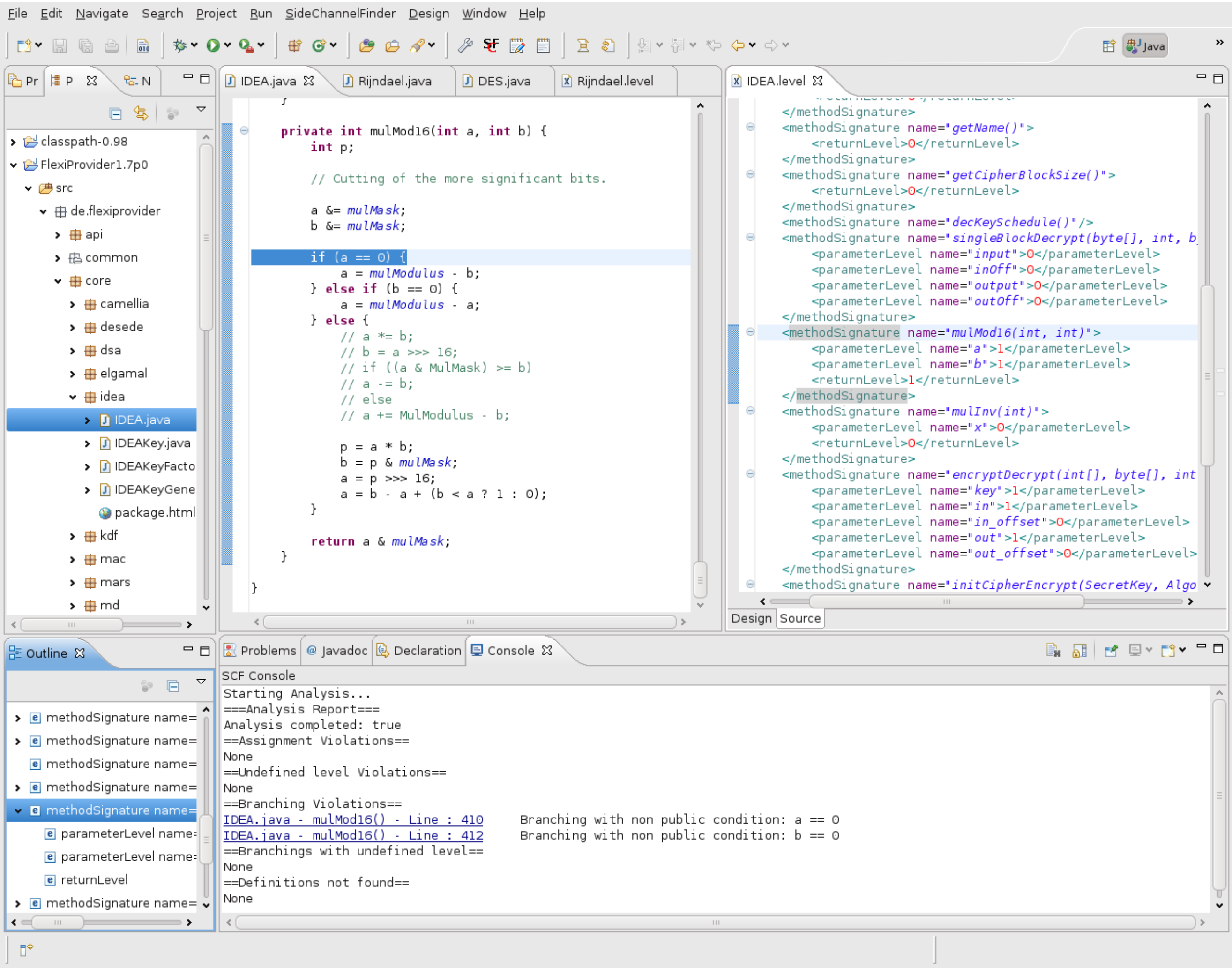

Figure 1 GUI of the SCF Eclipse plugin reporting potential timing channels in the FlexiProvider's IDEA implementation: source code with the highlighted line of a potential timing channel (middle), interface for editing security policies (right), and list of detected potential timing channels (bottom)

5 Case Study: Detecting and Exploiting a Timing Channel in an IDEA Implementation

In this Section we demonstrate how SCF 1.0 can be successfully applied to an existing implementation of a cryptographic algorithm. We consider the encryption scheme IDEA as implemented in the open-source library FlexiProvider (version 1.7p0) [3]. An excerpt of the relevant source code is depicted in Figure 2.

5.1 Analysis by SCF 1.0

The goal of the analysis is to protect the secret key at encryption. Encryption is implemented in the method singleBlockEncrypt(byte[] input, int inOff, byte[] output, int outOff) of the class IDEA in the package de.flexiprovider.core.idea. Each round of the IDEA encryption uses a different selection of bits of the secret key. These round keys are scheduled before the actual encryption. The scheduled secret key for the encryption is stored in the field encr of type int[].

As the first step, given the target of the analysis and the corresponding source files, SCF 1.0 automatically initialize three files: (i) a policy file IDEA.level which contains security levels for all fields an method parameters set to the default *low* (i.e., non-confidential), (ii) a file containing the paths to the source files that contain relevant code, IDEA.program, and (iii) a file to configure the whole analysis, IDEA.analysis. The last file contains the method to be analyzed, and paths to the relevant files.

The next step is to specify the security levels. In order to express that the encryption key is to be protected, we set the security level of the field encr (the scheduled encryption key) to the confidential level in

```
 1 private static final int mulModulus = 0x10001;
 2 private static final int mulMask = 0xffff;
 3 [...]
 4 protected void singleBlockEncrypt(byte[] input, int inOff,
 5    byte[] output, int outOff) {
 6      encryptDecrypt(encr, input, inOff, output, outOff);
 7 }
 8 [...]
 9 private void encryptDecrypt(int[] key, byte[] in, int in_offset,
10      byte[] out, int out_offset) {
11      [...]
12      int x0 = in[in_offset++] << 8;
13      x0 |= in[in_offset++] & 0xff;
14      [...]
15      for (int i = 0; i < rounds; ++i) {
16           x0 = mulMod16(x0, key[k++]);
17           x1 += key[k++];
18           x2 += key[k++];
19           x3 = mulMod16(x3, key[k++]);
20           [...]
21      }
22      [...]
23      out[out_offset++] = (byte) (x0 >>> 8);
24      out[out_offset++] = (byte) x0;
25      [...]
26 }
27 [...]
28 private int mulMod16(int a, int b) {
29      int p;
30      a &= mulMask;
31      b &= mulMask;
32
33      if (a == 0) {
34           a = mulModulus − b;
35      } else if (b == 0) {
36           a = mulModulus − a;
37      } else {
38           p = a * b;
39           b = p & mulMask;
40           a = p >>> 16;
41           a = b − a + (b < a ? 1 : 0);
42      }
43      return a & mulMask;
44 }
```

Figure 2 Excerpt from the IDEA source code in FlexiProvider (comments and empty lines are omitted)

```
<?xml version="1.0" encoding="UTF−8" standalone="no"?>
<informationLevelModel>
 <package name="de.flexiprovider.core.idea">
  <classSignature name="IDEA">
   <fieldLevel name="encr">1</fieldLevel>
   <fieldLevel name="mulModulus">0</fieldLevel>
   <fieldLevel name="decr">1</fieldLevel>
   <fieldLevel name="blockSize">0</fieldLevel>
   <fieldLevel name="keySize">0</fieldLevel>
   [...]
   <methodSignature name="singleBlockEncrypt(byte[],int,byte[],int)">
    <parameterLevel name="input">1</parameterLevel>
    <parameterLevel name="inOff">0</parameterLevel>
    <parameterLevel name="output">1</parameterLevel>
    <parameterLevel name="outOff">0</parameterLevel>
   </methodSignature>
   [...]
   <methodSignature name="mulMod16(int,int)">
    <parameterLevel name="a">1</parameterLevel>
    <parameterLevel name="b">1</parameterLevel>
    <returnLevel>1</returnLevel>
   </methodSignature>
   [...]
   <methodSignature name="encryptDecrypt(int[],byte[],int,byte[],int)">
    <parameterLevel name="key">1</parameterLevel>
    <parameterLevel name="in">1</parameterLevel>
    <parameterLevel name="in_offset">0</parameterLevel>
    <parameterLevel name="out">1</parameterLevel>
    <parameterLevel name="out_offset">0</parameterLevel>
   </methodSignature>
   [...]
  </classSignature>
 </package>
</informationLevelModel>
```

Figure 3 Excerpt of the policy for the IDEA encryption method: 1 represents confidential parameters and fields, whereas 0 corresponds to public parameters and fields

the SCF's interface for editing the security policy (right side of Figure 1). The performed modifications are stored then in the file IDEA.level. Running the analysis with this policy reveals that the method parameters at several points in the program are instantiated with confidential data or are used to store confidential data. Hence, we also set these parameters to the confidential level, that is the parameters input and output of singleBlockEncrypt, the parameters key, in, and out of encryptDecrypt, the parameters a and b of mulMod16, and the return value of mulMod16. Figure 3 shows an excerpt of the resulting policy. Note, that in the depicted policy 1 corresponds to the security level *high*, whereas 0 to the security level *low*.

Finally, we actually run the analysis by clicking the corresponding button of the SCF plugin. The report (see the console at the bottom of Figure 1) contains

```
IDEA.java − mulMod16() − Line : 410
Branching with non public condition: a == 0
```

Inspecting the findings in the source of the method mulMod16 reveals that there actually are certain values of the parameters that result in a special treatment, which is realized by branching on their values (Figure 2, lines 33–37).

This method implements multiplication modulo $2^{16}+1$. Only the lowest 16 bits of the variables are used. The value where all bits are 0 is interpreted as 2^{16}. This modular multiplication is applied several times within the encryption and decryption of IDEA.

The finding corresponds to a known [18] timing channel vulnerability that implementations of IDEA are prone to. In the following we show that the finding in the FlexiProvider implementation actually constitutes a timing channel.

5.2 Experimental Evidence of the Timing Channel

The objective of our experiment was to learn whether measurements of the running time of the IDEA-implementation in FlexiProvider actually can be used to obtain parts of the secret key. The experiment shows that this is the case, i.e., that the potential timing channel found by SCF 1.0 is actually a timing channel.

No.	key	key bits is positions 70–85	total avgerage time	maximum average time of clusters	16 bits (A) with maximum cluster time	estimated key bits $(2^{16}+1)-A$	key bits match
1	2	3	4	5	6	7	8
1	0x51305a2b8993beb 703a88d022f78b74c	0xea23	2960	3002	0x15de	0xea23	✓
2	0x00e148d92641ecf 99d428836b7bf0150	0x50a2	2934	2991	0xaf5f	0x50a2	✓
3	0x7633842df1c4f6a 5cc215537514141a4	0x0855	2893	2917	0x8894	0x776d	X
4	0x195bfbd477a8bf1 2a61f4ea42ba85b41	0x87d3	2888	2929	0x782e	0x87d3	✓
5	0x683b4ca4e842d4e b5180c34cef8adbc8	0x6030	2907	2992	0x9fd1	0x6030	✓
6	0x10530d320f7836b b6ed1a85d4fe2ef78	0xb46a	2900	2943	0x4b97	0xb46a	✓
7	0x3a71d80a3ae3d1f edb023239074ef509	0xc08c	2923	2965	0x3f75	0xc08c	✓
8	0x572e11ad1bc8a1f 8c787f13f2cd97aaf	0xe1fc	2883	2978	0x1e05	0xe1fc	✓
9	0x4512cac44b60db9 6c24fbd0814fc94f2	0x93ef	2908	2958	0x6c12	0x93ef	✓
10	0x1fab5773e8025ab 2299e192a5805b9bf	0x6786	2944	2966	0x987b	0x6786	✓

Table 1 Results of Experiments

Attack from the Literature Based on the assumption that the running time of the branches differs, in [18] two attacks are suggested against IDEA-implementations. We consider one aspect of the first attack, on that we build our experiment.

In IDEA, the lowest 16 bits of the ciphertext are the output of a multiplication modulo $2^{16}+1$, where the multiplicands are an intermediate value and 16 bits (bits at position 70–85) of the secret key. Hence, if one can determine for which lowest 16 bits of the ciphertext the first multiplicand is zero, then one can calculate the 16 bits of the secret key by subtracting the integer-interpretation of these 16 bits of the ciphertext from $2^{16}+1$. In implementations like the one in FlexiProvider the case where the first multiplicand is zero exactly corresponds to one branch in the control flow. The attack tries to determine cases where this branch is taken by timing measurements.

Approach to Evaluation of the Timing Channel Similarly to the attack, we measure the running time of encryption for many ciphertexts and a fixed key. We identify the lowest 16 bits of ciphertext that result from calculations with on average extreme running time. From these bits we estimate 16 bits of the ciphertext by a simple calculation. Then, we compare these estimated bits with the bits 70–85 of the actual key.

Setup We run the experiment on a standard machine, an IBM Thinkpad T60 2007-CTO with the processor Intel Core2 T7200 and 3GB RAM. The machine is installed with Ubuntu 10.10 i386. OpenJDK is installed in the version 6 in form of the standard-packages of Ubuntu (6b20-1.9.1-1ubuntu3). We run the experiment after a standard boot, logged in the console, and with the X-server stopped. The only parameter that we pass to the Java virtual machine is the `classpath`, i.e., we run the server-HotSpot virtual machine. Hence, we have a standard environment where the only tweak is not running the X-server.

The experiment runs as follows. First we have a measurements phase. We take a pseudo-random key (source `/dev/urandom`). We run a Java program that applies the encryption method on 16777215 pseudo-random 8-byte messages. Each encryption is conducted 64 times, whose execution time the program measures with `System.NanoTime()`. The program divides the time by 64, and stores the result together with the resulting ciphertext. After that, we evaluate the measurements. We cluster the pairs of a ciphertext and a running time according to the lowest 16 bits of the ciphertext. Then we calculate for each cluster the average running time of the cluster. We identify the cluster with the highest average running time. We subtract the 16-bit value (positive integer interpretation) of this cluster from $2^{16}+1$. The 16-bit representation of the result is our estimation of the key bits, which we compare to the bits 70–85 of the actual key.

We run the experiment for 10 different pseudo-random keys.

Results In Table 1 for each key (column 2) we present the total average running time (column 4) of the encryption, the highest average running time within the 16-bit-clusters (column 5), and the 16-bits of the cluster with the highest average time (column 6). We refer to this 16 bits by A. In order to compare we also present the attacked 16 bits (positions 70–85 of the secret key, column 3) and the estimated key-bits (column 7).

Comparing the third column (the attacked key bits) and the seventh column (the estimated key bits) shows that in nine out of the ten cases the key-bits estimated from the timing measurements are correct.

Conclusion In our setup the bits of the secret key can be reliably determined by taking timing measurements and simple calculations. Thus, the finding of the Side Channel Finder actually constitutes a timing channel.

6 Summary and Future Work

We have presented the Side Channel Finder, a tool that can statically detect timing channel vulnerabilities in implementations of cryptographic algorithms in the Java programming language. SCF 1.0 covers a non-trivial subset of Java including objects, arrays, and methods. These concepts are commonly used in Java implementations of cryptographic algorithms, for instance, in the examples we considered (see Section 5 and [22]). We described the policy language of the Side Channel Finder which identifies the secrets that must not leak through timing channels. The expressiveness of the policy language reflects the programming language that is covered by SCF 1.0, i.e., it supports security signatures for fields and methods. SCF 1.0 helps the designer of such policies by automatic generation of policies with default specifications. The Side Channel Finder is designed to analyze programs for branching of the control flow at which the branch taken depends on confidential input. The causes of such a branching could be conditional statements, loops, or polymorphic method calls. We carefully crafted a security type system which automatically checks whether this is the case for a given Java program. The case study discussed in the paper showed that SCF 1.0 is sufficiently mature to identify timing channels which actually could be exploited.

Currently we see the following directions for improvements in the Side Channel Finder and studies of its applications.

- First of all, we are currently implementing the missing coverage of a number of Java constructs. This includes, for instance, constructors which are quite similar to methods, or more complex scopes of field-access and method-call expressions which are currently limited to variable, field, or type names.
- Next, we will apply the Side Channel Finder to implementations of further algorithms. We will turn to the well-studied asymmetric scheme RSA, which essentially means analyzing the implementation of fast exponentiation. Further, we will consider implementations of post-quantum algorithms, for instance McEliece, which are a distinguishing feature of the library FlexiProvider.
- Further, we plan to implement automatic program transformations for elimination of discovered timing channels. Here we have a choice between a number of techniques, namely, cross-copying [7], unification [20], or conditional assignment [24].
- Finally, we are working in the direction of making the analysis semantically justified. This will be based on a semantic notion of timing channel security where the branching on the secret data will be allowed provided the branches take the same amount of time. The check, whether a program respects this notion will be implemented by means of a type system in the Side Channel Finder. For that, we have to decide how fine-grained the timing model has to be with possibilities ranging from counting the number of executed statements — through considering single expression evaluation steps — to careful estimation of the timing behavior of the compiled code instructions on the underlying architectures.

Acknowledgments We thank Richard Gay for helpful comments on the draft of this paper and the anonymous reviewers for their suggestions. This work was supported by CASED (www.cased.de).

References

1. Flow Caml. `http://www.normalesup.org/~simonet/soft/flowcaml/` (2003)
2. GNU Classpath. `http://www.gnu.org/software/classpath/` (2009)
3. FlexiProvider – A Toolkit for the Java Cryptography Architecture (JCA/JCE). `http://www.flexiprovider.de` (2010)
4. JIF: Java + information flow. `http://www.cs.cornell.edu/jif/` (2010)
5. Eclipse – The Eclipse Foundation open source community website. `http://www.eclipse.org/` (2011)
6. Aciiçmez, O., Schindler, W., Ç. K. Koç: Improving Brumley and Boneh timing attack on unprotected SSL implementations. In: Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS), pp. 139–146. ACM, Alexandria, VA, USA (2005)

7. Agat, J.: Transforming out Timing Leaks. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 40–53. ACM Press, Boston, MA, USA (2000)
8. Agat, J.: Type Based Techniques for Covert Channel Elimination and Register Allocation. Ph.D. thesis, Chalmers University of Technology (2001)
9. Bakhshi, B., Sadeghiyan, B.: A Timing Attack on Blakley's Modular Multiplication Algorithm, and Applications to DSA. In: Proceedings of the 5th International Conference on Applied Cryptography and Network Security (ACNS), LNCS 4521, pp. 129–140. Springer (2007)
10. Barthe, G., Rezk, T., Warnier, M.: Preventing Timing Leaks Through Transactional Branching Instructions. In: Proceedings of the 3rd Workshop on Quantitative Aspects of Programming Languages (QAPL), ENTCS 153, pp. 33–55. Elsevier, Edinburgh, UK (2006)
11. Brumley, D., Boneh, D.: Remote timing attacks are practical. Computer Networks **48**(5), 701–716 (2005)
12. Chapman, R., Hilton, A.: Enforcing Security and Safety Models with an Information Flow Analysis Tool. In: Proceedings of the 2004 annual ACM SIGAda international conference on Ada, SIGAda '04, pp. 39–46. ACM, New York, NY, USA (2004)
13. Dhem, J.F., Koeune, F., Leroux, P.A., Mestré, P., Quisquater, J.J., Willems, J.L.: A Practical Implementation of the Timing Attack. In: Proceedings of the 3rd International Conference on Smart Card. Research and Applications (CARDIS 98), LNCS 1820, pp. 167–182. Springer (1998)
14. Gesser, J.V.: javaparser. `http://code.google.com/p/javaparser/` (2010)
15. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 11–20. IEEE Computer Society, Oakland, CA, USA (1982)
16. Hammer, C.: Experiences with PDG-based IFC. In: Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS 2010), LNCS 5965, pp. 44–60. Springer (2010)
17. Hevia, A., Kiwi, M.: Strength of Two Data Encryption Standard Implementations Under Timing Attacks. In: Proceedings of the Theoretical Informatics Third Latin American Symposium (LATIN), LNCS 1380, pp. 192–205. Springer, Campinas, Brazil (1998)
18. Kelsey, J., Schneier, B., Wagner, D., Hall, C.: Side Channel Cryptanalysis of Product Ciphers. Journal of Computer Security **8**(2,3), 141–158 (2000)
19. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO, LNCS 1109, pp. 104–113. Springer, Santa Barbara, CA, USA (1996)
20. Köpf, B., Mantel, H.: Transformational Typing and Unification for Automatically Correcting Insecure Programs. International Journal of Information Security (IJIS) **6**(2–3), 107–131 (2007)
21. Lampson, B.W.: A note on the confinement problem. Communications of the ACM **16**, 613–615 (1973)
22. Lux, A., Mantel, H., Perner, M., Starostin, A.: Side Channel Finder (Version 1.0). Tech. Rep. TUD-CS-2010-0155, TU Darmstadt (2010)
23. Mitchell, J.C.: Handbook of theoretical computer science (vol. b). chap. Type systems for programming languages, pp. 365–458. MIT Press, Cambridge, MA, USA (1990)
24. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In: Proceedings of the 8th Annual International Conference on Information Security and Cryptology (ICISC), pp. 156–168. Springer, Seoul, Korea (2005)
25. Myers, A.C.: JFlow: Practical Mostly-Static Information Flow Control. In: Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL), pp. 228–241. ACM, San Antonio, TX, USA (1999)
26. Sabelfeld, A., Myers, A.C.: Language-based Information-Flow Security. IEEE Journal on Selected Areas in Communication **21**(1), 5–19 (2003)
27. Schindler, W.: A Timing Attack against RSA with the Chinese Remainder Theorem. In: Proceedings of Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS 1965, pp. 109–124. Springer, Worcester, MA, USA (2000)
28. Schindler, W.: On the Optimization of Side-Channel Attacks by Advanced Stochastic Methods. In: Proceedings of the 8th International Workshop on Theory and Practice in Public Key Cryptography (PKC), LNCS 3386, pp. 85–103. Springer, Les Diablerets, Switzerland (2005)
29. Shoufan, A., Strenzke, F., Molter, H.G., Stöttinger, M.: A Timing Attack against Patterson Algorithm in the McEliece PKC. In: Proceedings of the 12th International Conference on Information, Security and Cryptology (ICISC), LNCS 5984, pp. 161–175. Springer, Seoul, Korea (2009). Revised Selected Papers
30. Standaert, F.X., Malkin, T., Yung, M.: A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In: Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), LNCS 5479, pp. 443–461. Springer (2009)
31. Strenzke, F.: A Timing Attack against the Secret Permutation in the McEliece PKC. In: Proceedings of the Third International Workshop on Post-Quantum Cryptography (PQCrypto), LNCS 6061, pp. 95–107. Springer, Darmstadt, Germany (2010)
32. Strenzke, F., Tews, E., Molter, H.G., Overbeck, R., Shoufan, A.: Side Channels in the McEliece PKC. In: Proceedings of the 2nd International Workshop on Post-Quantum Cryptography (PQCrypto), LNCS 5299, pp. 216–229. Springer, Cincinnati, OH, USA (2008)
33. Volpano, D., Smith, G., Irvine, C.: A Sound Type System for Secure Flow Analysis. Journal of Computer Security **4(3)**, 1–21 (1996)