VMCrypt

V1.4

Lior Malka, 2010

Developers Manual,
Revision 1.0

Introduction

VMCrypt is a Java software library. It provides *secure computation capabilities* to Java applications. All the needed modules, including client-server modules, encryption modules, and oblivious transfer protocols, are included. VMCrypt can be easily integrated into existing projects. Moreover, all modules are customizable, and in fact they can be used to implement applications outside the realm of cryptography, such as generic client-server protocols. VMCrypt is based on Yao's garbled circuit technique. For scientific background on this technique and software architecture of VMCrypt, see the paper *VMCrypt - Modular Software Architecture for Scalable Secure Computation,2010* by Lior Malka and Jonathan Katz (available from ePrint: http://eprint.iacr.org/2010/584.pdf).

About this manual

The goal of this manual is to make VMCrypt integration as easy as possible to developers. The manual shows how to set up VMCrypt and run the sample applications included. Knowledge of cryptography is not necessary to run the examples, although it is needed to understand the underlying algorithms. Basic Java skills are required. The most technical topic in VMCrypt is component creation, which we do not cover here, but see the paper mentioned above: *VMCrypt - Modular Software Architecture for Scalable Secure Computation*. To automatically generate the documentation for VMCrypt, see the appendix. The revision history is also in the appendix.

Disclaimer

VMCrypt is free software. It is distributed on an "AS IS" basis, without warranty of any kind, either express or implied. See the License included with the source files for the specific language governing rights and limitations. Some countries impose export or trade restrictions on cryptographic technology. Some or all of the technologies used in VMCrypt may be protected by intellectual property laws. Other restrictions may apply. This manual should not be seen as a solicitation to use VMCrypt.

Setting up the development environment

Everything in this manual is written assuming you are using Linux. You may want to consider installing one of the Linux distributions, like Ubuntu, as the installation process is very simple and the user interface is almost identical to that of Windows. In any case, the principles given here equally apply to Windows. The version of your operating system does not matter. You will need a PC with text editor and Java installed. If you are not sure whether you have Java installed, open a command line interface (also known as *shell* or *terminal window*) and execute "javac" (the Java compiler) and "java" (the Java Virtual Machine). If one of these commands is not found, then you need to install Java. In Linux, you can install the entire Java Software Development Kit (SDK) by typing from the shell:

```
sudo apt-get install java6-sdk
```

Next, you will need to obtain two files: *samples.jar* and *VMCrypt.jar*. The first file contains the source code of the VMCrypt samples, and you will need to extract it. The second file contains the VMCrypt source code. If you only want to run the samples, then you do not have to extract it. However, in this manual we will show how to control some flags in the source, so you will have to extract this file too. The following instructions show how to extract the files. In this example, the home directory is */home/Lior*, and there is a *Documents* subdirectory where the files *samples.jar* and *VMCrypt.jar* are stored:

```
lior@ubuntu:~$ cd Documents

lior@ubuntu:~/Documents$ jar xf VMCrypt.jar

lior@ubuntu:~/Documents$ jar xf samples.jar
```

If all went well, you should see two new directories (*VMCrypt* and *samples*) created inside the directory *Documents.* You can safely erase *META-INF* if such directory was created. You can list the contents of the subdirectory in Linux by executing

```
lior@ubuntu:~/Documents$ ls –la
```

We remark that you do not have to store VMCrypt under */Documents* (which is under your home directory), but if you choose a different directory, then modify the path defined in the file *Makefile.inc1* found in *VMCrypt/makeInc*. The last step is to set the Java class path so that Java files can be located when you compile or run your Java programs. There are several ways to go about it, but in this manual we will simply set the CLASSPATH environment variable. This can be done by executing:

```
export CLASSPATH=/home/lior/Documents:.
```

Several remarks are in place. Firstly, notice that no extra space is allowed around the "=" symbol. Secondly, notice that the "export" command will only set the CLASSPATH variable in the shell where it was executed. In other words, the variable is not system wide and does not affect other shells. Hence, when you open a new shell, which you will do when you run a client and a server, you will need to repeat this step in the new shell. Thirdly, the ":" symbol is a separator between paths. In this case we have two directories in the class path: the *Documents* directory and the local (".") directory. The reason for this setup is that we will always execute java or javac from one of the sub directories of *sample*, and therefore we would like the path to include the local directory, as well as the root path to the VMCrypt package.

If your CLASSPATH is not set correctly, then when executing java on file name XXXXXX you may get an error like

```
Exception in thread "main" java.lang.NoClassDefFoundError: XXXXXX
```

and when executing javac on file name XXXXXX you may get an error like

```
XXXXXX.java:24: cannot find symbol
symbol  : class YYYYYY
location: class XXXXXX
```

If you see one of these error messages, then you need to set CLASSPATH correctly. Move on only after you fix this issue.

Example 1 – Computing a circuit

Like all the examples in this manual, this one assumes that you have set up your Linux environment as described earlier. We will start with the easiest thing in VMCrypt: computing a circuit. In VMCrypt, circuits are derived from abstract class *Component,* so from now on we just use the notion of a *component* whenever we refer to a circuit. Notice that there is no secure computation at this stage. We simply take a component, which is a representation of a piece of hardware (that is, gates and wires), feed it with input bits, and display to the screen the bits coming out of the output wires of the component. Let us start. Go to the *samples/component* directory and execute *make:*

```
lior@ubuntu:~/Documents/samples/component$ make
```

The *make* utility is part of Linux. It takes a file called *makefile* as input, and executes instructions based on this file. In this case the instructions are to compile the *TestComponents.java* and *Main.java* files. Once the .java files have been compiled, you will see that corresponding .class files have been generated for them. You are ready to run your first VMCrypt application! Execute:

```
lior@ubuntu:~/Documents/samples/component$ java Main
```

On the shell screen you will see a list of input-output pairs from three different components. Inputs and outputs are displayed at their bit representations. For example, the output

___ Testing BitADD on 2^3 inputs. Effective outDegree = 2 ___
Input 000 Output 00
Input 100 Output 01
Input 010 Output 01
Input 110 Output 10
Input 001 Output 01
Input 101 Output 10
Input 011 Output 10
Input 111 Output 11

Means that a component called *BitADD* is being computed. The inputs range from 000 to 111. These three bits include two input bits and a third carry bit. The output bits include one sum bit and one carry bit. If you open the *TestComponents.java* file, you will see that only one of many suite of tests are executed. Other tests have been commented out so that you do not get overwhelmed with the output.

Our next step in this example is to understand how the component is being computed. We only give a high level description. As you can see in *TestComponents.java, t*he VMCrypt class *TestModule* receives the component. This class is from VMCrypt.util, which means that it is located in the VMCrypt/util directory. *TestModule* creates two objects from VMCrypt classes called *StandardInput* and *StandardOutput*. At each iteration, *TestModule* proceeds to the next input by invoking the *next()* method of *StandardInput*. It then passes the component, the *StandardInput*, and the *StandardOutput* to a class called *CalculateNotifier* (from VMCrypt.function)*. CalculateNotifier* runs over the bit sequence from *StandardInput*, and for each bit in this sequence it notifies the corresponding input wire of the component. More accurately, it notifies the component, which then forwards the notification to the wire. The bit value (0 or 1) is not the only thing passed to the component. Other parameters include the function, which, in this case, is the *Calculate* function. When a gate has two inputs ready, it passes these inputs to *Calculate*, which computes the function represented by the gate, and returns the output to the gate. The gate notifies its output wire, and the process continues until *StandardOutput* is notified.

It is very hard to imagine how signals pass in a component. In fact, signals pass not only through gates and wires, but also through other objects, such as a *Bus,* a *Buffer* and the *PTable*. To help developers trace signals in components, VMCrypt provides a special class: the *Monitor*. Many VMCrypt classes report to the monitor about their events. For example, class *Gate* reports about gate build, gate notification, and gate destruction. To turn the monitor on, open *Monitor.java* (from VMCrypt.util) and check that the MONITOR variable is set to *true*. You will notice that the qualifier of MONITOR is *static final public.* All VMCrypt classes use this variable to decide whether they need to report or not. The fact that the variable is final means that its value cannot be changed after compilation. It is hard wired into the code. Thus, if we compile, for example, class *Gate* when MONITOR is *false*, then *Gate* will not send notifications, and even if we change MONITOR to *true* and recompile class *Monitor,* still class *Gate* will not send notifications. The lesson is: whenever you change the value of MONITOR, you need to rebuild the entire code. This can be done by executing from the VMCrypt directory:

```
lior@ubuntu:~/Documents/VMCrypt$ make again
```

"make again" first calls "make clean" (which removes all .class files) and then "make" to recompile all the *.java* files. You can run "make", "make again" or "make clean" from any VMCrypt subdirectory, and they will apply recursively to the subdirectories rooted at that directory. Back to class *Monitor,* notice that all events are reported to this class, but not all are displayed. In this example we want to see all events, so in class *Monitor* you should find array *componentEvents* and uncomment all the events so that they are included. Also make sure that the following variables are set:

- ✓ COMPONENT_EVENT_FILTERING_IS_ON = true
- ✓ COMPONENT_EVENT_FILTERING_IS_INCLUSIVE = true

You do not need to understand what these variables do at this stage, but the *Monitor* class is extremely simple, so you can later return to this file to learn how to control all the notifications. This will be very handy when debugging new components. When you change the value of these variables, you do not need to build the entire code again. All you need is to recompile *Monitor.java.* Remember - you can execute *make* from any subdirectory of VMCrypt, and it will compile all the files that have been modified under the hierarchy rooted at this subdirectory. So, to compile *Monitor.java* you can simply execute *make* from /Documents/VMCrypt.

Finally, before you rerun the test, open the file *TestComponents.java* mentioned earlier (located at samples/component) and in the method *start()* make sure that all tests except for *test1()* are commented out. Also, to make sure that *test1()* computes only one component, say the *BitMUX* (bit multiplexer), comment out the last two lines of *test1()* as shown below:

```
void test1() {
       test(new BitMUX(bus));
       //test(new BitCMP(bus));
       //test(new BitADD(bus));
}
```

Since you modified *TestComponents.java,* execute *make* to recompile and then run *Main.java:*

```
lior@ubuntu:~/Documents/samples/component$ make

lior@ubuntu:~/Documents/samples/component$ java Main
```

You will see that this time the output is much more verbose. This is because many objects are reporting to the Monitor. Notice that the *BitMUX* component has been computed on all possible inputs. Thus, you are actually looking at 8 computations. The reports sent to the Monitor can be roughly divided into two groups:

- ➢ Notifications: when an object reports about receiving or sending a value.
- ➢ Memory: when an object is constructed or destructed.

Notice that protocols also have their own set of events, which are useful for tracking protocol progress and obtaining protocol statistics. Back to our example, the output on your screen will look like this:

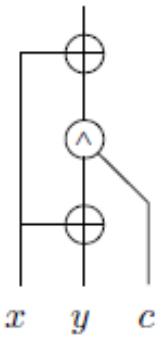
```
PTable initializes row 0 of size 2
PTable[0][0] sets object=0
PTable[0][0] sets object=1
___ Testing BitMUX on 2^3 inputs. Effective outDegree = 1 ___
(0,0) BitMUX is object-notified on port 0 with 0
(0,0) BitMUX Buffer of size 3 is constructed
(0,0) BitMUX Buffer is notified at port 0 with 0
(0,0) BitMUX is object-notified on port 1 with 0
(0,0) BitMUX Buffer is notified at port 1 with 0
(0,0) BitMUX is object-notified on port 2 with 0
(0,0) BitMUX Buffer is notified at port 2 with 0
(0,0) BitMUX is built. InDegree=3, OutDegree=1
XOR gate is built.
XOR gate is built.
AND gate is built.
(1,2) AND gate is notified on port 1 with value 0
(1,0) XOR gate is notified on port 1 with value 0
(0,0) BitMUX destucts Buffer of size 3
```

```
(1,0) XOR gate is notified on port 0 with value 0
(1,0) XOR gate is destructed.
(1,2) AND gate is notified on port 0 with value 0
(1,2) AND gate is destructed.
(1,1) XOR gate is notified on port 1 with value 0
(1,1) XOR gate is notified on port 0 with value 0
(1,1) XOR gate is destructed.
(0,0) BitMUX is destructed.
(0,0) BitMUX Bus forwards a notification to Standard Output on port 0 with value 0. Counter=0
Standard Output received 0 on port 0
Input 000 Output 0
```

We explain the above transcript. Ignoring the PTable, the test begins with the test announcement:

```
___ Testing BitMUX on 2^3 inputs. Effective outDegree = 1 ___
```

The announcement reports the name of the component being tested, followed by 2^3. The number 3 is the *inDegree* (input degree) of the BitMUX. The *outDegree* (output degree) is 1, and we will ignore the *Effective outDegree* for now. You can interpret the announcement as saying "this is a circuit with 3 input wires and 1 output wire, and we are going to feed it with all possible 2^3 inputs." The component is a bit multiplexer: it takes a first and a second bit (denoted *x* and *y*), and based on the third bit (denoted *c*) it outputs either *x* or *y*. Here is a mathematical sketch of this circuit. The links represent wires, and the circles represent gates (either *AND* or *XOR* gates).



(a) BitMUX

When a component receives a notification, it *routes* it to the appropriate sub component. In this case, however, we have a *Circuit* component. More accurately, *BitMUX* extends class *Circuit.* Because circuits have a *Buffer,* only when all notifications to the *BitMUX* have been received does the buffer flush the notifications into the circuit. Flushing simply means that the buffer notifies each wire with the signal it was supposed to receive.

Going back to the transcript, you can see that when the *BitMUX* is notified for the first time, it builds the buffer and notifies the buffer with the signal. Subsequently, each time *BitMUX* is notified, it notifies its buffer. You may have already guessed that *port* is the index on which notifications are received. In the case of circuits, when we say that *BitMUX* is notified on port 2 with 0, we mean that the value 0 is for wire number 2. However, remember that class *Circuit* is the only class with input wires, so for classes that do not have input wires we use the notion of a port to describe the destination of the signal.

The pair (0,0) means that *BitMUX* is the 0-th component at depth 0. This means "the root component". Inside the *BitMUX* there are three gates, so their depth is 1 higher, and they are numbered 0,1, and 2. For this reason they are described as (1,0), (1,1), and (1,2). The depth and the index are assigned automatically by VMCrypt. They are not used for any purpose other than reporting. Without them, you would not be able to follow signals travelling inside components, and you would not be able to control what events (e.g., from what depth) you would like the Monitor to display.

Once the buffer receives notifications on all ports (that is, 0,1, and 2) it invokes the *buildCircuit()* method of the circuit. As you can see in the transcript above, this method creates the internal sub components. In this case, these are three gates. At the end of the build process, wires are constructed and connected. Finally, the buffer calls the *notify_sub_component* method of the circuit, which in the case of class *Circuit* simply passes the notification to the appropriate wire. Wires do not notify the monitor

when they receive signals because this would be redundant; as you can see each gate reports to the monitor that it has received a signal. Each gate self-destructs as soon as it is done computing and then it forwards its output on its output wire.

Each VMCrypt component has a *Bus*. The bus has two functions. The first one is output. Specifically, the bus has an *out* variable referencing some notifiable object. This object is usually a component, but in this case it is *StandardOutput*. When a component wants to output a signal, the bus of this component is notified with this signal first. The bus then forwards the signal to the notifiable object referenced by *out*. In our example, the (1,1) XOR gate produces an output, this output flows on an output wire, the wire notifies the bus of the *BitMUX,* and then the bus notifies *StandardOutput*.

The second function of the bus is to destruct the component in which it resides. Specifically, the bus has a counter, initially set to the *outDegree* of the component (in this case 1). Each signal leaving the bus decreases this counter. When the counter hits 0, before the bus forwards the notification, it calls the *destruct()* method of the component. This can be seen in the transcript above: *BitMUX* destructs first, and *StandardOutput* receives the notification second. Misusing the *Bus* can cause many unforeseen bugs (see the Appendix).

We are done with this example. If you want to expand your learning, here are a few exercises:

1. Open the file *BitMUX.java* (*located at VMCryp/primitive*) and compare the content with the drawing shown above.
2. In the file *TestComponents.java* uncomment other tests to see what they compute.
3. In the file *Monitor.java* modify the set of displayable events. Also try filtering events by depth.
4. In the file *TestComponent.java*, find the variable controlling the number of tests. Set its value to, say, 3. Run the tests.
5. In the file *TestComponent.java*, add a new test method that computes the *MUX* component. Hint: your method should look exactly like test6(), except that instead of *BinaryMin* you need another class name.

Example 2 – Running a VMCrypt client-server application.

VMCrypt provides a generic client and a server. In addition, VMCrypt provides special input/output streams that allow the client and the server to send data to each other. The Java SDK streams *ObjectInputStream* and *ObjectOutputStream* provide this functionality too, but at the extreme speeds at which VMCrypt protocols run, these streams will crash your machine after five seconds. Another useful property of VMCrypt streams is that they can report the amount of data sent and received. No Java stream provides this capability. Moreover, VMCrypt streams can maintain multiple counters, so that even if your client and server run several protocols, the amount of data sent by each protocol can be measured individually. The example in this section demonstrates the VMCrypt Client, the VMCrypt Server, and the VMCrypt streams.

You will need to open two shells. In each shell you will need to set the CLASSPATH. If you are not sure how, then see the section on setting the development environment. Next, go to the directory *samples/net* and compile the java files:

```
lior@ubuntu:~/Documents/samples/net$ javac *.java
```

As shown above, we are assuming, as usual, that subdirectories *VMCrypt* and *samples* are stored under /home/lior/Documents.

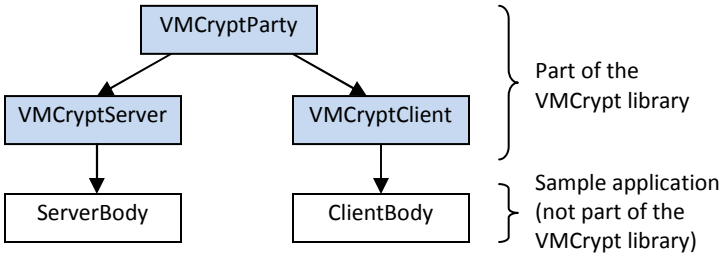
Now, from the first shell execute:

```
lior@ubuntu:~/Documents/samples/net$ java ServerMain
```

and from the other shell execute:

```
lior@ubuntu:~/Documents/samples/net$ java ClientMain
```

You should see some output on the screen, indicating that the client and the server are sending data to each other. Let us go through this example. Open the files *ServerMain.java* and *ClientMain.java*. You can see that the server simply creates an instance of *ServerBody* and starts it. The same applies to *ClientMain.* If you go to *VMCrypt.net,* you will see that Class *ServerBody* extends class *VMCryptServer* and that *ClientBody* extends *VMCryptClient*. Both *VMCryptServer* and *VMCryptClient* extend *VMCryptParty,* which owns the VMCrypt input and output streams.



Did you notice that in *ServerMain* the server is strated using the method *start(),* but in *ServerBody* there is no such method? What actually happens is that, when you start a party (that is, a client or a server), there are some methods in class *Party* that handle setting up the network connection, and after they finish their job, they call the *run()* method, which is what implemented in *ServerBody* and *ClientBody*.

Did you also notice that the stream variables *in* and *out* are defined in class *Party,* but they are overridden by the *in* and *out* variables in *ServerBody* and *ClientBody*? The stream variables *in* and *out* from *ServerBody* and *ClientBody* are instances of the special VMCrypt stream classes we talked about above. These classes are called *CounterInputStream* and *CounterOutputStream,* but the actual work they do is implemented by their super classes, *ArraysInputStream* and *ArraysOutputStream,* respectively. The stream classes can be found in *VMCrypt.io*. All VMCrypt protocols use these streams. They let *VMCryptParty.start()* initiate the communication channel (and connect *VMCryptParty.in and VMCryptParty.out* to the channel) and then they create instances of VMCrypt streams (and connect them to *VMCryptParty.in* and *VMCryptParty.out).*

There are two important conclusions from the above discussion:

1. If you want to create a client and a server from scratch, then extend *VMCryptServer* and *VMCryptClient.*
2. You can use your existing client and server, but if you want to run VMCrypt Protocols, then you must use the special VMCrypt streams. You instantiate these streams just as shown in *ServerBody* (or *ClientBody).*

We now look at the actual sending and receiving of messages. Looking at classes *ServerBody* and *ClientBody* you can see that writing to the streams is done by using *out.XXXX* where XXXX is the name of the method describing what is being sent. For example, in the *run()* method of *ServerBody* you can find the instruction *out.write_Int(length);* which sends an integer to the client. Similarly, reading from the stream is done using an instruction of the form *in.XXXX,* where XXXX is the name of the method describing what is being read. Again, if you look at the *run()* method of *ServerBody* you can find the line *message2A = in.readArray(message2A, length);* which reads a byte array from the network. Notice that "message2A=" implies that *in.readArray* returns a reference to the byte array. Why, then, do we also need to provide *message2A* as an argument to *readArray?* Because there are many *readArray* methods in *ArrayInputSream*, and they all read different arrays. When you pass *message2A* as an argument, you tell Java to invoke the *readArray* method that corresponds to a *byte[] array,* which is the type of *message2A.*

The instruction *out.flush()* flushes the stream. You must use it after each sequence of *write* operations, or else the protocol would freeze because the other party is waiting for a message that will never arrive.

We already mentioned that the VMCrypt streams are capable of counting the amount of data sent and received. You can see in the constructor of *ServerBody* that the counting unit is set to kilobyes by passing the constant *Counter.Unit.kilobyte*. This constant is actually implemented as a Java *enum* type, as you can see in class *Counter* in *VMCrypt.util.* You can set the counter unit to any measurement (e.g., megabyte), but notice that the *Counter* class has limited support for overflows (see the source).

Finally, you may wonder what the instruction *in.meta = ArraysInputStream.META._byte;* from ServerBody.*run()* does. The answer is that, when arrays are sent over VMCrypt streams, the length of the array is written into the stream, just before the array itself it written. This is done so that the receiver knows how many bytes it should read from the network. The above instruction tells the input stream that the size of the integer describing the length of the array is one byte. Thus, if your application is sending millions of small arrays, then instead of attaching an integer to each of these arrays, you save on bandwidth by only attaching one byte. In fact, VMCrypt streams can send and receive data with no metadata at all. This can save a lot of time and bandwidth, but it also means that you need to know exactly how many bytes to read on the receiver end. Almost all VMCrypt messages are sent without metadata, and even control metadata is not used (that is, the client side and the server side in all of the VMCrypt protocols are synchronized by design).

In our client-server example, the client is the party sending the last message. To test you understanding, try the following:

1. Make the client send an array containing "hello world" to the server.
2. Modify the server so that it displays this string in color other than green (see class *Color* in *VMCrypt.util*).
3. Make the client send the array "hello world" with metadata of size *int*.
4. Make the client send the array "hello world" without metadata (see classes *ArraysOutputStream* and *ArraysInputStream*).
5. Make the parties communicate via a different port than the default one (see *ClientBody* and *ServerBody*). If you have a second computer, try running the client and the server from different computers.

In the above exercise, make sure you modify both parties as necessary, and do not forget to flush the stream.

Example 3 – Running an Oblivious Transfer Protocol

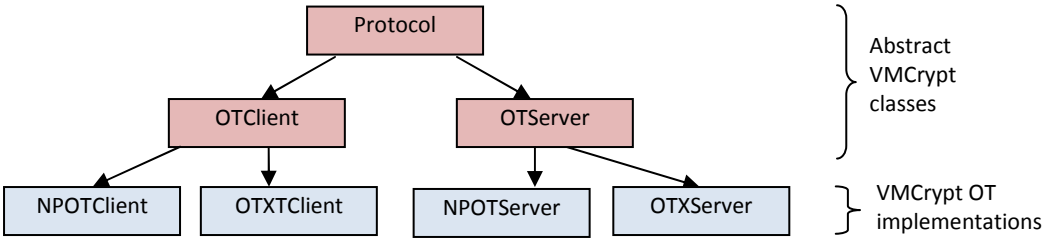
In this example we will run an Oblivious Transfer protocol (OT). Briefly, in such a protocol the server has a pair of messages, say <"secret1","secret2">, and the client wants to learn exactly one of these messages, but without telling the server which one. Similarly, the server is willing to reveal exactly one of the secrets, but not both. Thus, at the end of an OT protocol, the client learns "secret1" or "secret2" (but not both), and the server does not learn anything. In particular, the server does not know which secret the client chose.

To run this example, follow the same steps as in Example 2. That is, open two shells and define the CLASSPATH in each one of them. Then, go to the directory *samples/OT* and compile the java files. Finally, run *ServerMain* from one shell, and *ClientMain* from the other. The client displays messages it receives from the server, and the server does not display anything.

We go through the code, starting with */samples/OT/ServerBody.java.* In the *run()* method you will see that the server starts an OT server inside a loop. That is, the OT protocol is executed 5 times. The same loop appears in the *run()* method of the client, found in */samples/OT/ClientBody.java,* except that after the client finishes each iteration, it displays its output on the screen. In each of the five executions, the server has *arity*=4096 pairs of messages. These are stored in an array *m*. The client has an array *b* of size 4096, and b[i]=0 or 1 depending on whether i is even or odd . The client uses the value of b[i] to choose between m[i][0] to m[i][1]. This choice is displayed on the screen. To test for correctness, we set the first cell of m[i][0] to 0, and the first cell of m[i][1] to 1. Thus, the first byte in the i-th message learned by the client should be b[i].

It is important not to confuse protocols with servers and clients. In the previous example we have used the body of the client and the server to send and receive messages. This was done only to simplify the presentation. If you open files */samples/OT/ServerBody.java* and */samples/OT/ClientBody.java*, you will see that protocol logic is decoupled from the client and the server. This is how you should design protocols. In VMCrypt, all protocols derive from the base class *Protocol* (found in *VMCrypt.protoco.base*). If you plan on adding a new protocol called XXXX to VMCrypt, then create two classes (called XXXXServer and XXXXClient) that extend this class.

In the case of oblivious transfer, the classes extending *Protocol* are *OTClient* and *OTServer* (found in *VMCrypt.protocol.*OT). Notice that these are abstract classes; the OT protocols that VMCrypt provides extend these classes. This means that if you want to integrate your own OT protocols into VMCrypt or create new ones, then you should extend these classes. For more details, see http://www.lior.ca/publications/api_design.pdf (*How to Design APIs for Cryptographic Protocols*).



The methods *NPOT()* and *OTX()* instantiate an OT protocol. Regardless of which type of protocol they instantiate, this protocol is started in the method *run()*. VMCrypt implements two OT protocols (see *VMcrypt.protocol.OT*):

1. The Naor-Pinkas protocol [NP01], implemented in classes *NPOTClient* and *NPOTServer*. Since this protocol takes many parameters, we have created the class *NPOTParams* as a container for these parameters. The protocol generates a public key (which is not secret, as the name suggests) and store it in the file *NPOTServer.public.* If you erase the file, the protocol will regenerate it. We remark that the implementation of this protocol follows that of the original paper, so there is much room for improvement.
2. The Ishai-Killian-Nissim-Petrank protocol [IKNP03], implemented in classes *OTXClient* and *OTXServer*. The "X" stands for "extension". The reason for this "X" is that the [IKNP03] protocol actually extends other OT protocols. It does not stand on its own. In this protocol, the *OTXServer* runs an OT client as a sub protocol, and the *OTXClient* runs an OT server as a sub protocol. They run these underlying OT protocols 80 times to get an effective number of 4096 OT, hence the notion of extending 80 to 4096. The number 80, represented by the variable *sub_OT_arity,* is part of the security parameters. Do not decrease it. For more information regarding the security parameters see [IKNP03].

Example 4 – Running a Garbled Circuit Protocol

In this example we run a garbled circuit protocol. Following the same steps as in Example 2, open two shells and define the CLASSPATH in each one of them. Then, go to the directory *samples/GC* and run *make* to compile the java files. Finally, run *ServerMain* from one shell, and *ClientMain* from the other. If you see too many lines displayed on the screen, then this is probably due to uncommented component events in class *Monitor* that were previously commented. To fix this, open class *Monitor* in *VMCrypt.util*, comment all elements in the array *componentEvents,* and finally recompile by executing *make* from, e.g., the directory *VMCrypt.* We mention that having the Monitor turned on has a severe impact on performance, so if you are testing for performance, set *MONITOR=false* and rebuild (for details, see Example 1).

During execution, the client and the server display status information on the screen. The *index* is the current wire on which wire-labels are available. The index is displayed in multiples of 10,000. If, e.g., the last index displayed is 40,000, then the component being garbled has between 40 to 50 thousand input wires. Notice that in VMCrypt wire locations are formally called *ports*. The number of the gate being garbled (alternatively, evaluated) is also displayed. You can remove it by opening the file *Garble.java* (alternatively, *Eval.java*) in *VMCrypt.function,* and commenting the appropriate line.

Before we proceed, it is important to realize that, in VMCrypt, creating a component and running a GC protocol are two completely separate things. When you create a component, you first want to test it for functional correctness. For more information on how to create a component, see the paper *VMCrypt - Modular Software Architecture for Scalable Secure Computation,2010* by Lior Malka and Jonathan Katz (available from ePrint: http://eprint.iacr.org/2010/584.pdf). The ability to create and debug components independently of the underlying protocols is a huge plus in terms of software development.

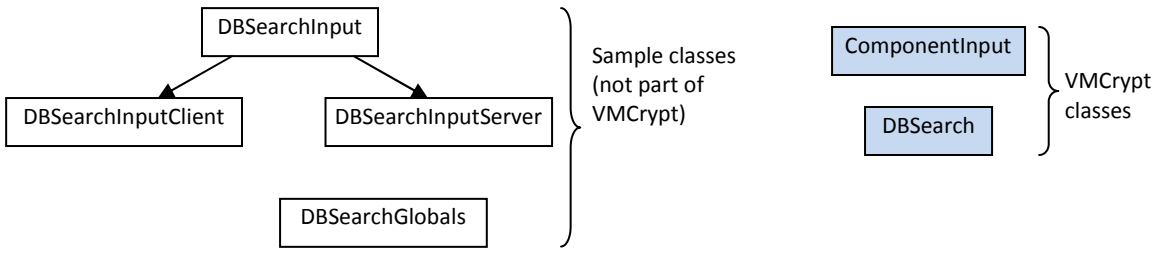
The *ServerBody* and *ClientBody* from our example can run the GC protocol on three different components. In *ServerBody.java* from */samples/GC* you can see that depending on which method is called, the instantiated component is as follows:

1. *testMIN()* – the component is computing the minimum of a set of numbers.
2. *testDBSearch()* – the component is doing a database search.
3. *testSetIntersection()* – the component is computing the intersection between two sets.

You can try running the GC protocol with other components. Just remember that when you modify *ServerBody,* you may also have to change *ClientBody* (this is the case if you want to test other components).

From an algorithmic perspective, the class that orchestrates the garbling and evaluation process is *Notifier* from *VMCrypt.protocol.WLTP. WLTP* stands for *Wire-Label Transport Protocol*. The notifier drives the GC protocol. It obtains objects from the WLTP protocol and feeds these objects into the component. Specifically, in WLTPServer, which is the server side of the WLTP protocol, the notifier obtains pairs of wire-labels. It notifies the ports of the component with these pairs. In WLTPClient, which is the client side of the WLTP protocol, the notifier obtains wire labels (as opposed to wire-label pairs) and notifies the ports of the component with these wire-labels. The component is notified not only with objects, but also with a function, which is *Garble* on the server side and *Eval* on the client side. The notification travels in the component and will eventually hit a gate. When a gate receives all of its objects, it applies the function to the objects. The output wire of the gate is then notified with the output of the function, and the process continues until all gates are notified. We mention that VMCrypt has two types of gates: UniGates (one input, one output) and BinaryGates (two inputs, one output), both can be found in *VMCrypt.component*.

We will walk through the GC protocol only with respect to the *DBSearch* component; the *MIN* and *SetIntersection* components follow the same principles. The input of the client is defined by the class *DBSearchInputClient.* Similarly, the input of the server is defined by the class *DBSearchInputServer.* Both classes extend *DBSearchInput,* which implements *ComponentInput*. Global variables, such as the database size and the bit length of database elements are contained in the class DBSearchGlobals. Intuitively, the purpose behind these classes is to allow VMCrypt to access the input to the parties. Notice that you can calculate the DBSearch component as we did in Example1.



The *isServerInput(i)* method in *DBSearchInput* returns true if and only if the i-th input bit to the component is a server input. The WLTP parties use this method in order to decide whether a wire-label should be transferred using OT or not. The *inDegree()* method in *DBSearchInput* allows the WLTP parties to determine how many wire-label pairs will be transferred. The *getInput(i)* method in *DBSearchInputServer* is used for a similar purpose. Specifically, given a wire-label pair on port i, the *WLTPServer* calls *getInput(i)* to decide which of the wire-labels should be sent to the client. Similarly, the *WLTPClient* invokes *getInput(i)* of *DBSearchInputClient* to decide which wire-label to ask for during the OT sub-protocol.

You will notice that the other components (MIN and SetIntersection) also have four files to support their secure computation. If you want to run the GC protocol on these components, then all you need to do is make sure their parameters reflect your computation. These parameters include: the number of input bits, which inputs belong to which party, the value on these inputs, and so on. For any other component, you will need to make copies of these files (under a different name, of course) and then modify them to suit your needs.

Let us walk through the code in the *run() method from ServerBody.* The *testDBSearch()* method instantiates the *DBSearch* component. It sets the *Bus* of this component to be a *Terminal*. Informally, the Terminal is needed because the function computed on each gate may depend on whether the output wire of this gate is also an output wire of the entire component or not. We remark that VMCrypt wires "do not know" if they are output wires of the component (you may be tempted to think that if a wire points to NULL, then it is an output wire of the component, but this is not the case). VMCrypt uses the *Terminal* to define behaviors that should be taken on output wires. See the *terminalCompute* method from *Function* (in *VMCrypt.component)* for more. The *PTable* defined *testDBSearch()* is a global two dimensional table where components can push messages for other components to pull. Once the variables *component, componentInput,* and *pTable* are defined, the *run()* method in *ServerBody* can instantiate the server side of the GC protocol (called *GCServer*) and start it. The above description also applies to the *run()* method from *ClientBody,* except that there we have a few more lines that take care of displaying the client output and measuring the running time.

Before we conclude this section, we note two non-trivial facts. The first one is that the *PTable* is a global table. Components push objects to and pull objects from the *PTable*. Yet, the *PTable* is not one of arguments passed to the components during notification. Why? Because the *PTable* is included inside the *Function,* which is passed as an argument during notification. The second fact pertains to the *Garble* function. For each gate, this function prepares a lookup table. This lookup table depends on whether the output wire of the gate is an output wire of the component or not. However, as we mentioned earlier, VMCrypt output wires "do not know" if they are output wires of the component or not. Consequently, when the *Garble* function handles a gate, it does not know whether it should garble for an inner gate or not. To work around this problem, the *Garble* function is doing "delayed" garbling. That is, it stores all of its input in a state called *StateOfBinaryGate* (an inner class of *Garble*) instead of preparing a lookup table. Notice that after a function (in this case *Garble*) is computed on a gate, the output wire of the gate is notified. Now, if this wire is an output wire, then the notification will hit the *Terminal* immediately after *StateOfBinaryGate* is updated. This will cause the *terminalCompute* in Function (in this case, the *terminalCompute in Garble)* to take the contents of *StateOfBinaryGate* and prepare a different lookup table than for inner gates. This solves the problem.

VMCrypt directory structure

VMCrypt contains many modules. These modules are stored in subdirectories. In addition, the modules are part of a Java package. The package name and the directory must coincide. For example, the first line in the *Monitor.java* file is "package VMCrypt.util;". Since Monitor belongs to *VMCrypt.util*, it is stored in VMCrypt/util. The table below describes the subdirectories and their contents.

Directory	Contents	Example
component	The "kernel" of VMCrypt. This directory contains the core classes for building and managing components	PTable, UniGate, BinaryGate, Wire, Bus, Buffer, Circuit, Switch, UniSwitch, Map, ComponentEvent, Function
primitive	Basic components	MUX, CMP, BIN, EQ
composite	Complex components	MIN, DBSearch, EQMUX
function	Functions that can be applied to gates	Calculate, Garble, Eval
net	Network support	VMCryptClient, VMCryptServer
io	Input and output support for streams	CounterInputStream, CounterOutputStream
util	Utilities	Monitor, TestModule, BitOperations
crypto	Cryptographic tools	LookupTable, SymmetricEncryption
protocol		
base	Generic protocol classes	Protocol, ProtocolEvent
GC	Yao's Garbled Circuit protocol	GCParty, GCClient, GCServer, Notifier
OT	Oblivious Transfer protocols	OTClient, OTServer, OTXClient, OTXServer
WLTP	Wire Label Transfer Protocol	WLTPClient, WLTPServer, ComponentInput

Appendix – Bus pitfalls

This section provides insight into how you may accidently create bugs related to the Bus, and what to do so that such bugs are avoided. You may encounter such bugs either during the development of new components, or the usage of existing ones.

1. When you construct a component, you pass an instance of class *Bus* that is already being used by another component. This is fatal and can cause arbitrary behavior. Remember: each component must have its own bus.
2. You have created a wire in component A. This wire is connected to component B. This is also fatal because the bus of component A will not be able to count the signal leaving component A. Consequently, component A will not destruct when it finishes computing. Remember: subcomponents of A only notify other subcomponents of A or the bus of A.
3. You want to use a component several times (e.g., you want to compute it on different inputs, like we have done in Example 1), but you forget that once the component finishes computing, the bus destructs not only the component, but also itself (that is, the bus is destructed). The next time you use the component, you will get some error that stems from the fact that the bus (originally not NULL) has a NULL value. Remember: whenever you use a component more than once, some of the objects you have originally constructed are set to NULL.
4. You have wired your component incorrectly, causing more signals to leave through the bus than the *outDegree* of your component. Consequently, the bus of your component destructs your component (and itself) prematurely. The next time your component is used (e.g., it receives a notification), you get an error stemming from the fact that your component (originally not NULL) has a NULL value. Remember: the outDegree is used only for one purpose – to tell the Bus when to destruct the component. A component should output exactly outDegree signals. Not more, not less.

Appendix – Packaging VMCrypt

VMCrypt and its samples come in two Java Archive Files (JAR) files: *VMCrypt.jar* and *samples.jar*. You may have unpacked these files and modified the source code. If you want to repack VMCrypt, then go the directory containing VMCrypt, which we assume is */home/lior/Documents,* and execute the following

```
lior@ubuntu:~/Documents$ jar cf VMCrypt.jar VMCrypt
```

This will create the file VMCrypt.jar in the directory */home/lior/Documents*. In the same way you can pack the samples.

Appendix – Generating the Documentation

There are four ways to learn VMCrypt

- ✓ Following the examples in this manual.
- ✓ Reading the paper *VMCrypt - Modular Software Architecture for Scalable Secure Computation,2010* by Lior Malka and Jonathan Katz (available from ePrint: http://eprint.iacr.org/2010/584.pdf).
- ✓ Reviewing the presentation *How to Design APIs for Cryptographic Protocols* (available from http://www.lior.ca/publications/api_design.pdf).
- ✓ Reading the VMCrypt source code and documentation – described below.

The Java SDK that you have installed on your computer should come with a tool called *Javadoc.* This tool automatically generates the documentation for VMCrypt. You can run Javadoc after any modification to the source, and it will generate new, up-to-date documentation. As usual, assuming VMCrypt is located in */home/lior/Documents,* you will generate the documentation by entering the directory where VMCrypt is stored, and executing

```
lior@ubuntu:~/Documents/VMCrypt$ make documentation
```

This will create a *doc* directory with all the documentation for VMCrypt. The *doc* directory will be stored at the parent directory of VMCrypt. That is, at */home/lior/Documents*. Enter this directory. You will see that it contains an *index.html* file (and many directories that this *index.html* file links to). This *index.html* file is a webpage. If you double click it, your browser will launch and display the documentation. If you plan to add comments to the source code of VMCrypt, make sure they are formatted according to Javadoc specifications. This will guarantee proper display of your comments in the web pages generated by Javadoc.

Revision History

Revision	Comments
1.0	Original revision