Side Channel Finder (Version 1.0)

Technical Report TUD-CS-2010-0155

October 2010

Alexander Lux,
Heiko Mantel,
Matthias Perner,
Artem Starostin

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Modeling and Analysis
of Information Systems

CASED

Contents

1 Introduction

The security of an information system often depends on a cryptographic mechanism, for instance if the mechanism establishes confidential communication over untrusted networks by encryption. Hence, the security of cryptographic algorithms is important. However, considering security on the level of algorithms is not enough because additional aspects of reality become relevant for security when implementing cryptographic algorithms. Side channel attacks exploit the fact that an attacker can observe behavior of a cryptographic mechanism that is not modeled by the cryptographic algorithm. This enables the attacker to infer secrets from the observed behavior. Hence, when developing a cryptographic mechanism it is desirable to detect whether its implementation potentially opens up side channels.

One possibility to launch a side channel attack is to exploit the variance in the running time of a software implementation of a cryptographic algorithm. In this work we focus on this kind of side channels. It has been demonstrated that such a vulnerability even can be exploited remotely over a network [BB05].

We present the *Side Channel Finder* in the version 1.0 (short SCF 1.0), a tool for the detection of potential timing channels in Java implementations of cryptographic algorithms. The purpose of the tool is to support a programmer of an implementation of a cryptographic algorithm in assessing his code for that it is not susceptible to timing channel attacks. SCF 1.0 lets the programmer specify which input of an implemented algorithm constitutes a secret that must not be leaked, especially not through timing channels. SCF 1.0 then analyzes the given program code by checking whether the control flow potentially depends on this secret input, and, hence, the time the code consumes while executing. Version 1.0 of the tool can analyze several actual implementations of cryptographic algorithms in existing cryptographic libraries.

Structure of the Report Section 2 presents the programming language for implementations which can be analyzed by SCF 1.0. Section 3 describes a policy language for specifying which are the secrets to be protected. Section 4 elaborates on the syntactic security model that guided the design of SCF 1.0. Section 5 demonstrates that SCF 1.0 can analyze real world implementations of cryptographic algorithms. We discuss the related work in Section 6 and summarize our results in Section 7.

Notation In this paper we only use total functions $f : A \to B$, no partial functions. In order to represent partial functions we define the set $B_\perp$ for each set B as $B_\perp = \{\lfloor b \rfloor | b \in B\} \cup \perp$ and use such a set as the co-domain of a function: $f : A \to B_\perp$. The meaning of $f(a_1) = \perp$ is that the partial function that is represented by f is not defined for a_1 and the meaning of $f(a_2) = \lfloor b \rfloor$ is that the partial function represented by f maps a_2 to b.

We use the following notation for function updates. Given a function $f : X \to Y$ and elements $x \in X$, $y \in Y$, the function $f\{x \mapsto y\}$ is the function that coincides with f on all values $x' \in X$ such that $x' \neq x$ and that maps x to y.

2 Target Language

The purpose of SCF 1.0 is to detect potential timing channel vulnerabilities in given Java implementations of cryptographic algorithms. Therefore SCF 1.0 has to cover a subset of Java that such programs typically use. Below we present Java_{SCF}, the subset of Java covered by SCF 1.0.

2.1 Syntax

Structure We assume given a set of names *Names*. The structure of programs into packages, classes, fields, and methods is as follows. A Java_{SCF}-program P consists of a finite set of *class declarations* $\mathit{ClassDecl}_P$. A class declaration is a quadruple that consists of a *class name* from *Names* that identifies the class, a second class name from *Names*, which determines a *direct superclass*, a finite set of *field declarations*, and a finite set of *method declarations*. The class name is unique within P. Hence, a class declaration uniquely determines a superclass, a set of field declarations, and a set of method declarations for a class name c.

Types The declarations of methods and fields employ *types*. The set of types *Types* is the smallest set such that $\mathsf{int}, \mathsf{boolean}, \mathsf{long} \in \mathit{Types}$, $\mathit{Names} \subseteq \mathit{Types}$ (for classes as types), and $ty[] \in \mathit{Types}$ for each $ty \in \mathit{Types}$.

Fields and Methods Fields in Java_{SCF} have a type. A *field declaration* is a pair of a *field name* from *Names* and a *field type* from *Types*. Methods in Java_{SCF} have a type signature of their input and output, and a body containing the actual program code. A *method declaration* is a quintuple which comprises the following components: (a) a *method name* from the set *Names*, (b) a *return type* from $\mathit{Types} \cup \{\mathsf{void}\}$, (c) a list of *parameter types* from *Types*, (d) a list of *parameter names* from *Names*, and (e) a *method body* which constitutes the actual program code. The lists of parameter types and the parameter names have equal lengths. The i-th element of the parameter types list contains the type of the parameter whose name is given at position i of the parameter names list. The set of all method declarations is *MethodDecl*.

Program Code Program code consists of two main kinds of elements, namely, statements and expressions. Expressions are elements that are intended to be evaluated (i.e., to be ascertain a value) whereas statements are not. However, expressions can be used as statements.

The grammar of the considered language is presented in Figure 1. For each symbol *sym* that appears in the grammar we denote by Lang_{sym} a sub-language defined by the grammar for that symbol. We assume given a set of *constant expressions Consts*, a set of *unary operators UnOps*, and a set of *binary operators BinOps*. A method body is a statement from $\mathit{Lang}_{\text{Statement}}$.

Realization in SCF 1.0 In order to generate an AST-representation from actual files containing Java source code, SCF 1.0 employs the *javaparser* [Ges10] in the version 1.0.8 on a set of files specified by the user. Syntactic constructions beyond the target language of SCF 1.0 are reported when running SCF 1.0. Examples of those include exception throw statements and a scope for access to a field or a method that is not a name.

Statement	::=	Expression;
	|	if (Expression) { [Statement] } [else { [Statement] }]
	|	for ([ListOfExpressions]; Expression; [ListOfExpressions]) { [Statement] }
	|	while (Expression) { [Statement] }
	|	return; | return Expression;
	|	BlockStmt
Expression	::=	(Expression) ? Expression : Expression
	|	VariableDeclarationExpression
	|	NameExpr = Expression
	|	*UNOP* Expression
	|	Expression *BINOP* Expression
	|	(Expression)
	|	(*TYPENAME*) Expression
	|	Expression instanceof *TYPENAME*
	|	NameExpr
	|	*CONSTANTEXPR*
	|	*METHODNAME*([ArgumentList])
	|	Scope.*METHODNAME*([ArgumentList])
ListOfExpressions	::=	Expression | Expression, ListOfExpressions
BlockStmt	::=	Statement | Statement BlockStatement
VariableDeclarationExpr	::=	*TYPENAME* ListOfVariableDeclarator
ListOfVariableDeclarator	::=	VariableDeclarator
	|	VariableDeclarator, ListOfVariableDeclarator
VariableDeclarator	::=	*VARIABLENAME* [= Expression]
NameExpr	::=	*VARIABLENAME*
	|	*VARIABLENAME*[Expression]
	|	*FIELDNAME*
	|	*FIELDNAME*[Expression]
	|	Scope.*FIELDNAME*
	|	Scope.*FIELDNAME*[Expression]
ArgumentList	::=	*TYPENAME VARIABLENAME*
	|	*TYPENAME VARIABLENAME*, ArgumentList
Scope	::=	*VARIABLENAME* | *FIELDNAME* | *TYPENAME*

where *VARIABLENAME*, *FIELDNAME*, *METHODNAME* $\in$ *Names*, *TYPENAME* $\in$ *Types*, *UNOP* $\in$ *UnOps*, *BINOP* $\in$ *BinOps*, and *CONSTANTEXPR* $\in$ *Consts*.

Figure 1: Grammar for Program Code in the Target Language of SCF 1.0

2.2 Semantics

SCF 1.0 detects where the control flow depends on information that is specified as confidential. Hence, how information is propagated and how control flow is data-dependent are the relevant aspects of program semantics.

SCF 1.0 respects the following semantic features regarding the propagation of information. SCF 1.0 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. SCF 1.0 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 an instruction 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.

SCF 1.0 respects the following semantic features about the control flow. A conditional branching with a condition on a confidential value is considered a potential timing channel, for instance if (v==0) then {*do something*} else {*do something different*}. Similarly, SCF 1.0 considers conditions on a confidential values in for-loops and while-loops. A further cause of branching in the control-flow that the SCF 1.0 respects are polymorphic method calls. Let us 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 that actually is executed depends on the class of the object that v points to.

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

3 Security Policy

The core purpose of a 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 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 input for cryptographic algorithms is realized in Java implementations. In Java libraries, cryptographic algorithms are implemented in certain methods. Hence, certain 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 of the analysis, a policy for the analysis contains one or more specification 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 needs to be kept secret. Multiple specification for a method may be provided in order to make the analysis *context sensitive*, that is the SCF 1.0 can deal with different security levels for arguments at different calls to the same method.

3.1 Policy Language

The policy assigns security levels 0 or 1 to fields, method parameters, and method return values. The level-value 1 represents confidential information and the level-value 0 represents non-confidential information. Information is only considered not to be confidential if program entities necessary to access it all have level 0. For instance, the content of a field with level 0 where the field is a part of an object where the parameter with the reference has level 1 is considered to be confidential. Letting level 1 represent confidential information means that information stored in program entities with the level 1 must not influence the running time of the 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 level 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 form of an XML-document. Figure 2 presents the grammar for policies. The specifications for each class are organized in packages (the XML-elements package) similarly to actual classes of Java. Within these package elements the security signatures (specifications) for the classes are listed, the XML-elements classSignature. Such a security signature for a class specifies exactly one level for each of its fields and at least one signature

LevelModel ::= <informationLevelModel>Packages</informationLevelModel>

Packages ::= <package name="[Packagename]">Types</package>
| <package name="[Packagename]">Types</package>Packages

Types ::= <classSignature name="*NAME*">Members</classSignature>
| <classSignature name="*NAME*">Members</classSignature>Types

Members ::= <fieldLevel name="*NAME*">*LEVEL*</fieldLevel>
| <fieldLevel name="*NAME*">*LEVEL*</fieldLevel>Members
| <methodSignature name="*NAME*([Parameters])">
Method<methodSignature>
| <methodSignature name="*NAME*([Parameters])">
Method</methodSignature>Members

Method ::= Params
| Params<returnLevel>*LEVEL*</returnLevel>

Params ::= <parameterLevel name="*NAME*">*LEVEL*</parameterLevel>
| <parameterLevel name="*NAME*">*LEVEL*</parameterLevel>Params

Parameters ::= *PARAMETERTYPE*
| *PARAMETERTYPE*,Parameters

Packagename ::= *NAME*
| *NAME*.Packagename

where $NAME \in Names$, $PARAMETERTYPE \in Types$, $LEVEL \in \{0, 1\}$

Figure 2: Grammar for Policy Definitions

for each of its method. That is each XML-element classSignature contains one level for each of the fields of the respective class, where the fields are identified by the name-attributes of the XML-elements fieldLevel. Each XML-element classSignature also contains at least one signature for each method of the respective class. The respective method is identified by the method name and the parameter types in the name-attribute of the XML-element methodSignature. A method signature needs to provide levels for its parameters which firstly determine with which levels for arguments the method may be called, and, secondly, which levels are assumed for the parameters when executing the method. A method signature also needs to provide a level for the return value if the method is not void. A method signature lists levels for the parameters by the XML-elements parameterLevel, and, optionally, for the return level with the XML-element returnLevel.

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.

3.2 Designing a Policy

Policies for SCF 1.0 do not have to be written from the scratch. SCF 1.0 can initialize policies given the respective source files. It generates complete signatures for all classes in these files. All levels initially are set to 0. The next step is to specify which parameters of the entry method for the cryptographic algorithm and which fields constitute secret input. Decisions for that need to be done by the designer of policies based on the meaning of the input by the actual algorithm.

SCF 1.0 also helps to specify further program entities to hold confidential information in order to guide the analysis. Running the analysis can report two kinds of violations. The first one is the branching of the control flow in which the branch to be taken depends on confidential information. These are the potential timing channel vulnerabilities that SCF 1.0 is built to find. The second kind are violations where the levels of entities between that information is moved do not match, for instance when a secret parameter is read and stored into a non-confidential field. In such a case we know that the entity into which information is moved potentially holds confidential information. Hence it is reasonable to raise its level to 1 and repeat the analysis.

On the whole, although the policy represents more information, the knowledge that a designer of the policy needs to put into the process of producing a policy only is which entities represent secret input of the respective cryptographic algorithm.

4 Security Type System

In this section we describe how the SCF 1.0 checks security of a given program against a given policy with respect to timing channels.

Branchings of the control flow, like conditionals, loops, or polymorphic method calls, can lead to observable variations of the program running time if the branches have different running time. To detect such potential causes of timing channels SCF 1.0 analyses programs for branching of the control flow at which the branch taken depends on confidential input. The check is based on the idea of security type systems for information flow security.

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 from Section 3. That a method declaration conforms to given method and field security signatures is expressed by *type judgments* which we define below in this section. For the security type system signatures are modeled as follows. The set of security levels is $L = \{high, low\}$ where *high* corresponds to 1 in the policy and *low* to 0. A *local signature* is a function $ls : Names \to L_\perp$ that represents levels of the method parameters and local variables for that the *ls* is defined. A *method signature* reflect a methodSignature-element of a given policy and is a quadruple $(c, m, ls, l_{\text{ret}})$, where the class name c and the method name m identify the method declaration, the local signature *ls* assigns security levels to the parameters, and the level l_{ret} is the level of the value returned by the methods with this signature. The local assignment of a method signature is only defined for parameter names because security levels of variables are intended to be inferred by type inference. We write *MS* for a list of method signatures. *Field signatures* reflects the fieldLevel-elements of a given policy where the class name of the field is determined by the surrounding classSignature-element. Field signatures are modeled by a function $fs : (Names \times Names) \to L_\perp$, where the first parameter represents a class name and the second a field name.

Properties to Check The type check of a specified method declaration is considered successful if the type judgment for this method declaration can be derived by the type rules. Derivable judgments of the type system are intended to ensure two aspects of statements and expressions. Firstly, for assignment statements they ensure 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.

Security Type Judgments and Rules In order to determine these results it is required to have (a) program code, (b) a class within that the respective expression or statement is checked (identified by a class name), (c) a list of method security signatures, (d) a field security signature

(security levels of fields), and (e) a security level against that returned values need to be checked.

We introduce three judgments: one for expressions, one for statements, and one for method declarations. The judgment for expressions is:

$$P, c, MS, fs \vdash exp : ls \rightarrow ls', l$$

Firstly, the judgment means that within the program P and within the class identified by c the value of the expression exp does only depend on information lower or equal than l, given the signatures MS, fs, and ls. Consider for instance the type rule for deriving the judgment for binary operations:

$$\text{[BinopExp]}\frac{\begin{array}{c}P, c, MS, fs \vdash exp_1 : ls \rightarrow ls_1, l_1 \\ P, c, MS, fs \vdash exp_2 : ls_1 \rightarrow ls_2, l_2\end{array}}{P, c, MS, fs \vdash exp_1\ binop\ exp_2 : ls \rightarrow ls_2, l'}\ l' = max\{l_1, l_2\}$$

Here the composed expression has the highest level of its subexpression. Secondly, the judgment means that in the expression no branching of control flow with a high condition exists. Consider for instance the type rule for deriving the judgment for the ternary operator:

$$\text{[CondExp]}\frac{\begin{array}{c}P, c, MS, fs \vdash exp : ls \rightarrow ls', low \\ P, c, MS, fs \vdash exp_1 : ls' \rightarrow ls'', l_1 \\ P, c, MS, fs \vdash exp_2 : ls' \rightarrow ls'', l_2\end{array}}{P, c, MS, fs \vdash (exp)\ ?\ exp_1 : exp_2 : ls \rightarrow ls'', l'}\ l' = max\{l_1, l_2\}$$

Here the expression of the condition is required to have the level *low*. Thirdly, the judgment means that when the local signature is ls before the execution of the expression the local signature is ls after the the execution. Consider for instance one rule for deriving the judgment for assignment expressions:

$$\text{[VarAssignExp1]}\frac{P, c, MS, fs \vdash exp : ls \rightarrow ls', l}{P, c, MS, fs \vdash x = exp : ls \rightarrow ls'', l}\ \begin{array}{l}isPrimitiveType(x) \vee (ls'(x) = \bot) \\ ls'' = ls'\{x \mapsto l\}\end{array}$$

where the predicate *isPrimitiveType* on variable names is a holds if the variable name is declared to be of a primitive type, for instance int. Here, after execution of the assignment expression, the target variable x has the security level of the expression exp. Finally, the judgment means that when program execution moves data, security levels of source and destination memory locations are such that no high data will be moved to low-typed locations during program execution. The rule for deriving the judgment for assignment expressions illustrates this as well.

The judgment for statements is:

$$P, c, MS, fs, l_{\text{ret}} \vdash stm : ls \rightarrow ls'$$

Its meaning is similar to that of the judgment for expressions, except that statements do not have a value for which level is stated, and that in case the statement returns, the return value has a level lower or equal to l_{ret}.

The judgment for methods is:

$$P, c, MS, fs \vdash md : ms$$

It means that the method declaration *md* within the class identified by *c* and within the program *P* complies to the method signature *ms*, given the set of method signatures *MS* and the field signatures *fs*, i.e. the body statement of the declaration can be typed. A method is typable against a method signature if its body is typable against a local signature and a return level that are specified by the method signature:

$$[\text{MthdDecl}]\frac{P, c, MS, fs, l_{\text{ret}} \vdash bodyOf(md) : ls \rightarrow ls'}{P, c, MS, fs \vdash md : (c, name(md), ls, l_{\text{ret}})}$$

In the following presentation of security type rules we omit P, c, MS, fs, and l_{ret} from the judgment for simplicity of presentation. In the rules we use various shorthands that are based on the structure of programs. The function $typeOf : Lang_{\text{Scope}} \rightarrow Types$ maps scopes to their static types. The functions $name : MethodDecl \rightarrow Names$, $paraNameSet : MethodDecl \rightarrow \mathcal{P}(Names)$, and $paraNameList : MethodDecl \rightarrow Names^*$ are selector functions on method declarations and map a given method declaration to its method name, its set of parameter names, or its list of parameter names, respectively. A method call expression does not necessarily determine a unique method to be executed. By polymorphism the actual method to be executed depends on the runtime-type of the object of which the method is called. In order to represent possible call-targets we introduce the following function. The function $poly : (Names \times Names) \rightarrow \mathcal{P}(Names \times MethodDecl)$ takes a class name and a method name and returns a set of pairs of class names and method declarations such that this set represents the methods that are possibly selected for execution. The rules for method calls where all possible targets need to be checked use *poly*.

The type rules usually break down the type check on the components of the respective statement or expression. The type rules for statements are presented in Figure 3. Note that, the rules for loops and conditionals additionally check that the condition has the security level *low* in order to detect potential timing channels. Judgments for some expressions like assignments have to update the local signature in some cases. Such updates usually are propagated through the syntactic structure by the rules for statements and the sequential composition in the rule [BlockStmt] connects the changes. For control structures the correct treatment of updates is a source for common mistakes, for instance forgetting that what holds after execution of the body of a loop is relevant for the condition of the loop. In order to avoid such mistakes we take a conservative approach and the rules [WhileStmt] and [ForStmt] for loops do require that in their components the local signature is not changed. For a similar reason the rule [IfStmt1] for conditionals requires equal changes in both branches and the rule [IfStmt2] permits no change at all such that the further statements only need to be considered with respect to one local signature.

The remaining type rules for expressions are presented in Figures 4 and 5. Note that, in the rules for expressions that change memory, like assignments, variable declarations, and method calls, it is checked that the data depending on memory locations with the level *high* is not moved to memory locations with the level *low*.

$$[\text{BlockStmt}]\ \frac{\vdash stm_0 : ls \to ls_0 \quad \vdash stm_1 : ls_0 \to ls_1 \quad \ldots \quad \vdash stm_{n-1} : ls_{n-1} \to ls_{n-1}}{\vdash stm_0\ stm_1\ \ldots\ stm_{n-1} : ls \to ls_{n-1}}$$

$$[\text{ExpStmt}]\ \frac{\vdash exp : ls \to ls', l}{\vdash exp; : ls \to ls'}$$

$$[\text{ReturnStmt1}]\ \frac{\vdash exp : ls \to ls', l}{\vdash \mathsf{return}\ exp; : ls \to ls'}\ l \leq l_{\text{ret}} \qquad [\text{ReturnStmt2}]\ \frac{}{\vdash \mathsf{return}; : ls \to ls}$$

$$[\text{IfStmt1}]\ \frac{\vdash exp : ls \to ls', low \quad \vdash stm_1 : ls' \to ls'' \quad \vdash stm_2 : ls' \to ls''}{\vdash \mathsf{if}\ (exp)\ \{stm_1\}\ \mathsf{else}\ \{stm_2\} : ls \to ls''}$$

$$[\text{IfStmt2}]\ \frac{\vdash exp : ls \to ls', low \quad \vdash stm : ls' \to ls'}{\vdash \mathsf{if}\ (exp)\ \{stm\} : ls \to ls'}$$

$$[\text{WhileStmt}]\ \frac{\vdash exp : ls \to ls, low \quad \vdash stm : ls \to ls}{\vdash \mathsf{while}\ (exp)\ \{stm\} : ls \to ls}$$

$$[\text{ForStmt}]\ \frac{\begin{array}{c}\vdash exp : ls \to ls, low \quad \vdash stm : ls \to ls \\ \vdash exp_0 : ls \to ls, l_0 \quad \ldots \vdash exp_{n-1} : ls_{n-2} \to ls, l_{n-1} \\ \vdash exp'_0 : ls \to ls, l'_0 \quad \ldots \vdash exp'_{n'-1} : ls_{n'-2} \to ls, l'_{n'-1}\end{array}}{\vdash \mathsf{for}\ (exp_0, \ldots, exp_{n-1}; exp; exp'_0, \ldots, exp'_{n'-1})\ \{stm\} : ls \to ls}$$

Figure 3: Security Type Rules for Statements

$$\text{[ConstExp]}\ \frac{}{\vdash const\ :\ ls \rightarrow ls, low}$$

$$\text{[NameExp1]}\ \frac{}{\vdash name\ :\ ls \rightarrow ls, l}\ ls(name) = \lfloor l \rfloor$$

$$\text{[NameExp2]}\ \frac{}{\vdash name\ :\ ls \rightarrow ls, l}\ \begin{array}{l} ls(name) = \bot \\ fs(c, name) = \lfloor l \rfloor \end{array}$$

$$\text{[FieldAccExp1]}\ \frac{}{\vdash scope.f\ :\ ls \rightarrow ls, l''}\ \begin{array}{l} fs(typeOf(scope), f) = \lfloor l \rfloor \\ ls(scope) = \lfloor l' \rfloor \\ l'' = max\{l, l'\} \end{array}$$

$$\text{[FieldAccExp2]}\ \frac{}{\vdash scope.f\ :\ ls \rightarrow ls, l''}\ \begin{array}{l} ls(scope) = \bot \\ fs(typeOf(scope), f) = \lfloor l \rfloor \\ fs(c, scope) = \lfloor l' \rfloor \\ l'' = max\{l, l'\} \end{array}$$

$$\text{[ArrAccExp1]}\ \frac{\vdash exp\ :\ ls \rightarrow ls', l'}{\vdash name[exp]\ :\ ls \rightarrow ls', l''}\ \begin{array}{l} ls(name) = \lfloor l \rfloor \\ l'' = max\{l, l'\} \end{array}$$

$$\text{[ArrAccExp2]}\ \frac{\vdash exp\ :\ ls \rightarrow ls', l'}{\vdash name[exp]\ :\ ls \rightarrow ls', l''}\ \begin{array}{l} ls(name) = \bot \\ fs(c, name) = \lfloor l \rfloor \\ l'' = max\{l, l'\} \end{array}$$

$$\text{[ArrAccExp3]}\ \frac{\vdash exp\ :\ ls \rightarrow ls', l''}{\vdash scope.f[exp]\ :\ ls \rightarrow ls', l'''}\ \begin{array}{l} fs(typeOf(scope), f) = \lfloor l \rfloor \\ ls(scope) = \lfloor l' \rfloor \\ l''' = max\{l, l', l''\} \end{array}$$

$$\text{[ArrAccExp4]}\ \frac{\vdash exp\ :\ ls \rightarrow ls', l''}{\vdash scope.f[exp]\ :\ ls \rightarrow ls', l'''}\ \begin{array}{l} ls(scope) = \bot \\ fs(typeOf(scope), f) = \lfloor l \rfloor \\ fs(c, scope) = \lfloor l' \rfloor \\ l'' = max\{l, l'\} \end{array}$$

$$\text{[EnclExp]}\ \frac{\vdash exp\ :\ ls \rightarrow ls', l}{\vdash (exp)\ :\ ls \rightarrow ls', l} \qquad \text{[CastExp]}\ \frac{\vdash exp\ :\ ls \rightarrow ls', l}{\vdash (ty)exp\ :\ ls \rightarrow ls', l}$$

$$\text{[UnopExp]}\ \frac{\vdash exp\ :\ ls \rightarrow ls', l}{\vdash unop\ exp\ :\ ls \rightarrow ls', l}$$

Figure 4: Security Type Rules for Expressions (part 1)

$$[\text{VarAssignExp2}]\ \frac{\vdash exp\ :\ ls \to ls', l}{\vdash x = exp\ :\ ls \to ls', l}\ \begin{array}{l}\neg isPrimitiveType(x)\\ ls'(x) = \lfloor l \rfloor\end{array}$$

$$[\text{ArrAssignExp1}]\ \frac{\vdash exp_1\ :\ ls \to ls_1, l' \qquad \vdash exp_2\ :\ ls_1 \to ls_2, l}{\vdash name[exp_1] = exp_2\ :\ ls \to ls_2, l}\ \begin{array}{l}ls(name) = \lfloor l \rfloor\\ l' \leq l\end{array}$$

$$[\text{ArrAssignExp2}]\ \frac{\vdash exp_1\ :\ ls \to ls_1, l' \qquad \vdash exp_2\ :\ ls_1 \to ls_2, l}{\vdash name[exp_1] = exp_2\ :\ ls \to ls_2, l}\ \begin{array}{l}ls(name) = \bot\\ fs(c, name) = \lfloor l \rfloor\\ l' \leq l\end{array}$$

$$[\text{ArrAssignExp3}]\ \frac{\vdash exp_1\ :\ ls \to ls_1, l'' \qquad \vdash exp_2\ :\ ls_1 \to ls_2, l'''}{\vdash scope.f[exp_1] = exp_2\ :\ ls \to ls_2, l'''}\ \begin{array}{l}fs(typeOf(scope), f) = \lfloor l \rfloor\\ l'' \leq l'''\\ ls(scope) = \lfloor l' \rfloor\\ l''' = max\{l, l'\}\end{array}$$

$$[\text{ArrAssignExp4}]\ \frac{\begin{array}{c}\vdash exp_1\ :\ ls \to ls_1, l''\\ \vdash exp_2\ :\ ls_1 \to ls_2, l'''\end{array}}{\vdash scope.f[exp_1] = exp_2\ :\ ls \to ls_2, l'''}\ \begin{array}{l}ls(name) = \bot\\ fs(typeOf(scope), f) = \lfloor l \rfloor\\ fs(c, name) = \lfloor l' \rfloor\\ l''' = max\{l, l'\}\end{array}$$

$$[\text{VarDeclExp}]\ \frac{\vdash VarDecl_0\ :\ ls \to ls_0, l_0 \quad \ldots \quad \vdash VarDecl_{n-1}\ :\ ls_{n-2} \to ls_{n-1} l_{n-1},}{\vdash typename\ VarDecl_0, \ldots, VarDecl_{n-1}\ :\ ls \to ls_{n-1}, l_{n-1}}$$

$$[\text{VarDecl1}]\ \frac{\vdash exp\ :\ ls \to ls', l}{\vdash x{=}exp\ :\ ls \to ls'', l}\ ls'\{x \mapsto l\} = ls'' \qquad [\text{VarDecl2}]\ \frac{}{\vdash x\ :\ ls \to ls, low}$$

$$[\text{CallExp1}]\ \frac{\vdash exp_0\ :\ ls \to ls_0, l_0 \quad \ldots \quad \vdash exp_{n-1}\ :\ ls_{n-2} \to ls_{n-1}, l_{n-1}}{\vdash m(exp_0, \ldots, exp_{n-1})\ :\ ls \to ls_{n-1}, l'_{\text{ret}}}\ SideCond1$$

where *SideCond1* is $\forall (c', md) \in poly(c, m).$
$\exists ls_{\text{called}} : paraNameSet(md) \to L.$
$$\left[\begin{array}{l}(c', name(md), ls_{\text{called}}, l'_{\text{ret}}) \in MS\\ \wedge\ P, c', MS, fs \vdash md\ :\ (c', name(md), ls_{\text{called}}, l'_{\text{ret}})\\ \wedge\ map(ls_{\text{called}}, paraNameList(md)) = l_0 \ldots l_{n-1}\end{array}\right]$$

$$[\text{CallExp2}]\ \frac{\begin{array}{c}\vdash scope\ :\ ls \to ls, low\\ \vdash exp_0\ :\ ls \to ls_0, l_0 \quad \ldots \quad \vdash exp_{n-1}\ :\ ls_{n-2} \to ls_{n-1}, l_{n-1}\end{array}}{\vdash scope.m(exp_0, \ldots, exp_{n-1})\ :\ ls \to ls_{n-1}, l'_{\text{ret}}}\ SideCond2$$

where *SideCond2* is $\forall (c', md) \in poly(typeOf(scope), m).$
$\exists ls_{\text{called}} : paraNameSet(md) \to L.$
$$\left[\begin{array}{l}(c', name(md), ls_{\text{called}}, l'_{\text{ret}}) \in MS\\ \wedge\ P, c', MS, fs \vdash md\ :\ (c', name(md), ls_{\text{called}}, l'_{\text{ret}})\\ \wedge\ map(ls_{\text{called}}, paraNameList(md)) = l_0 \ldots l_{n-1}\end{array}\right]$$

Figure 5: Security Type Rules for Expressions (part 2)

5 Examples

In this Section we demonstrate how SCF 1.0 can be applied on existing implementations of cryptographic algorithms. We present three examples that cover three software implementations of algorithms for encryption or decryption in two different cryptographic libraries.

5.1 AES (Implementation in FlexiProvider)

At first let us consider the encryption standard AES as implemented in the library *FlexiProvider* (Version 1.6p7) [Fle10]. The goal of the analysis is to protect the secret key at decryption (encryption works in the same manner).

Decryption is implemented in the method singleBlockDecrypt(byte[] input, int inOff, byte[] output, int outOff) of the class Rijndael in the package de.flexiprovider.core.rijndael (see an excerpt in Figure 6). For executing the decryption itself, an expansion of the symmetric secret key is used. The already expanded secret key for decryption is stored in the field Ki of type int[].

The first step to analyze the decryption method with SCF 1.0 is creating configuration files and a policy, as defined in Section 3. SCF 1.0 can do this automatically. Most notably, it automatically initializes the policy with all relevant field- and method-signatures, where all levels are set to 0 (*low*). The command for executing the initialization is

```
java userinterfaces.SimpleCommandLine init "$ANALYZEPATH" "AES" \
  "de.flexiprovider.core.rijndael" "Rijndael" \
  "singleBlockDecrypt(byte[], int, byte[], int)" \
  "$SRCPATH/de/flexiprovider/core/rijndael/Rijndael.java" \
  "$SRCPATH/de/flexiprovider/common/util/BigEndianConversions.java
```

where $ANALYZEPATH is the directory for the analysis configuration and result files, whereas $SRCPATH is the path to the source files of FlexiProvider. The argument init commands SCF 1.0 to initialize the files, the argument "AES" provides the prefix for configuration files and result files, the following three arguments specify the method to be analyzed, and the last two arguments specify the source files that contain relevant code. The command generates three files: (a) a policy file, AES.level, (b) a file containing the paths to the source files that contain relevant code, AES.program, and (c) a file to configure the whole analysis, AES.analysis. The last file stores the method to be analyzed, paths to the other two files, and the names of the output files AES.log and AES.report.

The next step is to specify the secret input as well as the fields or method parameters that potentially contain confidential information. The field Ki holds the expanded secret key, hence, we specify the field Ki to contain a secret, i.e., we modify the file AES.level such that Ki gets assigned the level 1. For completeness, we also associate the field K with the level 1, which actually is not used by the decryption routine but which contains the expansion of the secret key for encryption. We also associate the parameter output of the method singleBlockDecrypt with the level 1, because it is used to store the results of decryption and, thus, it will contain information that depends on the secret key. Note, that leaving the level of output at 0 would not result in a successful analysis because SCF 1.0 would report a violation.

```
protected void singleBlockDecrypt(byte[] input, int inOff, byte[]
    output, int outOff) {

        int i, j;
        int d0, d1, d2, d3, d4, d5, d6, d7;
        int a0, a1, a2, a3, a4, a5, a6, a7;
        [...]
        if (blockSize == 8) {

            // convert input bytes to ints
            d0 = BigEndianConversions.OS2IP(input, inOff);
            [...]
            // XOR keys and data
            d0 ^= Ki[0];
            [...]
            d7 ^= Ki[7];

            // (n - 1) transformation rounds
            for (j = 1; j < numRounds; j++) {
                i = j * blockSize;
                a0 = T0i[(d0>>>24) & 0xff] ^ T1i[(d7>>16) & 0xff]
                ^ T2i[(d5>>8) & 0xff] ^ T3i[d4 & 0xff] ^ Ki[i];
                [...]
                a7 = T0i[(d7>>>24) & 0xff] ^ T1i[(d6>>16) & 0xff]
                ^ T2i[(d4>>8) & 0xff] ^ T3i[d3 & 0xff] ^ Ki[i + 7];
                [...]
                d7 = a7;
            }

            // convert ints to output bytes plus last
               transformation round
            i = numRounds * blockSize;
            output[outOff++] = (byte) (Si[(d0>>>24) & 0xff]
                               ^ (Ki[i]>>>24));
            [...]
        }
    }
```

Figure 6: Excerpt of the AES decryption method in FlexiProvider [Fle10]

Figure 7 shows an excerpt of the resulting policy.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<informationLevelModel>
    <package name="de.flexiprovider.common.util">
        <classSignature name="BigEndianConversions">
            [...]
        </classSignature>
    </package>
    <package name="de.flexiprovider.core.rijndael">
       <classSignature name="Rijndael">
            [...]
            <fieldLevel name="T2i">0</fieldLevel>
            <fieldLevel name="Ki">1</fieldLevel>
            <fieldLevel name="blockSize">0</fieldLevel>
            [...]
            <methodSignature name="singleBlockDecrypt(byte[],␣int,␣
                byte[],␣int)">
                <parameterLevel name="input">0</parameterLevel>
                <parameterLevel name="inOff">0</parameterLevel>
                <parameterLevel name="output">1</parameterLevel>
                <parameterLevel name="outOff">0</parameterLevel>
            </methodSignature>
            [...]
        </classSignature>
    </package>
</informationLevelModel>
```

Figure 7: Excerpt of the policy for the AES decryption method

Finally, we actually run the analysis by issuing the following command where we just supply the configuration file for the analysis:

```
java userinterfaces.SimpleCommandLine
    analyze "$ANALYZEPATH/AES.analysis"
```

The resulting file AES.report contains:

```
===Analysis Report===

Analysis completed: true

==Overloading Warnings==
```

```
de.flexiprovider.common.util.BigEndianConversions.I2OSP(long) has
    the same name and amount of parameters like an already existing
     method
de.flexiprovider.common.util.BigEndianConversions.I2OSP(long, byte
    [], int) has the same name and amount of parameters like an
    already existing method
==Assignment Violations==
Amount: 0
Violations:
==Undefined level Violations==
Amount: 0
Violations:
==Branching Violations==
Amount: 0
Violations:
==Branchings with undefined level==
Amount: 0
Violations:
==Definitions not found==
Amount: 0
Violations:
```

From that we, first of all, obtain that the analysis has run through (Analysis completed: true). Secondly, we obtain a warning about overloading, which SCF 1.0 supports only partially based on the number of arguments. However, the methods warned about actually are never called within the analyzed method, hence it is not relevant for this analysis. Finally, we obtain that no violations have been found, that is neither a branching with a branch condition that depends on secrets, nor that we have specified inconsistent security levels.

5.2 IDEA (Implementation in FlexiProvider)

Now let us consider the encryption scheme IDEA as implemented in the library FlexiProvider. 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 (see an excerpt in Figure 8). Each round of IDEA encryption uses different bits of the secret key. These round keys are scheduled before the actual encryption. The scheduled secret key for encryption is stored in the field encr of type int[].

As the first step we initialize the configuration files and the policy with a command:

```
java userinterfaces.SimpleCommandLine init "$ANALYZEPATH" "IDEA" \
  "de.flexiprovider.core.idea" "IDEA" \
  "singleBlockEncrypt(byte[], int, byte[], int)" \
  "$SRCPATH/de/flexiprovider/core/idea/IDEA.java"
```

```
protected void singleBlockEncrypt(byte[] input, int inOff,
   byte[] output, int outOff) {
    encryptDecrypt(encr, input, inOff, output, outOff);
}
[...]
private void encryptDecrypt(int[] key, byte[] in, int in_offset,
        byte[] out, int out_offset) {
    [...]
    int x0 = in[in_offset++] << 8;
    x0 |= in[in_offset++] & 0xff;
    [...]
    for (int i = 0; i < rounds; ++i) {
        x0 = mulMod16(x0, key[k++]);
        x1 += key[k++];
        x2 += key[k++];
        x3 = mulMod16(x3, key[k++]);
        [...]
    }
    [...]
    out[out_offset++] = (byte) (x0 >>> 8);
    out[out_offset++] = (byte) x0;
    [...]
}
[...]
private int mulMod16(int a, int b) {
    int p;
    a &= mulMask;
    b &= mulMask;

    if (a == 0) {
        a = mulModulus - b;
    } else if (b == 0) {
        a = mulModulus - a;
    } else {
        p = a * b;
        b = p & mulMask;
        a = p >>> 16;
        a = b - a + (b < a ? 1 : 0);
    }

    return a & mulMask;
}
```

Figure 8: Excerpt of the IDEA encryption method in FlexiProvider [Fle10]

where $ANALYZEPATH is the directory for the analysis configuration files and result files, and $SRCPATH is the path to the source files of FlexiProvider.

The next step is again to specify the security levels. We modify the file IDEA.level such that the field encr for the scheduled encryption key is set to level 1 (and similarly the scheduled decryption key decr for completeness). Running the analysis with this policy reveals that method parameters at several points in the program are instantiated with confidential data or are used to store confidential date. Hence we also set these parameters to the level 1, 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 9 shows an excerpt of the resulting policy.

Finally, we actually run the analysis by issuing the following command, where we supply the configuration file for the analysis:

```
java userinterfaces.SimpleCommandLine
    analyze "$ANALYZEPATH/IDEA.analysis"
```

The resulting file IDEA.report contains:

```
===Analysis Report===

Analysis completed: true

==Assignment Violations==
Amount: 0
Violations:
==Undefined level Violations==
Amount: 0
Violations:
==Branching Violations==
Amount: 3
Violations:
de.flexiprovider.core.idea.IDEA.mulMod16(int, int)[(1, 1)] in line
    407;
        Branching with non public condition: a == 0
de.flexiprovider.core.idea.IDEA.mulMod16(int, int)[(1, 1)] in line
    409;
        Branching with non public condition: b == 0
de.flexiprovider.core.idea.IDEA.mulMod16(int, int)[(1, 1)] in line
    422;
        Branching with non public condition: b < a
==Branchings with undefined level==
Amount: 0
Violations:
==Definitions not found==
Amount: 0
Violations:
```

```
<?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 9: Excerpt of the policy for the IDEA encryption method

From this report we, firstly, obtain that the analysis has run through (Analysis completed: true). Next, we see that there are branches with a condition that depends on confidential data. Inspecting the findings in the source (Figure 8) of the method mulMod16 reveal that there actually are certain values of the parameters that result in a special treatment, which is realized by branching on their values. The finding constitutes a known timing channel vulnerability for implementations of IDEA [KSWH00].

5.3 DES (Implementation in GNU classpath)

As the third example let us consider an implementation in a second Java library that contains implementations of cryptographic algorithms, namely the *GNU Classpath* (version 0.98) [GNU09], which aims at providing free core class libraries for Java. The goal of the analysis is to protect the secret key at DES-encryption or -decryption.

The encryption is implemented in the method desFunc(byte[] in, int i, byte[] out, int o, int[] key) of the class DES in the package gnu.javax.crypto.cipher (see an excerpt in Figure 10). The secret key (also already expanded) is passed as the parameter key.

Note that here we analyze the actual core of the encryption scheme. The class DES contains wrappers encrypt and decrypt which essentially pass either the expanded key for encryption or the expanded key for decryption. These two wrapper methods cannot be analyzed by SCF 1.0, because they apply a method call on a casted object, which is out of scope of SCF 1.0 (see Section 2). However, the actual routine is implemented in desFunc, which can be analyzed well.

The first step is again automatic initialization:

```
java userinterfaces.SimpleCommandLine init "$ANALYZEPATH" "DES" \
  "gnu.javax.crypto.cipher" "DES" \
  "desFunc(byte[], int, byte[], int, byte[])" \
  "$SRCPATH/gnu/javax/crypto/cipher/DES.java"
```

where $ANALYZEPATH is the directory for the analysis configuration files and result files, $SRCPATH is the path to the source files of GNU Classpath.

We specify level 1 for the secret method parameter key, which holds the secret to be protected, and the method parameter out, in which the results are stored. See Figure 9 for an excerpt of the resulting policy. Now we run the analysis by issuing the following command, where we supply the configuration file for the analysis:

```
java userinterfaces.SimpleCommandLine
    analyze "$ANALYZEPATH/DES.analysis"
```

The resulting file DES.report contains:

```
===Analysis Report===

Analysis completed: true

==Assignment Violations==
Amount: 0
```

```
private static void desFunc(byte[] in, int i, byte[] out,
    int o, int[] key)
{
  int right, left, work;
  // Load.
  left =  (in[i++] & 0xff) << 24
        | (in[i++] & 0xff) << 16
        | (in[i++] & 0xff) << 8
        |  in[i++] & 0xff;
  [...]
  // Initial permutation.
  work = ((left >>> 4) ^ right) & 0x0F0F0F0F;
  left ^= work << 4;
  right ^= work;

  [...]

  right = ((right << 1) | ((right >>> 31) & 1)) & 0xFFFFFFFF;
  work = (left ^ right) & 0xAAAAAAAA;
  left ^= work;
  right ^= work;
  left = ((left << 1) | ((left >>> 31) & 1)) & 0xFFFFFFFF;

  int k = 0, t;
  for (int round = 0; round < 8; round++)
    {
      work = right >>> 4 | right << 28;
      work ^= key[k++];
      t = SP7[work & 0x3F];
      [...]
      left ^= t;

      [...]
    }
  [...]

  out[o++] = (byte)(right >>> 24);
  [...]
  out[o  ] = (byte) left;
}
```

Figure 10: Excerpt of the DES encryption- and decryption method in GNU Classpath [GNU09]

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<informationLevelModel>
    <package name="gnu.javax.crypto.cipher">
        <classSignature name="DES">
            [...]
            <methodSignature name="desFunc(byte[], int, byte[], int
                , int[])">
                <parameterLevel name="in">0</parameterLevel>
                <parameterLevel name="i">0</parameterLevel>
                <parameterLevel name="out">1</parameterLevel>
                <parameterLevel name="o">0</parameterLevel>
                <parameterLevel name="key">1</parameterLevel>
            </methodSignature>
            [...]
        </classSignature>
    </package>
</informationLevelModel>
```

Figure 11: Excerpt of the policy for the DES encryption- and decryption method

```
Violations:
==Undefined level Violations==
Amount: 0
Violations:
==Branching Violations==
Amount: 0
Violations:
==Branchings with undefined level==
Amount: 0
Violations:
==Definitions not found==
Amount: 0
Violations:
```

This means SCF 1.0 has completed the analysis successfully on this third implementation and did not find any violations.

6 Related Work

Timing channel attacks on cryptographic systems have been explored since 15 years [Koc96]. They have been practically demonstrated [BB05], optimized [Sch05], and evaluated [SMY09].

We are not aware of an existing tool that automatically analyzes programs implemented in Java source code for potential timing channels. However, there are implementations of detection and transformation mechanisms for analyzing implementations in other programming languages. In [Aga01] an approach of automatically transforming out timing leaks by cross-copying [Aga00] is evaluated by realizing it for subset of Java bytecode without objects. The paper [MPSW05] suggests and realizes transforming out timing leaks in C programs (without function calls and pointers) by encoding conditional branches into assignments of expressions. In [CVBS09] a transformation at the compiler back-end for x86 code is suggested based on an exploration of the actual timing behavior of x86-instructions.

The analysis mechanism of SCF 1.0 is based on security type systems for information flow control [VSI96], and the security model of SCF 1.0 on the noninterference-like information flow properties that such type systems enforce. JIF [JIF08] (introduced in [Mye99]) provides information flow security by a security type system for an extended version of Java. In comparison, SCF 1.0 is applicable to unmodified Java programs.

The language-based information flow community has addressed the issue of timing channels also in settings where the attacker cannot observe the time itself [SM03]. When considering multi-threaded programs, the running time of a thread may influence the decisions of a scheduler, which can resolve non-determinism that is caused by race conditions such that the decision becomes visible to an attacker by different observable values. Several approaches to define security properties that adequately capture this problem exist: by imposing requirements about the scheduler-visible timing-behavior of threads based on a strong bisimulation (strong security condition) [SS00], by forbidding certain non-determinism in programs [ZM03, HWS06], or by using schedulers with a special interface [RS06]. A novel approach tries to relax the strict requirements of the strong security condition by providing and exploiting a precise specification of realistic schedulers [MS10]. Especially the first approach is interesting if considering attackers that can observe the time directly, because it does not address the effects but the cause (timing) itself. Already [SS00] provides a security type system for transforming out timing leaks by cross-copying of branches. A more advanced transformation for the same security condition based on unification of branches is provided in [KM07]. The strong security condition itself also has been extended with declassification [MR07, LM09].

7 Conclusion

We have presented SCF 1.0, a tool that can detect potential timing channel vulnerabilities in implementations of cryptographic algorithms in the Java source language. SCF 1.0 covers a non-trivial subset of Java (see Section 2) 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).

We also presented the policy language (see Section 3) which SCF 1.0 reads and that determines the secrets that must not be accessible 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.

SCF 1.0 is designed to analyze programs for branching of the control flow at which the branch taken depends on confidential input. We carefully crafted a security type system (see Section 4) which automatically checks whether this is the case for a given Java program.

The examples shown (see Section 5) that SCF 1.0 is sufficiently mature to analyze several actual implementations of cryptographic algorithms in existing cryptographic libraries.

Acknowledgments. We thank Markus Aderhold for helpful comments and Mohamed El Yousfi for helpful discussion. This work was supported by CASED (`www.cased.de`).

References

[Aga00] J. Agat. Transforming out Timing Leaks. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages (POPL)*, pages 40–53, 2000.

[Aga01] J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Chalmers University of Technology, 2001.

[BB05] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[CVBS09] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (S&P*, pages 45–60, Washington, DC, USA, 2009. IEEE Computer Society.

[Fle10] FlexiProvider – A Toolkit for the Java Cryptography Architecture (JCA/JCE). see `http://www.flexiprovider.de`, 2010.

[Ges10] J. V. Gesser. javaparser. see `http://code.google.com/p/javaparser/`, 2010.

[GNU09] GNU Classpath. see `http://www.gnu.org/software/classpath/`, 2009.

[HWS06] M. Huisman, P. Worah, and K. Sunesen. A Temporal Logic Characterisation of Observational Determinism. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 3–15. IEEE, 2006.

[JIF08] JIF: Java + information flow). see `http://www.cs.cornell.edu/jif/`, 2008.

[KM07] B. Köpf and H. Mantel. Transformational Typing and Unification for Automatically Correcting Insecure Programs. *International Journal of Information Security (IJIS)*, 6(2–3):107–131, 2007.

[Koc96] P. C. Kocher. 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*, LNCS 1109, pages 104–113. Springer-Verlag, 1996.

[KSWH00] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2,3):141–158, 2000.

[LM09] A. Lux and H. Mantel. Declassification with Explicit Reference Points. In M. Backes and P. Ning, editors, *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)*, LNCS 5789, pages 69–85. Springer, 2009.

[MPSW05] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. 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)*, LNCS 3935, pages 156–168, 2005.

[MR07] H. Mantel and A. Reinhard. Controlling the What and Where of Declassification in Language-Based Security. In *Proceedings of the 16th European Symposium on Programming (ESOP)*, LNCS 4421, pages 141–156. Springer, 2007.

[MS10] H. Mantel and H. Sudbrock. Flexible scheduler-independent security. In *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS)*, LNCS 6345, pages 116–133. Springer, 2010. (to appear).

[Mye99] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[RS06] A. Russo and A. Sabelfeld. Securing Interaction between Threads and the Scheduler. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 177–189. IEEE, 2006.

[Sch05] W. Schindler. 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*, LNCS 3386, pages 85–103, 2005.

[SM03] A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.

[SMY09] François-Xavier Standaert, Tal Malkin, and Moti Yung. 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*, volume LNCS 5479, pages 443–461, 2009.

[SS00] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 200–215, Cambridge, UK, 2000.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[ZM03] S. Zdancewic and A. C. Myers. Observational Determinism for Concurrent Program Security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 29–43. IEEE, 2003.