

Validy® SoftNaos for Java® Evaluation Guide

Copyright © 2007, 2008 Validy Net, Inc., S.A. Validy

- Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Validy Net Inc. and S.A. Validy are independent of Sun Microsystems, Inc.
- Apache is a trademark of The Apache Software Foundation.
- Windows is a registered trademark of Microsoft Corporation in the United States and other countries.
- Linux is the registered trademark of Linus Torvalds in the United States and other countries.
- Validy is a registered trademark of Validy Net Inc. and S.A. Validy.

COLLABORATORS

	<i>TITLE :</i> Validy® SoftNaos for Java® Evaluation Guide		<i>REFERENCE :</i>
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Validy Net, Inc. and S.A. Validy	March 31, 2008	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.0.8	28 March 2008	Minor change in the verbose option.	CV
1.0.7	6 Feb 2008	Documented the support for object serialization and the option for backward compatibility. Documented the methods to select the version of the token instruction set to be used by the translator and the virtual runtime. Renaming to SoftNaos	CV
1.0.6	28 Sep 2007	Replaced the class path by the application, library, and system paths. Added a section about the Ant task dependencies. Added the enabled parameter to annotations and a section on markers files. Documented the secure call graph builder tool.	CV
1.0.5	14 Aug 2007	Added a reference to the runtime that interfaces with the actual coprocessor.	CV
1.0.4	26 Jun 2007	Added support for byte, short and enum types.	CV

REVISION HISTORY			
-------------------------	--	--	--

NUMBER	DATE	DESCRIPTION	NAME
1.0.1	24 Apr 2007	Initial release	CV

Contents

1	Introduction	1
1.1	Organization of this Guide	1
1.2	Limitations of the Evaluation Version	1
1.3	Third-Party Libraries and Tools	2
2	Getting Started	3
2.1	Installing the tools	3
2.2	Annotating fields and methods	3
2.3	Running the translator	4
2.4	Running the protected application	5
2.5	Looking at the communication	5
3	Overview	6
3.1	Subtractive Protection	6
3.1.1	Data	6
3.1.2	Instructions	8
3.2	Detection and Coercion	8
3.2.1	Setting Tags	9
3.2.2	Checking Tags	9
3.2.3	Linking through Tags	10
3.2.4	Call protection	10
3.2.5	Other applications	11
3.3	Runtime	11
3.4	Virtual Coprocessor	11
4	Usage	12
4.1	Class Domains	12
4.2	Selecting and Marking Fields and Methods	13

4.2.1	Fields	13
4.2.2	Methods	15
4.2.3	Call Graphs	18
4.2.4	Annotations	18
4.2.5	Markers	19
4.2.6	Object Serialization	20
4.2.6.1	Introduction	20
4.2.6.2	Default Behavior	21
4.2.6.3	Backward Compatibility	21
4.2.6.4	Caveats	22
4.3	Running the Tools	22
4.3.1	Translator	22
4.3.1.1	Command Line	22
4.3.1.2	Integrating with an Ant build script	24
4.3.1.2.1	Configuration	24
4.3.1.2.2	Parameters specified as Attributes	25
4.3.1.2.3	Parameters specified as Nested Elements	25
4.3.1.2.4	Dependencies	27
4.3.2	Secure Call Graph Builder	27
4.3.2.1	Command Line	27
4.3.2.2	Integrating with an Ant build script	28
4.3.2.2.1	Configuration	28
4.3.2.2.2	Parameters specified as Attributes	28
4.3.2.2.3	Parameters specified as Nested Elements	28
4.4	Running the protected application	29
A	Single user license for the Validy SoftNaos for Java software package, evaluation version 1.0	30
A.1	Object	30
A.2	Period of authorization	30
A.3	Scope of license	30
A.3.1	30
A.3.2	31
A.3.3	31
A.3.4	31
A.3.5	31
A.3.6	31

A.4	Absence of financial compensation	32
A.5	Product keys	32
A.6	Entire agreement	32
A.6.1	32
A.6.2	32
A.6.3	32
A.6.4	32
A.6.5	33
A.7	Severability	33
A.8	Installation and distribution by your care of the usage rights	33
A.8.1	33
A.8.2	33
A.8.3	33
A.8.4	33
A.8.5	34
A.9	Disclosure of the evaluation results	34
A.10	Backup copies	34
A.11	Documentation	34
A.12	Free transfer to a third party	35
A.13	Export restrictions	35
A.14	Applicable law	35
A.15	Limitation and exclusion of responsibility in case of damage	35
A.16	Limited warranty	35
A.16.1	Limited warranty	35
A.16.2	Period of warranty	35
A.16.3	Exclusions from warranty	36
A.17	Additional functionalities	36

B Third Party Libraries and Tools Licenses 37

List of Figures

3.1 Relocation of fields to the coprocessor	7
---	---

List of Tables

4.1	Attributes for the vldy task	25
4.2	Some attributes inherited from the java task	26
4.3	Attributes for the vldygraph task	29

Chapter 1

Introduction

Validy Technology is an innovative subtractive and hardware-based software protection method that works by splitting an application into two parts: the first part, still executed by the original processor, and the second part that is executed by a secure coprocessor. At application protection time, the instructions that constitute the second part are ciphered using a secret key and embedded inside the first part. As the protected application executes, the ciphered instructions are sent to the coprocessor where they are decrypted and executed. Data is also exchanged back and forth and the secure coprocessor maintains a subset of the application's state inside its own memory.

When the second part is an essential subset of the original application, the protected application cannot be run without its secure coprocessor.

At the present time, the process of splitting the application is semi automatic, performed by a tool from hints provided by the developer. This document is a guide to the evaluation version of such a tool, Validy SoftNaos, specialized in the transformation of Java® classes. The tool is a bytecode partial translator: it transforms already compiled Java class files by converting a subset of the original Java instructions to the instruction set of the secure coprocessor. In the build of an application, the translator can be called at the same stage as the bytecode weaver of Aspect Oriented Programming. However, in contrast with a weaver, it does not add extra code to the application but simply translates some of the existing code.

1.1 Organization of this Guide

Chapter 2 describes how to install the translator, run it to protect a simple application, and then run the protected application.

Chapter 3 gives more details on Validy Technology.

Chapter 4 shows how to use the translator or its associated Ant task and describes their respective options.

1.2 Limitations of the Evaluation Version

The purpose of the evaluation version of Validy SoftNaos is to let interested parties study Validy Technology software protection in depth. Several aspects of the technology can be scrutinized.

- The selection of fields and methods to protect.
 - The transformation of the bytecode by the translator.
-

- The communication between the transformed program and the coprocessor.

An important asset of the technology is that it does not depend on hidden procedures. The technique is not strong because it performs a secret transformation of the bytecode but because it uses a secret key in a tamper resistant coprocessor. An analog is the difference between restricted cryptographic algorithms for which the algorithm itself must be kept secret and unrestricted ones that combine a public algorithm with a secret or private key. The virtual coprocessor, a pure software implementation of the secure coprocessor, is part of the evaluation package so that applications protected by the translator can be executed. This coprocessor is *not* tamper resistant. By running the protected application under a debugger, it is possible to access the state of the coprocessor and the instructions after they have been deciphered. Therefore, *this evaluation package alone does not provide any actual security*.

With a secure hardware coprocessor, the stream of data and instructions exchanged between the application and the coprocessor is still open to scrutiny and modification. However, to observe the hidden state and the deciphered instructions, one must break the secure processor.

In the current version, the use of a software coprocessor makes it difficult to estimate the performance of the transformed application when using a hardware coprocessor. The hardware coprocessor runs slower than the main processor and communicates with it through a bus that limits the rate and latency of the communications but it uses a hardware accelerator to decipher the instructions.

This package is distributed under a Validy license that can be found in Appendix [A](#).

1.3 Third-Party Libraries and Tools

Validy SoftNaos for Java depends upon the following third-party libraries and tools:

- The [ASM](#) bytecode manipulation framework to read and write Java class files.
- The [Apache Jakarta Commons](#)' CLI library to parse command line arguments.
- The [IzPack](#) Java installer to package the tool and its support libraries.

Many thanks to their authors and contributors. The licenses under which these libraries and tools are distributed can be found in Appendix [B](#).

Chapter 2

Getting Started

You need an installation of the Java 5 Runtime Environment or higher to install and run the translator. The Validy SoftNaos tools are written in Java. It has been tested on the Windows® XP operating system and the Linux® operating system but should run on any platform supported by Java 5 RE.

In the current version, the runtime that interfaces with the hardware secure coprocessor uses the `smartcardio` package under `javax`. This package is new to Java 6 so the Java 6 Runtime Environment is required to run a protected application with the hardware coprocessor. This limitation will be removed in a future release.

2.1 Installing the tools

Simply run the `vldy-softnaos-eval-1.0.x.y.jar` installer. After execution, you will find the following directories under the install root location you have chosen:

lib

The jar files for the translator, annotations, Ant task, virtual coprocessor, and interface with the secure coprocessor.

lib\libs

The third party libraries used by the translator.

doc

This evaluation guide in PDF and HTML formats.

examples

Sample applications.

2.2 Annotating fields and methods

To trigger the transformations applied by the translator, some fields and methods must be annotated with the `SecureField` and `SecureMethod` annotations from package `com.validy.technology.annotation`. Annotated fields will be removed from their original class to be stored inside the secure coprocessor and calls to secured methods will be transformed so they cannot be removed from the application. These transformations are explained in more details in Chapter 3. You can look at the "minimal" application example and see that one field and two methods from `Minimal` are annotated as shown below.

```
public class Minimal {

    @SecureField
    private int counter = 0;

    @SecureMethod
    public void inc() {
        counter++;
    }

    @SecureMethod
    public boolean overflow() {
        return counter > 10;
    }
    ...
}
```

2.3 Running the translator

To protect the minimal application, first make a copy of the `minimal` directory somewhere you have write access. From the copied `minimal` directory, run the following command:

```
java -jar <install path>/lib/vldy-tech-translator.jar -v -i ./classes -o ./sclasses -x ./dasm
```

This tells the translator to transform classes found under the `classes` directory, put the modified classes under the `sclasses` directory, and put disassembled listings of the class files before and after transformation under the `dasm` directory. These and other options are described in more details in Section 4.3.

Under the `dasm` directory, two subdirectories `input` and `output` contain a listing of the bytecode for the `Minimal` class, before and after the transformation. By comparing these files, it is possible to see that the `counter` field and its annotation has been replaced by another field that holds a pointer to the secure coprocessor memory.

```
// access flags 2
private I counter
@Lcom/validy/technology/annotation/SecureField;() // invisible
```

is transformed into

```
// access flags 4114
private final Lcom/validy/technology/runtime/Memory; k$This
```

The `inc` method increments the counter field by one. The instruction `IADD` performs this operation in the original bytecode below.

```
// access flags 1
public inc()V
@Lcom/validy/technology/annotation/SecureMethod;() // invisible
  ALOAD 0
  DUP
  GETFIELD Minimal.counter : I
  ICONST_1
  IADD
  PUTFIELD Minimal.counter : I
  RETURN
```

In the transformed bytecode, the instructions that manipulate the *counter* field have been converted to be executed by the secure coprocessor virtual machine. In the bytecode below, these instructions appear as calls to a method from the Validy Technology runtime that receive a long constant - the ciphered instruction - as parameter.

```
// access flags 1
public inc()V
@Lcom/validy/technology/annotation/SecureMethod;() // invisible
LDC -1871519230529205310
INVOKESTATIC com/validy/technology/runtime/Token.exe (J)V
LDC 5960922510504744152
INVOKESTATIC com/validy/technology/runtime/Token.exe (J)V
LDC 2237779730922682514
INVOKESTATIC com/validy/technology/runtime/Token.exe (J)V
LDC -2778962704774231912
INVOKESTATIC com/validy/technology/runtime/Token.exe (J)V
LDC 8090141189977720929
INVOKESTATIC com/validy/technology/runtime/Token.exe (J)V
LDC 8491204469737689782
INVOKESTATIC com/validy/technology/runtime/Token.exe (J)V
RETURN
```

Because the minimal application is very small, all instructions have been selected to be executed by the secure coprocessor and the size of the class file more than doubles. In a more complex application, only a relatively small fraction of the instructions will be converted.

2.4 Running the protected application

To run, the protected application needs to have access to a secure coprocessor that will be able to decipher and execute the ciphered instructions it now contains. For this first run, we will use a virtual coprocessor that implements the virtual machine entirely in software. Its jar file, `vldy-tech-vruntime.jar` must be placed on the class path.

```
java -cp ./classes;<install path>/lib/vldy-tech-vruntime.jar Minimal 15
```

To use the actual secure coprocessor, the runtime that communicates with a hardware token instead of simulating it must be used. The jar file `vldy-tech-vruntime.jar` must be replaced by `vldy-tech-runtime.jar` on the class path.

2.5 Looking at the communication

The virtual coprocessor can be configured to log the values and instructions sent by the protected application to the coprocessor and the values returned by the coprocessor. With the `logging.properties` file located under `examples`, values sent by the application to the coprocessor and by the coprocessor to the application are printed to standard error: They are prefixed respectively by `>` and `<`.

```
java -cp ./classes;<install path>/lib/vldy-tech-vruntime.jar -Djava.util.logging.config.file=logging.properties
Minimal 7
```

If the `ConsoleHandle` level is changed from `FINE` to `FINER`, the ciphered operations are also printed. They are prefixed by `X`.

Chapter 3

Overview

Validy technology is described more precisely in several documents and presentations available at [Validy's web site](#). This chapter gives a short overview of the technology to understand the work performed by the translator.

3.1 Subtractive Protection

The first characteristic of Validy technology is that the application is protected by subtraction. The coprocessor is not used to check whether the application is authorized to execute. Instead a subset of the application data is stored in the token memory and a subset of the instructions is executed by the token processor. Since the token is secure, the data it contains and the instructions it executes are not observable so they are virtually removed from the original application.

3.1.1 Data

The protection of an application with Validy Technology begins with the relocation of some of the instance and static fields of some of its classes inside the memory of the secure coprocessor. When such a transformed class is loaded or such a transformed object is created, a chunk of memory is allocated in the heap of the coprocessor to hold the values of the relocated fields. In the transformed classes, the relocated instance and static fields are replaced by the address of these chunks of memory. These addresses are stored in two fields named *k\$this* and *k\$class* for instance and static fields respectively. Figure 3.1 shows an object of class A and its fields before and after the relocation of two fields to the coprocessor.

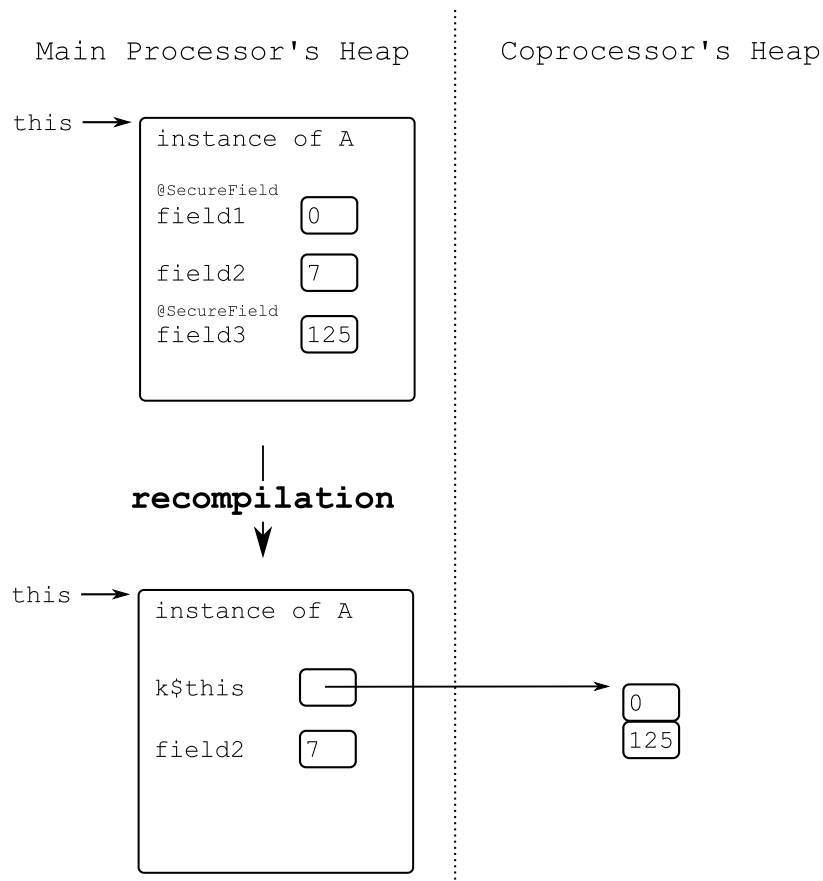


Figure 3.1: Relocation of fields to the coprocessor

When the value of a relocated instance field must be read or written by the application, the value of `k$this` is first transmitted to the coprocessor and then used as the address operand in a load or a store instruction. If `field3` in an object of class A was originally initialized by the following bytecode:

```
ALOAD0
LDC 125
STFLD A.field3 I
```

The transformed pseudo bytecode will look like:

```
ALOAD0
LDFLD A.k$this I
OUT
VM_LDI R1 125
VM_STORE R1 R0 1
```

where `OUT` is an instruction that transfers the top of the Java stack to register `R0` in the coprocessor and `VM_LDI` and `VM_STORE` are coprocessor instructions¹. The 1 at the end of the store instruction is the offset of `field3` from the value of `k$this` stored in `R0`.

Note

While the Java virtual machine is stack based, the coprocessor uses registers. The conversion between the two models is performed by the translator.

¹the way these instructions are represented in the final Java bytecode will be explained later on.

3.1.2 Instructions

Simply sending the value from the main processor to the coprocessor before each store and retrieving it after each load would not provide any protection to the application. Instead, the translator selects instructions around all the accesses to secured fields and transforms the bytecode so that these instructions are executed by the coprocessor. The virtual machine of the coprocessor is not limited to loading and storing field values to and from its registers. Its instruction set can be divided into the following categories:

- Load, store, and move. The values can be immediate, stored in the heap, or on the stack .
- Push to and pop from the coprocessor stack.
- Binary operations such as arithmetic operations, bitwise logical operations, and comparisons.
- Cryptographic operations.
- Security support operations.

One important difference between this instruction set and the one of the Java VM is the lack of control flow instructions. The coprocessor does not maintain a program counter that could be changed to implement control flow². Instead, the responsibility for control flow rests completely on the main processor. Each thread of the application executing in the Java VM produces a stream of coprocessor instructions. These streams are serialized, transmitted to the coprocessor and executed in the order they are received.

The instructions that are executed by the coprocessor instead of the main processor are automatically selected by the translator. The first step in the selection builds a data dependency graph of the instructions of a method. In this graph, instructions that are supported by the coprocessor instruction set and that are located less than a given distance from instructions that load or store the value of a secured field are selected. Straddling values, that is values that are computed by an instruction of the main processor (respectively the coprocessor) but used by an instruction of the coprocessor (respectively the main processor) must be exchanged between the two processors. The translator makes sure this is the case by inserting `out` (respectively `in`) instructions at the right locations. An `out` instruction transfers the value on the top of the Java VM stack to register `R0` of the coprocessor. An `in` instruction transfers the value in register `R0` of the coprocessor to the top of the Java VM stack.

After this step, the application is a mix of Java and coprocessor VM instructions with `out` and `in` instructions interspersed. When the translator generates Java bytecode back from its intermediate representation, coprocessor instruction words are ciphered using a secret key and replaced by calls to a runtime method that transfer these enciphered words to the coprocessor. This implies that the meaning of coprocessor instructions cannot be found by simply looking at the bytecode.

3.2 Detection and Coercion

While the individual instructions executed by the secure coprocessor are ciphered, they perform elementary operations. A cracker may try to understand what a given instruction does by executing it in isolation, or by removing it from the flow sent to the token and analyzing the differences in the output (the values returned by the `in` instructions).

²the coprocessor does support a conditional move instruction that can be used to implement some control flow by executing the two branches of an `if` statement but keeping only the result of one of the branches. However, the translator does not use it to transform the original control flow yet.

By giving to the coprocessor means to detect and react to manipulated streams of instructions, that is streams that are not produced by the normal execution of the application, Validy Technology is able to thwart this kind of attack.

The basic technique used to accomplish this involves two steps:

1. At protection time, the translator computes data dependencies between instructions and encodes assertions about these dependencies inside the instructions
2. At runtime, the coprocessor checks the assertions and can be configured to react to a failed assertion in different ways (from resetting its transient state to permanently erasing its secret key)

The encoding of data dependencies assertions is based on the notion of tag.

3.2.1 Setting Tags

A tag is a random value associated with each coprocessor instruction by the translator. It can be thought of as an identifier but since the current implementation uses byte valued tags, it is not a unique identifier except maybe for very simple applications. When an instruction is executed, its tag is stored along with the result value in the destination register. In the case of instructions without a destination register, the tag is stored in an alternate location, in the heap for the store instruction, or on the stack for the push instruction. Space is reserved to store a tag for each memory location in the coprocessor (registers, heap, stack) and every time the virtual machine stores the result of an instruction, it automatically stores its tag too.

3.2.2 Checking Tags

At each location where the value in a register is used by an instruction, the translator identifies the instruction(s) that (may) have computed this value using a well known data flow analysis called reaching definitions. The list of the tags for these instructions is encoded by the translator so that the coprocessor can check that one of the expected tags is present in the register before proceeding.

The coprocessor instruction words are 64 bit long. Opcodes, register numbers, and tags are byte values³ so the typical binary operation has room for three expected tag values. Two are used for the left operand and one for the right operand. If the reaching definition analysis shows that the value in a source register may have been computed by more instructions than that, special join instructions are inserted⁴ to check tags and reduce the number of tags reaching the binary operation.

The sample code below mixes Java control flow with coprocessor VM instructions. Tags are shown in square brackets. The value before the opcode is the instruction tag and those after each source register are the expected tags:

```
[162] VM_ADD R3 R2 [244, 153] R1 [181]
if (predicate) {
[12] VM_LDI R4 1
} else {
[120] VM_LDI R4 -1
}
[124] VM_ADD R2 R4 [12, 120] R3 [162]
```

There are two valid streams of VM instructions for this fragment of code, depending on the boolean value of the predicate: (162, 12, 124), or (162, 120, 124). If any of the instructions 162, 12, 120 is removed, it is highly unlikely that the tags for R3 and R4 will match what instruction 124 expects.

³these figures correspond to the current implementation and are given only to simplify the exposition.

⁴the special instructions are inserted where a compiler building the Single Static Assignment form of the code would insert ϕ -nodes.

Another advantage of tags is that a good part of the instruction word is now random so that the probability that two instructions that perform the same operation on the same source and destination registers have the same instruction word is extremely low. After they are ciphered, virtually all instructions are different, whether they are semantically the same or not.

3.2.3 Linking through Tags

With tag checking, it becomes possible to link two otherwise unrelated computations by adding a special instruction called a mutual check. This instruction changes the tags of two registers provided they both have the expected tags on entry. If such an instruction is inserted at the end of two computations to link the registers that hold their respective results, it is then impossible to remove or tamper with one computation without losing the result of the other one.

3.2.4 Call protection

Using mutual checks, it is possible to add new secured instructions to the application that cannot be removed because they are linked to one of the original secured instructions. These instructions can be used to perform integrity checks while the program is running. Call protection is one kind of integrity check that can be implemented semi automatically by the translator. It consists in performing a handshake between the call site of a method and the called method. It can be pictured as :

- An extra hidden parameter passed by the caller and whose tag is checked by the callee,
- An extra hidden value returned by the callee and whose tag is checked by the caller.

These extra values are passed on the coprocessor stack or in one of its registers. The value themselves are meaningless⁵, only the tags are important. If the handshake instructions are linked to other secured instructions inside the caller and the callee, it becomes impossible to remove an existing call or add a new call to the method.

Because Java supports virtual method calls and interfaces, the case where a single method implementation is secured by itself is rare. In practice, call protection applies to groups of methods and call sites connected by the following relationships:

- Two methods belong to the same group if they can be called from at least one common call site.
- Two call sites belong to the same group if there exists one method implementation that can be called from both sites.

In Java, all the methods in a group share a common signature and name. For each group, the translator derives two sets of tags, the caller set and the callee set, from this common signature, name, and the secret key used to cipher instructions. Then code is inserted in the calling methods to :

- Choose one tag from the caller set at random and set it before the call.
- Check for any tag from the callee set after the call.

And code is inserted in the called methods to:

- Check for any tag from the caller set on entry.
- Choose one tag from the callee set at random and set it before returning.

The callee tag is not set nor checked if the called method raises an exception because this would require the generation of a try/catch block at each secured call site.

⁵but the translator can use the parameter if it needs to pass an actual value

3.2.5 Other applications

To be completed.

3.3 Runtime

The Validy technology runtime handles the communication between the transformed Java application running inside the host VM and the token VM running inside the secure coprocessor. Its implementation is split into a Java library on the host side and part of the firmware on the coprocessor side. The high level functions let the application perform the following operations:

- Send data to the token (`out`).
- Send ciphered instructions to the token for execution (`exe`).
- Retrieve data computed by the token (`in`).

The translator inserts calls to these functions into the bytecode of the application automatically as it transforms it. The transformed application thus depends on the Java part of the runtime library.

The runtime performs the following actions to ensure that the execution of the protected application is faithful to that of the original one and as efficient as possible.

- The runtime serializes instructions from different threads to respect the [Java memory model](#).
- The runtime buffers the flow of data and ciphered instructions.
- The runtime caches recently executed instructions to decrease the cost of transmitting them several times to the coprocessor.

Tampering with the runtime to modify the dataflow to the token is not of any help to an attacker, if the program remains semantically correct it continues to work, but if it is modified, the token notices it very rapidly and retaliates.

3.4 Virtual Coprocessor

A virtual coprocessor is provided to allow the evaluation of Validy Technology in a software only environment. It is a pure Java implementation of the combination of the runtime and the coprocessor. For the transformed application, it performs exactly the same functions as the secure coprocessor except it *does not provide any actual security* since the instructions it executes and the state it maintains are accessible in clear on the host processor. To appreciate the security of the solution, one must consider that the virtual coprocessor is a black box and that an attacker has only access to the flow of data and ciphered instructions exchanged between the host and the secure coprocessor.

Beyond evaluation, the virtual coprocessor can be used to run automated tests of the protected software without having to handle a physical token.

Chapter 4

Usage

The translator is designed to be easily integrated in the normal Java development process. Modifications must be made at three steps in the process.

1. When writing the source files, select and annotate some fields and methods (or list them in a text file).
2. After the Java source files have been compiled to class files (and possibly put into a jar file), run the translator.
3. Test the transformed application; a virtual coprocessor is provided so that automated tests can be run without having to plug a hardware coprocessor.

4.1 Class Domains

Given the nature of the transformations performed by the translator, several constraints must be obeyed to ensure that the transformed code is consistent. For example:

- if a class A has at least one secured field, all the classes derived directly or indirectly from A must be modified
- if a method m is call secured, all the sites from which m could be called must be modified

Therefore, when transforming an application, the translator must be able to check whether it is allowed to modify a given class or not. Since a Java application can execute the bytecode from a collection of jar files, directories, or other sources using specialized class loaders and since the translator processes only a single jar or directory at a time, the definition of that input jar or directory is not sufficient. To address this problem, the translator defines three domains:

application

the classes of the application.

library

the classes in thirdparty libraries used by the application.

system

the classes from the Java runtime (found by default in the boot class path and the extension directories).

The translator accepts three options to specify three associated class paths: the application path, the library path, and the system path. When searching for a class, the translator searches the application path first, then the library path, and finally the system path. If the class is found, its domain corresponds to the path in which it was found. This search

order is different from the one used by the Java virtual machine. When an application is executed, the boot class path and the extension directories are searched first. However, if an application redefines a class of the Java runtime, the boot class path must be modified for the new definition to be picked up and searching the application path first at transformation time is appropriate.

Warning



To verify the constraints mentioned above, the translator must examine the `extends` and `implements` relationships between classes and interfaces in order to build the class and interface hierarchies and the virtual method tables. To shorten this process by limiting it to application classes, the translator assumes that system and library classes do not depend on application classes. If this assumption is not verified in practice, incorrect code might be generated.

The translator up to version 1.0.5 was limited to the definition of a single path, the class path (option `-c` for the translator or `pclasspath` for the Ant task). The use of the class path is now deprecated. The option can still be used and now sets the library path but the use of the application, library, and system paths is preferred.

In the present version, the translator actually only cares about application classes vs. non application classes. However, defining the system and library paths correctly is advised, should future versions of the translator use them.

4.2 Selecting and Marking Fields and Methods

The selection of the fields stored inside the token and the call secured methods is an important step in the protection of an application. It is currently left to the developer of the application¹ and we provide some guidance in this choice below.

4.2.1 Fields

Selecting a field for storage in the secure coprocessor causes the translator to perform the following operations:

1. The translator replaces the field by a pointer and inserts code in the constructor to allocate space in the memory of the coprocessor to hold the value of the field and its associated tag.
2. The translator transforms instructions located around the accesses to the field so that they are executed by the coprocessor.
3. The translator inserts instructions to exchange values between the main processor and the coprocessor where needed.

The transformations have two consequences on the application. On one hand, the application becomes potentially more difficult to crack. More information about its state and the operations it performs is hidden from a potential cracker. On the other hand, the application takes a performance hit. The coprocessor is slower than the main processor and the communication bus introduces latency when the main processor must retrieve a value from the coprocessor. A good selection must balance these two conflicting consequences. Below are some characteristics of fields that are good candidates for storage in the coprocessor.

A field that has a long lifetime relative to the runtime of the application and that is modified and used several times over this lifetime is a good candidate. The analysis of the values returned by the coprocessor is more difficult if set and get operations are dissociated and performed over a long period.

¹it is expected that future improved versions of the tools will be able to suggest or automatically select good candidates by analyzing the application.

A field that contains a configuration value or a state variable of the application is a good candidate. Such a field is often assigned constant values that need not be transmitted at runtime. Instead the constants are encoded inside ciphered instructions at recompilation time and the instructions that set the value of the field are completely indistinguishable from other ciphered instructions. The change is not linked to the transmission of a given value to the coprocessor but to the execution of a given branch of the code. The value of such a field is also often used to drive the flow of control of the application. When a conditional branch uses the value of the field in a comparison, the translator relocates the comparison inside the coprocessor and its boolean result is the only information that is returned to the main processor.

A field that triggers the execution of heavy numerical operations in the coprocessor is likely to cause performance problems. However if a computation takes some parameters that can be precomputed from configuration values, these parameters and the code that precomputes them can be relocated in the coprocessor. The precomputed values are then retrieved and used in the heavy computation that is performed by the main processor.

There is no tool yet to evaluate automatically the quality of protection afforded by a given choice of secured fields. One key point must be kept in mind when trying to estimate this quality: The result of all the information hiding performed by the translator is embodied in the streams of values flowing from the coprocessor to the main processor as the application is executed. These streams are the values returned by the calls to the `in` method of the runtime. There is one stream for each thread of execution making such calls. A given selection of fields can provide a weak protection in two different cases:

- A cracker can guess the future values in the streams by analyzing the streams produced by the application execution up to a given point in time. This indicates that the hidden part of the application is not complex enough. The log of the virtual coprocessor can be used to extract the streams of `in` values. The log record of a call to `in` is prefixed with the `<` character. It contains the retrieved value and the id of the thread in square brackets. The code of the ciphered instruction executed right before the call to `in` can be used to further classify the streams by `in` call site.
- Altering the stream does not make the application unusable. This indicates that the fields hidden inside the coprocessor are not significant enough to the behavior of the application.

In the evaluation version of the Validy SoftNaos translator and virtual coprocessor package, only *scalar* fields of type *byte*, *short*, or *int* and Java 5 *enumerated types* can be stored inside the coprocessor. We plan to add support for the following types in the next versions of the tool.

- Boolean, and possibly char types.
- Long integers.
- Arrays of the preceding types.

When a field of an enumerated type is stored inside the coprocessor, the value that is actually stored is the ordinal of the enum constant. The ordinal is the integer rank of the constant among all the constants defined for the enum type. If the field is null, a negative integer is stored in place of the ordinal. When an enum field is stored inside the coprocessor, the following operations are performed inside the token:

- assignment of a constant or another enum local or field to the secured field
- comparison of the secured field with a constant or another enum local or field

If the enum is used as the key in a switch statement, the ordinal is retrieved from the coprocessor and used in place of the constant. For operations that require access to the enum constant object itself, such as displaying the string representation of the constant, the ordinal is retrieved and the corresponding enum constant instance is used by the translator.



Warning

To simplify the association between storage locations and tags inside the coprocessor, the minimal unit of memory allocation is the 32 bit integer. A field of type byte or short stored inside the coprocessor occupies the same space as a field of type int.



Warning

One provision of the specification for the Java language in the binary compatibility chapter raises several difficulties for the translator. §13.4.26 *Evolution of Enums* states that ‘Adding or reordering constants from an enum type will not break compatibility with pre-existing binaries.’ However, because the actual value stored inside the coprocessor is the ordinal of the enum, the constraint cannot be maintained and a class that contains a secured enumerated field must be compiled and run through the translator when enum constants are added, reordered or removed.

Another consequence of the same provision is that when an enum type is compiled, the Java compiler must synthesize special code to handle the mapping between constants and their ordinal. This code is not part of the specification of the Java language or the Java virtual machine and is thus compiler dependent. At this time, the Validy Technology translator supports and has been tested with bytecodes generated by Sun’s `javac` compiler (version 1.5.x and 1.6.x) and by the Eclipse internal compiler (version 3.2.x). Please contact Validy if you need support for another compiler.

4.2.2 Methods

The current implementation of the translator does not automatically select methods for call protection. The developer must mark methods with the `SecureMethod` annotation or list them in a separate text file (see option **markers**).

Note

There is no need to mark the call sites. They are identified automatically by the translator.

Because methods are secured in groups (see Section 3.2.4), the methods that can be selected must satisfy a few constraints.

- All the call sites must be transformed by the translator; this rules out interfaces or virtual methods defined outside the application because all the call sites may not be accessible to the translator².
- Methods that are called directly by the virtual machine cannot be secured; this includes class constructors, `finalize` methods, `writeObject/readObject` methods used for serialization.
- Methods called using reflection cannot be secured.

A slightly different constraint addresses how (rather than which) method groups should be marked so that all members of the group are identified by the translator. A method should always be marked in the highest³ class or interfaces in which it is defined. Consider the following counterexample:

```
class A {  
  
    public abstract void f();  
}
```

²unless you are willing and authorized to distribute modified copies of third party libraries

³in the extends or implements relationship

```
}  
  
class B extends A {  
  
    @SecureMethod  
    public void f () {  
    }  
  
}  
  
class C extends A {  
  
    public void f () {  
    }  
  
}
```

If C is processed before or independently from B, C.f will not be treated as secured and the application will fail if C.f is called through a reference to A. The same problem occurs if A is defined as an interface and the `extends` keywords are replaced by `implements` keywords.

Warning

Some method groups require several marks in order to be properly identified by the translator. In the example below, `A.f` and `D.f` belong to the same group because:

- Method `A.f` and `B.f` can be called from the same site (`invokeinterface I.f`) since `B.f` is the implementation of `I.f` for class `C`.
- Method `B.f` and `D.f` can be called from the same site (`invokevirtual B.f`).

Both `SecureMethod` annotations below are required.

```
interface I {  
  
    @SecureMethod  
    void f();  
  
}  
  
class A implements I {  
  
    public void f() {  
    }  
  
}  
  
class B {  
  
    @SecureMethod  
    public void f() {  
    }  
  
}  
  
class C extends B implements I {  
  
    public void f() {  
    }  
  
}  
  
class D extends A {  
  
    public void f() {  
    }  
  
}
```



Within the constraints exposed above, the only limit to the number of secured method calls is performance degradation. However, secured calls have less impact on performance than secured fields for the following reasons:

- Secured calls do not allocate extra memory in the coprocessor.
 - Secured calls do not retrieve any value produced by the coprocessor. Their only potential effect is a failure of the application if it is tampered with.
-

4.2.3 Call Graphs

The selection of methods one at a time for call protection can be tedious. A call graph builder was developed to simplify the process. This tool computes call graphs starting from entry point methods, checks that the call graph will be linked to the original secured code, and provides the list of methods that have to be secured. This is a standalone tool but its output can be used directly as an input for the translator and the provided Ant tasks allow easy integration.

Using the secure call graph builder has the following advantages over manual selection of methods to secure:

multiplication

by selecting only one entry point, many methods are secured

connexity

at runtime, the selected methods automatically form a secured call stack rooted at the entry point

security

by default, the tool outputs only the methods of the call graphs that are actually linked to some of the original secured code. If they were not, all the code added to freeze the graph could be removed in one block without affecting the application. For a graph to be secure, the code of at least one of its methods must access a secured field, allocate an object with a secured field, or make a call to a secured API. A call graph that shares methods with another secure call graph is also secure.

The call graphs computed by the builder are geared to their use for call protection. Compared to complete Java call graphs, they have the following limitations:

- Only explicit method calls are explored. Implicit calls such as calls to class initializers are not considered because they cannot be secured. The calling context is not completely under control of the translator so that a proper secured call context cannot be created.
- Only methods in application classes are explored. Building the call graph further is unnecessary since the bytecode outside of application classes cannot or are not allowed to be modified to perform secured calls.
- No type analysis is performed to prune the method implementations that cannot actually be called from a given site. This analysis could produce a more precise call graph but for call protection, all the methods that share at least one common call site must be secured in one group. Proving that a method implementation cannot be called from a given site is not enough. The correct verification would require a global analysis at the application level to show that the implementation is never called.

4.2.4 Annotations

The `com.validy.technology.annotation` package defines three annotation classes, `SecureField`, `SecureMethod` and `SecureCallGraph`. To use these annotations, you must add the `vldy-tech-annot.jar` file to your class path. The last annotation is used only by the call graph builder, the translator simply ignores it and removes it from the bytecode it transforms.

The `SecureField` annotation takes one optional parameter called `registerNumber`. Instance fields have as many copies as there are objects of the class they belong to. They can only be allocated in the heap of the coprocessor and they are accessed using an offset from the `k$this` pointer. Static fields on the other hand are unique and can be allocated in one of two ways: in the heap or in a global register. The allocation of static fields in the heap is similar to that of an instance field. A chunk of coprocessor memory is allocated in the class constructor and stored in a static pointer called `k$class`. This is the default behavior. When a register number is specified, memory is not allocated and the value of the field is stored permanently in the given global register.

Storing a static field in a register has the following advantages:

- Accesses to the field are slightly more efficient because stores and loads are not needed.
- If all secured static fields of a class are put into registers, `k$class` is not defined. There is not hint that the class has secured static fields.

The characteristics of the coprocessor limit the number of static fields that can be allocated in global registers. The virtual coprocessor has 16 global registers numbered from 32 to 47. With the annotation below, the `counter` field of class `A` is allocated to register 40.

```
class A {

@SecureField(registerNumber=40)
private static int counter = 0;

...

}
```

The `SecureField`, `SecureMethod`, and `SecureCallGraph` annotations accept an optional boolean parameter called `enabled` whose default value is `true`. When this parameter is set to `false`, it instructs the translator *not* to secure the given field or method, or to output a list of methods *not* to secure in the case of the call graph.

The need to annotate an object to avoid protection seems paradoxical since objects that are not annotated at all are not secured. However using markers (see Section 4.2.5), it is possible to use automated tools to produce lists of fields or methods selected for protection and thus the same object may have several annotations. Adding a disabled annotation is useful when an automatically selected object must not be changed by the translator for a reason not taken into account by the tool. If an object has at least one disabled annotation, the translator will not change it.

For example, if the method `g` from class `A` is selected by a tool that automates call protection but must remain unchanged because it is called through reflection in some part of the application, the following annotation makes sure the translator does not apply call protection to it:

```
class A {

@SecureMethod(enabled=false)
public int g() {
...
}

...

}
```

4.2.5 Markers

This is an alternative method to using annotations and the only possible method for Java 1.4. It can also be used to accept the output of a tool that selects fields or methods automatically for protection.

To mark a field or a method, its containing class or interface name, its name and its signature must be listed on one line separated by spaces. The class name and the signature must follow the form defined in the [Java Virtual Machine specification](#). Lines starting with a `+` character correspond to enabled annotations while lines starting with a `-` character correspond to disabled ones. If the `+` or `-` character is omitted, `+` is assumed by default.

Lines starting with a `#` character are ignored and can be used to add comments.

For example, to produce the same result as the following annotations:

```
package com.validy.sample;

import com.validy.technology.annotation.*;

public class Test {

    @SecureField
    private int value;

    @SecureMethod
    public final void f(String name);

    @SecureMethod(enabled=false)
    public int g();

}
```

the following lines should be added to the markers file:

```
+ com/validy/sample/Test value I
+ com/validy/sample/Test f (Ljava/lang/String;)V
# g is called through reflection in ...
- com/validy/sample/Test g ()I
```

4.2.6 Object Serialization

4.2.6.1 Introduction

Serialization is a way of flattening the state of objects to produce a data stream. This stream can then be stored on disk or transmitted over the network and later parsed to reconstruct a copy of the original objects. The Java language's builtin support for serialization is described in the [Java Object Serialization Specification](#). Even though it can be customized by implementing special methods (`writeObject` and `readObject`), the support for serialization is largely declarative in nature:

- classes whose instances need to be serialized are simply marked by having them inherit from the `java.io.Serializable` interface,
- fields in these classes that should not be serialized are marked with the `transient` modifier.

The Java runtime is responsible for the actual process of serializing and deserializing object instances. It works by inspecting the definition of classes at runtime using reflection to decide what should be put in the output stream and what should be done with the content of the input stream. The declarative nature of Java serialization makes it easy to use even if its interaction with class evolution requires careful planning. The Validy SoftNaos translator strives to retain this simplicity. No extra work is required to support the serialization of classes with secured fields or methods. However in the absence of runtime support, the translator must use the customization hooks provided by the Java language to implement the secure serialization and deserialization of fields stored inside the token. The rest of this section presents this implementation and its limitations. It assumes some familiarity with Java object serialization.

4.2.6.2 Default Behavior

When one or more fields are marked for storage inside the secure token, the translator removes their definition from the class and they are replaced by a pointer to token memory named `k$this`⁴. The Java runtime does not have enough information to be able to serialize or deserialize the object. Therefore, the translator must provide special support for serialization by:

- overriding the default definition of persistent fields through the declaration of a custom `serialPersistentFields` (or the modification of an existing declaration),
- implementing custom `writeObject` and `readObject` methods (or modifying existing implementations).

When a secured field is serialized, its value is first loaded from the token memory to a token register, concatenated with a random nonce, checksummed, ciphered, and the resulting block is retrieved from the token to be put in the output stream. By default, when a field is deserialized, the ciphered block is sent to the token, deciphered, the checksum is verified, the nonce discarded, and if the value has not been tampered with, it is stored in the token memory at the proper location. The operations performed on the value of a field before serialization (nonce, checksum, cipher) are added to:

- hide the actual value of the field when it is retrieved and stored outside of the token,
- make it impossible to know whether a field has changed value or not between two serializations of the same object,
- make it difficult for an attacker to tamper with the serialized value to try to alter the behavior of the secured application.

These operations use instructions from the token virtual machine and benefit from the same protection as other instructions generated by the translator (linked using tags and ciphered). The key used to cipher and decipher serialized values is stored inside the token at customization time and is independent from the one used to decipher instructions. Because of the nonce and checksum, a byte, short or int value that occupies one 32 bit word in the memory of the token is serialized as an 8 byte array.

4.2.6.3 Backward Compatibility

By default, streams of objects that were produced before the application was transformed and contain the value of secured fields in clear can be read but the secured fields are not initialized and trying to access them later on triggers an error in the secure token. There are two ways to handle this problem:

1. a custom `readObject` method can be defined that initializes the secured fields with a default value before calling `defaultReadObject`.
2. an option can be passed **on the command line** or as an **attribute** of the Ant task to change the default behavior. When this option is specified, the implementation of `readObject` is altered to accept either ciphered blocks or clear values for secured fields in the input stream. While ciphered blocks are handled as described above, clear values are transmitted to the token as is and stored directly in memory. This makes it possible to deserialize objects that were serialized before the application was transformed by the translator at the cost of extra storage⁵.

⁴because it references an external resource (the secure token heap), `k$this` is marked with the transient modifier. The translator generates the code that recreates it upon deserialization.

⁵both the clear and the ciphered value of each secured field must be declared in `serialPersistentFields` for the application to be able to retrieve either the clear or the ciphered value of the field. Because of the way the Java runtime handles this declaration, both values will be stored in the output stream when an object is serialized by the transformed application. The clear value of the field will always be zero and the ciphered block will contain the correct value.



Warning

When this option is used, the protection against tampering becomes ineffective since an attacker can always remove the ciphered block and set the clear value in the serialized stream before it is deserialized.

4.2.6.4 Caveats

When a field declared in a non serializable class is secured, the translator removes the declaration of this field from the bytecode of the application and all accesses to the field are performed by ciphered instructions of the virtual machine. The only remaining information is the presence of a *k\$this* field that indicates the presence of at least one secured field in the class. However if the class is serializable, the name and type of the field are declared in the *serialPersistentFields* field and appear in the class constructor's bytecode.

When custom `readObject` and `writeObject` methods are defined, their special handling by the translator depends on locating the call to `defaultReadObject` or `readFields` and `defaultWriteObject` or `putFields` respectively. If one of these methods is not called, a warning is issued, the methods are handled normally, and the fields will be serialized in clear.

The current version of the translator has no support for externalization or secure custom serialization. If an application class implements `java.io.Externalizable` and its implementation of `writeExternal` reads the value of a secured field to stream it, the translator generates code that retrieves the value of the field in clear from the token. A future version of the translator may add support for an assembly-level interface to the virtual machine that would let the developer insert checksumming and ciphering instructions 'by hand' but still have the translator do register allocation, set the right tag checks, and cipher the instructions.

The following missing functionalities may be added in future versions of the translator depending on demand:

- possibility to remove the secure annotation on a field and still be able to read streams where its value was stored as a ciphered block,
- control of code generation for backward compatibility at the class or field level using annotations,

4.3 Running the Tools

4.3.1 Translator

4.3.1.1 Command Line

To run the Java bytecode translator, just type:

```
java -jar vldy-tech-translator.jar [options] [class names]
```

It is also possible to put options and class names inside a text file and pass it to the translator using the @ character:

```
java -jar vldy-tech-translator.jar @optionfile.txt
```

The input classes can be selected by specifying either a base directory or a jar file. The output classes are written to either an output directory or a jar file. However, the combination of an input directory and an output jar file is not supported, please use the **jar** tool to package the classes after transformation instead. By default, all the classes found under the given directory or in the given jar file are processed but it is possible to restrict the transformation by adding the names of the classes to be transformed at the end of the command line, after all other options. The

files or resources that are not class files and the class files that are not transformed are copied unchanged to the output directory or jar file.

The translator must also be able to find the bytecode of all the classes used or referenced by the input classes. The class paths used by the translator to lookup classes is specified using option `-a` (`--applicationpath`), `-l` (`--librarypath`), and `-s` (`--systempath`). They are *not* the class paths used by the Java virtual machine to run the translator itself. By default, the application and library paths are empty and the system path is initialized using the `sun.boot.class.path` and `java.ext.dirs` system properties.

The translator accepts the following options:

-i or --input *directory_name* or *jar_file_name*

Specifies the directory or jar file containing the classes to be transformed.

-o or --output *directory_name* or *jar_file_name*

Specifies the directory or jar file where the transformed classes should be stored. The argument can be a jar file only if the input itself is a jar file. The translator cannot be used to create jars.

-1

Specifies that the translator must target version 1 of the token instruction set.

-2

Specifies that the translator must target version 2 of the token instruction set.

-a or --applicationpath *class_path*

Specifies the path to the application classes. The input jar or directory is automatically prepended to the application path.

-l or --librarypath *class_path*

Specifies the path to the library classes referenced by the application classes.

-s or --systempath *class_path*

Specifies the path to the Java runtime classes referenced by the application or library classes. This makes it possible to protect an application that relies on a runtime different from the one used to run the translator itself (Java 5 or later). If this option is not specified, the system path defaults to the path defined by the `sun.boot.class.path` system property and the jar files found in the directories defined by the `java.ext.dirs` system property.

-c or --classpath *class_path* [*deprecated*]

Specifies the path the translator should use to lookup classes referenced by the input classes. This option is deprecated and is now an alias for option `--librarypath`, please specify the application, library, and system paths instead.

-m or --markers *file_name*

Specifies the name of a file containing a list of fields and methods to protect (or not to protect). This option can be used several times if there are several markers files.

-x or --disassemble *directory_name*

Instructs the translator to save a disassembled copy of the class files it processes before and after transformation. These files are produced using ASM's TraceVisitor.

-h or --help

Lists the translator options with a short description.

-v or --verbose *level*

Instructs the translator to output progress messages.


```
<path id="validy.translator.path">
  <path element location="{vldy.home}/vldy-tech-translator.jar" />
</path>
```

4.3.1.2.2 Parameters specified as Attributes

The following tables list the supported parameters.

Attribute	Description
injar	The jar file that contains the class files to be transformed. Only one of injar and indir can be specified.
indir	The directory in which to look for classes to be transformed. Only one of injar and indir can be specified.
instructionSet	The version of the token instruction set that must be targeted by the translator. The possible values are v1 or v2. v2 is the default.
marker	The text file containing the list of fields and methods to secure.
outjar	The jar file in which to place the transformed class files. Only one of outjar and destdir can be specified. Must be used with injar.
outdir	The directory in which to place the transformed class files. Only one of outjar and outdir can be specified. Can be used with injar or indir.
disassembledir	The directory in which to place assembly files of the bytecode before and after transformation.
rseed	The integer value of the seed for the random number generator used to pick tags.
key	The key used to cipher instructions. It must be a 48 character long hexadecimal string.
verbose	If <code>true</code> , the translator will output progress messages.
treatwarningaserror	If <code>true</code> , the translator will fail when a warning is encountered.
backwardcompatibleserialization	If <code>true</code> , the translator will generate code that accepts clear values for secured fields in serialization streams.

Table 4.1: Attributes for the vldy task

The vldy task is an extension of the standard `java` task and as such supports many of its parameters. The exceptions are `classname` and `jar` which are hard coded to run the translator.

For the complete list of attributes, please refer to the [documentation of the java Ant task](#).

4.3.1.2.3 Parameters specified as Nested Elements

The following parameters can be specified as nested elements:

Attribute	Description
classpathref	The class path to use, given as reference to a PATH defined elsewhere. This path should contain the <code>vldy-tech-translator.jar</code> file.
fork	If enabled, triggers the execution in another VM (disabled by default).
maxmemory	Max amount of memory to allocate to the forked VM (ignored if fork is disabled)
failonerror	Stop the build process if the command exits with a return code other than 0. Default is "false".
resultproperty	The name of a property in which the return code of the command should be stored. Only of interest if <code>failonerror=false</code> and if <code>fork=true</code> .

Table 4.2: Some attributes inherited from the java task

applicationpath

Use a nested `<applicationpath>` element to specify the path to the application classes. It is a **path** like structure.

librarypath

Use a nested `<librarypath>` element to specify the path to the library classes used or referenced by the application classes. It is a **path** like structure.

systempath

Use a nested `<systempath>` element to specify the path to the Java runtime classes used or referenced by the application or library classes. It is a **path** like structure.

pclasspath

Use a nested `<pclasspath>` element to specify the path used by the translator to lookup classes used or referenced by the input classes (see option **classpath**). This parameter is deprecated, please use `<applicationpath>`, `<librarypath>`, and `<systempath>` instead.

indirset

Use a nested `<indirset>` element as an alternative to the `indir` attribute that can be used to apply a filter to select the classes to transform. It is a **fileset** like structure.

markers

Use a nested `<markers>` element as an alternative to the `markers` attribute when there are several markers files. It is a **patternset** like structure.

The following nested elements are inherited from the standard **java** task:

jvmarg

Use nested `<jvmarg>` elements to specify arguments for the forked VM.

sysproperty

Use nested `<sysproperty>` elements to specify system properties required by the class. These properties will be made available to the VM during the execution of the class (either ANT's VM or the forked VM). The attributes for this element are the same as for **environment variables**.

The translator run shown at the end of section can be replicated as an Ant task that forks the translator as follows:

```
<vldy indir="./build/classes" outdir="./build/classes.secured"  
  markers="./secured.txt" classpathref="validy.translator.path"  
  fork="true" failonerror="true">  
  <systempath>  
    <pathelement location="{java.home}/jre/lib/rt.jar" />  
  </systempath>  
</vldy>
```

4.3.1.2.4 Dependencies

Before running the translator, the vldy task checks its dependencies. If one of the expected output file is missing or is older than any of the input files, the translator is run. All the files listed below are considered as input files to the translator:

- the input jar file, any class file found under the input directory, or any class file listed in the <indirset> nested element,
- the jar files listed in the application path and any class file found under a directory listed in the application path,
- the files containing protection markers.

4.3.2 Secure Call Graph Builder

4.3.2.1 Command Line

The call graph builder tool is stored in the same jar file as the translator. To run it, just type:

```
java -classpath vldy-tech-translator.jar com.validy.technology.hekla.SecureCallGraph [options]
```

It is also possible to put options inside a text file and pass it to the builder tool using the @ character:

```
java -classpath vldy-tech-translator.jar com.validy.technology.hekla.SecureCallGraph @option-file.txt
```

The builder tool does not take a jar or directory as input. All the classes found in the application class path are considered to build the graphs starting from the entry points provided as input. It is important that the graph builder has access to the definition of secured fields to check that a call graph is secure (see option -m).

The call graph builder tool accepts the following options:

-e or --entrypoints *entry_points_file*

Specifies the name of a file containing a list of entry point methods. Each method will be the root of one call graph. This file must have the same format as a markers file (see Section 4.2.5).

-o or --output *markers_file*

Specifies the name of the file where the methods selected by the tool will be listed. This file can then be used as an input markers file for the translator using option -m. If the entry point for a call graph is given on a line starting with a + or a - character, the methods from the graph will be listed starting with the same character. If this option is omitted, the methods are listed on the standard output.

-d or --depth *max_depth*

Specifies the maximum depth of the call graphs that will be built. The maximum depth may not be reached because the computed graphs are constrained to remain within application classes only. The maximum depth is set to 5 by default.

-a or --applicationpath *class_path*

Specifies the path to the application classes.

-l or --librarypath *class_path*

Specifies the path to the library classes referenced by the application classes.

-s or --systempath *class_path*

Specifies the path to the Java runtime classes referenced by the application or library classes. This makes it possible to protect an application that relies on a runtime different from the one used to run the translator itself (Java 5 or later). If this option is not specified, the system path defaults to the path defined by the `sun.boot.class.path` system property and the jar files found in the directories defined by the `java.ext.dirs` system property.

-m or --markers *file_name*

Specifies the name of a file containing a list of fields and methods to protect (or not to protect). This option can be used several times if there are several markers files.

-f or --force

Specifies that the methods of all the computed call graphs should be included in the output. By default, the methods of call graphs that are not linked to original secure code are not listed.

-h or --help

Lists the builder options with a short description.

-v or --verbose

Instructs the builder to output progress messages.

4.3.2.2 Integrating with an Ant build script

The call graph builder tool can also be integrated to an [Ant](#) build script (version 1.6 and higher) using the `vldygraph` custom task provided in `vldy-tech-ant.jar`.

4.3.2.2.1 Configuration

See Section [4.3.1.2.1](#).

4.3.2.2.2 Parameters specified as Attributes

The following table lists the supported parameters.

The `vldygraph` task is an extension of the standard `java` task and as such supports many of its parameters. The exceptions are `classname` and `jar` which are hard coded to run the call graph builder tool. See [Table 4.2](#) for a list of useful attributes.

4.3.2.2.3 Parameters specified as Nested Elements

The following parameters can be specified as nested elements:

applicationpath

Use a nested `<applicationpath>` element to specify the path to the application classes. It is a [path](#) like structure.

Attribute	Description
entrypoints	The file that contains the list of call graph entry points.
output	The file in which to list selected methods.
maxdepth	The maximum depth of the call graph that will be computed.
markers	The jar file in which to place the transformed class files. Only one of outjar and destdir can be specified. Must be used with injar.
force	If <code>true</code> , the methods in the call graphs that are not linked to original secure code are included in the output. If <code>false</code> they are ignored.
verbose	If <code>true</code> , the builder tool will output progress messages.

Table 4.3: Attributes for the vldygraph task

librarypath

Use a nested `<librarypath>` element to specify the path to the library classes used or referenced by the application classes. It is a `path` like structure.

systempath

Use a nested `<systempath>` element to specify the path to the Java runtime classes used or referenced by the application or library classes. It is a `path` like structure.

markers

Use a nested `<markers>` element as an alternative to the markers attribute when there are several markers files. It is a `patternset` like structure.

Other nested elements are inherited from the standard `java` task, see Section 4.3.1.2.3 for more information.

4.4 Running the protected application

The virtual coprocessor is implemented by the `vldy-tech-vruntime.jar` file. It uses the same key as the one used by default by the translator to cipher instructions so simply adding the jar to the class path and running the application should be enough. Data and instructions exchanged between the application and the coprocessor are logged to a `java.util.logging` logger named `com.validy.technology.runtime`. The stream can be captured simply by configuring the logging system. A sample `logging.properties` file can be found in the `examples` directory.

The virtual coprocessor supports both versions of the token instruction set. To choose which version should be simulated, the system property `com.validy.technology.runtime.version` must be set to either 1 or 2. By default, version 2 of the instruction set is simulated.

The interface with the hardware secure coprocessor is implemented by the `vldy-tech-runtime.jar` file. In the current version, Java 6 is required to use this jar. Data and instructions exchanged between the application and the coprocessor are *not* logged by the runtime.

The hardware coprocessor must be plugged in before launching the protected application.

Appendix A

Single user license for the Validy SoftNaos for Java software package, evaluation version 1.0

The terms of this license constitute a contract between Validy Net Inc, public company under the Oregon law (USA), registered on May 26th, 1998, in Portland under number 635381-86, having its head office 1001 SW Fifth Avenue - Suite 1100, Portland, OR 97204, and its secondary office 145455 NW Greenbrier - Suite 210, Beaverton, OR 97006, hereafter referred to as 'VALIDY' and yourself, as a user hereafter referred to as the 'authorized evaluator'

Read these terms carefully. They concern the Validy® Recompiler for Java®, evaluation version 1.0, hereafter referred to as the 'software package', including the media on which you have received it, if applicable.

BY USING THIS 'SOFTWARE PACKAGE', YOU AGREE TO THE TERMS OF THIS CONTRACT. IF YOU DO NOT ACCEPT THOSE TERMS, DO NOT USE THIS 'SOFTWARE PACKAGE'.

Copyright Validy Net Inc, 2007

A.1 Object

The present contract grants you a limited use license of the Validy SoftNaos for Java 'software package', evaluation version 1.0, property of S.A. VALIDY and VALIDY NET Inc. The use rights covered by the present contract are granted to you by S.A. VALIDY as the author and publisher of the 'software package'.

The 'software package' functionalities and conditions of use are described in the user documentation attached to this contract.

A.2 Period of authorization

The right to use the 'software package' is granted for the time needed to conduct the requested evaluation. This period cannot exceed the duration of the intellectual property rights for this 'software package' as enforced by regulations on copyright.

A.3 Scope of license

A.3.1

The object of the present agreement is to authorize the use of the 'software package' by the 'authorized evaluator' in order to perform a technical evaluation of the 'software package' and of the patented VALIDY TECHNOLOGY

solution that it allows to simulate using a virtual token. The ‘software package’ constitutes a simple evaluation tool and without a secure physical token, its use cannot provide an effective protection of the software.

A.3.2

All rights to the ‘software package’ apart from those granted to you for the evaluation are reserved by ‘VALIDY’.

The ‘software package’ is licensed, not sold to you. This contract aims to grant you a limited use license with the sole goal of allowing you to operate a scientific and technical evaluation of the ‘software package’. The present license is therefore granted for the exclusive needs of this evaluation. Any other use be it gratuitous or in return for payment, for civilian or military purposes, for private or professional purpose and for commercial and/or industrial purposes of this ‘software package’ is strictly forbidden.

The ‘authorized evaluator’ refrains himself from carrying out any sort of computing processes or services for himself or a third party by using directly or indirectly this ‘software package’ outside of what is required for the technical evaluation for which this use right is granted.

A.3.3

Unless applicable law gives you more rights despite this limitation, you are authorized to use this ‘software package’ only in accordance with the terms of this agreement. To this end, you must comply with the technical restrictions of the ‘software package’ that allow you to use it only in a certain way.

A.3.4

Outside the exception allowed and defined in sections Section [A.8](#) and Section [A.10](#) of this contract, the ‘authorized evaluator’ strictly refrains himself from reproducing permanently or for a limited time the totality or part of the ‘software package’, by any means and under any form, including during the loading, display, execution, transmission, or storage of the ‘software package’.

A.3.5

The ‘authorized evaluator’ strictly refrains himself from adapting, arranging, or modifying the ‘software package’, from decompiling it, exporting it, or merging it with other softwares.

The present license does not grant directly or indirectly to the ‘authorized evaluator’ access rights to the source code of the ‘software package’.

Considering the final goal of this license and the limited nature of its usage, the ‘authorized evaluator’ refrains himself from any action or operation aimed directly or indirectly at reverse engineering the ‘software package’.

A.3.6

Thus you are not authorized to:

- work around any technical limitations in the ‘software package’;
 - reverse engineer the ‘software package’, decompile it, or disassemble it, except if applicable law explicitly permits it, despite this limitation;
 - publish the ‘software package’ for reproduction by a third party that is not authorized by this contract;
-

- rent or sell the ‘software package’;
- use the ‘software package’ for the commercial promotion of your brands or programs.

A.4 Absence of financial compensation

The authorization to use the ‘software package’ for evaluation is free of charge. It will not bring any payment or license fee of any form. The evaluation is conducted by you and at your expenses.

A.5 Product keys

The ‘software package’ does not require a key to install or use it. You are responsible for the correct and trustful usage of the limited use rights you have been granted.

A.6 Entire agreement

A.6.1

The present contract constitutes the entire agreement regarding the ‘software package’ and the limited use rights granted to it.

A.6.2

Nothing in what is contained in this agreement must be interpreted as creating between ‘VALIDY’ and the ‘authorized evaluator’ and/or his sub-user, any kind of judicial link, contractual commitment, sales promise or money order in the aim of a partnership, cooperation, or a representation, or even as equaling a concession license, even tacitly and temporarily, of use and or commercialization be it industrially or professionally and of any sort of intellectual or industrial property or even as equaling a reservation on the behalf of the ‘authorized evaluator’ of any kind of know how on the ‘software package’.

A.6.3

Nothing in what is contained in this agreement must be interpreted as giving, directly or indirectly, tacitly or explicitly, any right to partially or completely carry out the use and/or exploitation even temporarily, of any of the patented inventions used in VALIDY TECHNOLOGY and whose commercialization belongs solely to S.A. VALIDY and VALIDY NET Inc.

A.6.4

No rights or obligations other than the ones formulated and enumerated in the present limited license agreement will be inferred from this agreement, or deduced from it. The authorized use, for testing purposes will not give, directly or indirectly, to the ‘authorized evaluator’, and/or to his sub-user, any right of use, and/or commercialization of all or part of the patented inventions.

A.6.5

The default of one of the party to assert their right following the breach of any of the provisions of the present agreement by the other party does not constitute a waiver of any right to the said provision or of any other right covered by the terms of this agreement.

A.7 Severability

If any provision of this agreement is deemed to be invalid, illegal or unenforceable, the other provisions will remain fully valid and effective. This agreement can not be amended except in writing and signed by a delegate duly authorized by each party.

A.8 Installation and distribution by your care of the usage rights

A.8.1

The use of the ‘software package’ is granted to you personally. The ‘software package’ must be installed by you for your own use.

A.8.2

You may hand over a copy of the ‘software package’ to a third party as a sub-user authorized by you, in order for him to test and evaluate the ‘software package’ as well as your own programs. The users of your programs will therefore be allowed to use the ‘software package’ to perform some tests on your own programs recompiled with VALIDY.

The copy provided to this end must imperatively contain every item of the ‘software package’ without exception. It will necessarily include the installation file `vldy-softnaos-eval-1.0.x.y.jar` and the file containing the evaluation guide as well as the terms of this license.

A.8.3

The use of the ‘software package’ by the authorized sub-user is subject to the acceptance of all the terms of this license agreement, otherwise the rights of use are immediately and completely removed.

A.8.4

The tests performed by yourself, and by your sub-user can in no way include the installation, even temporarily of the ‘software package’ in a production and/or commercialization environment.

This limited use license does not grant you, directly or indirectly, rights to use the ‘software package’ and/or executable code generated by the ‘VALIDY’ translator tool in a production environment or for commercial, industrial, or professional purposes, whether it be for civilian or military applications.

A.8.5

The ‘software package’ contains and generates code that neither you nor the sub-user you designated, is authorized to distribute in programs meant to be used, either by you, or by third party users, for commercial, industrial or professional purposes.

In no way is the code contained, and/or generated by the ‘software package’ to be used in production, and/or commercialization, for civil or military, private, public, commercial, industrial, or professional purposes.

It belongs to you to always consider a ‘VALIDY’ recompiled software in the frame of the evaluation permitted by the ‘software package’ as test material. This software has an experimental value and is not fit for commercial, industrial or professional use.

A.9 Disclosure of the evaluation results

You are authorized to disclose the results of your evaluation tests of the ‘software package’ provided you comply with the following conditions:

- a. you must disclose the complete set of data necessary to reproduce the test operations, including a detailed description of the methodology followed, the test scripts/cases, the configuration parameters, the software and hardware platforms used, the name and version number of any third party tool used to carry out the test;
- b. you must indicate the date on which the evaluation tests took place, the version of the program tested, and the version of this ‘software package’ with the copyrights included;
- c. your evaluation tests must be done using all the performance adjustments, and the best possible practices;
- d. you may make your results accessible to the general public, on the condition that any public disclosure of your evaluation test results indicates explicitly the place where all the above required information are accessible the general public.

A.10 Backup copies

Apart from the copies authorized for the purpose of Section [A.8](#) of this contract, you are authorized to make a backup copy of the ‘software package’. This copy must imperatively contain all the components of the ‘software package’ without exception, including the installation file `vldy-softnaos-eval-1.0.x.y.jar` and the file containing the evaluation guide as well as the terms of this limited use license.

You can only use the backup copy to reinstall the ‘software package’. The sole purpose of the backup copy is to be used in case of failure of the ‘software package’ handed to the ‘authorized evaluator’.

A.11 Documentation

Any user having a legitimate access to your computer or your company internal network is authorized to copy and use the documentation for the purpose covered by the present license.

A.12 Free transfer to a third party

You have the right to transfer free of charge the ‘software package’ and the present contract, directly to a third party. Before the transfer the third party must acknowledge, or else under the terms defined by this agreement he can not use it, that the points contained in this agreement apply with no restriction to the transfer and to the use of the ‘software package’ by himself.

A.13 Export restrictions

The ‘software package’ distributed by VALIDY Net Inc. is subject to United States export laws and regulations.

You must in any case comply with the domestic and international laws and regulations that apply to the ‘software package’. These regulations may restrict the country of destination, the status of the intermediate and end users but also the end use of the ‘software package’. These restrictions apply to you in full. Any breach of these rules will bring the immediate cancellation, without formality, of the usage rights granted to you by the present agreement.

A.14 Applicable law

Any action related to this Agreement will be governed by Oregon state law and controlling U.S. federal law.

A.15 Limitation and exclusion of responsibility in case of damage

You cannot obtain from ‘VALIDY’ any kind of reparation in case of direct damage, and this because the ‘software package’ was handed freely to you with the sole purpose of operating, at your own initiative, and under your sole responsibility, an strictly scientific and technical evaluation and this outside of any commercialization, production or industrial exploitation perspectives.

Having committed yourself not to use the ‘software package’ for any other purpose than its evaluation under your sole responsibility, you cannot claim any reparation for any kind of damage, be it material or immaterial, including direct or indirect special damage, or even damage linked to some main damage.

A.16 Limited warranty

A.16.1 Limited warranty

If you follow the instructions, the ‘software package’ will operate as expected for evaluation purposes. The ‘software package’ is delivered as is.

A.16.2 Period of warranty

The limited warranty covers the ‘software package’ for six month after the date it was first put at your disposal. During this period, if the user transfers the ‘software package’, the time length left on the warranty applies to the new user.

TO THE EXTENT ALLOWED BY ENFORCEABLE REGULATION, ANY IMPLIED WARRANTY WILL BE ENFORCEABLE ONLY DURING THE TERM OF THE LIMITED WARRANTY.

In the eventuality of the warranty coming into play, 'VALIDY' commits itself to replacing the 'software package' free of charge using a download. It will be up to you to identify yourself to 'VALIDY' and to supply documentary evidence that you are legitimately in possession of the 'software package' in order to benefit from the warranty.

THE LIMITED WARRANTY IS THE ONLY DIRECT WARRANTY GIVEN BY 'VALIDY'. 'VALIDY' DOES NOT HAND ANY OTHER EXPLICIT WARRANTY. TAKING IN ACCOUNT THE NATURE AND THE FINALITY OF THE USAGE RIGHTS CONCEDED, 'VALIDY' EXCLUDES ANY IMPLIED WARRANTY OF QUALITY OR OF ADEQUACY TO A SPECIFIC USAGE, AS WELL AS THE ABSENCE OF COUNTERFEITS.

If the laws of your country grant you implicit guarantees, despite the exclusion and the nature, and the finality of the present license, your resorts are the ones are those stated in the article herein, relating to the resorts in case of guarantee violation, in the limits allowed by the rights of your country.

A.16.3 Exclusions from warranty

This warranty does not cover problems and malfunctions of any nature generated by your own actions, errors, or negligence, as well as by any action from a third party or any event external to the 'software package'.

THE PRESENT WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY ALSO BENEFIT FROM OTHER RIGHTS UNDER THE LAW OF YOUR COUNTRY.

A.17 Additional functionalities

'VALIDY' can supply additional functionalities to the 'software package'. These additional functionalities may be subject to other license conditions.

I declare having taken knowledge of the terms of this contract, and accept all its clauses.

Appendix B

Third Party Libraries and Tools Licenses

ASM is distributed under the following license:

```
Copyright (c) 2000-2005 INRIA, France Telecom  
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"  
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE  
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE  
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE  
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR  
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF  
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS  
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN  
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)  
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF  
THE POSSIBILITY OF SUCH DAMAGE.
```

Jakarta Commons CLI and IzPack are distributed under the following license:

Apache License

Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner

or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
 3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
 4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and
-

attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

- (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
 6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
 7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
-

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
