VMCrypt - Modular Software Architecture for Scalable Secure Computation

Lior Malka
Intel (work done while at UMD)
lior34@gmail.com

ABSTRACT

Garbled circuits play a key role in secure computation, but existing implementations do not scale and are not modular. In this paper we present VMCrypt, a library for secure computation. This library introduces novel algorithms that, regardless of the circuit being garbled or its size, have a very small memory requirement and use no disk storage. By providing an API (*Abstract Programming Interface*), VMCrypt can be integrated into existing projects and customized without any modifications to its source code. We measured the performance of VMCrypt on several circuits with hundreds of millions of gates. These are the largest *scalable* secure computations done to date.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*security and protection*

General Terms

Security, Performance, Algorithms

Keywords

Secure Computation, Software API, Scalable

1. INTRODUCTION

Secure computation enables parties to collaborate while keeping their information private. Specifically, a two-party protocol for secure computation allows a server (holding input x) and a client (holding input y) to compute $f(x, y)$ such that either party learns nothing beyond the output of f.

Yao's Garbled Circuits Technique [31] is central to secure computation. The first phase in this approach is to create a representation of f as a boolean circuit C. Next, in the *garbling* phase, the server chooses secret keys (called *wire labels*) for each wire in C, encrypts a *a lookup table* for each gate, and sends all lookup tables (the *garbled circuit*) to the client. The server also sends wire labels corresponding to x. Wire labels corresponding to y require oblivious transfer (OT). Finally, in the *evaluation phase* the client uses the wire labels for x and y to decrypt the lookup tables until the output gates are decrypted and $f(x, y)$ is revealed.

Yao's technique has been implemented in Fairplay [21] and TASTY [11], but these works focused on automation aspects of secure computation (more details in Section 1.3). This paper, however, is concerned with software engineering challenges.

The first challenge is scalability. Suppose, for example, that f computes the intersection of databases x and y. The circuit C representing $f(x, y)$ could easily have billions of gates. Holding it memory would require terabytes of RAM. Wire labels or lookup tables would require a large hard drive. Read/write operations will paralyze the operating system. Recompilation (needed in Fairplay and TASTY) will take a significant amount of time, ruling out secure computation on the fly.

The second challenge is software modularity. Consider, for example, a scenario where garbled circuits are to be integrated into an application that already has a client and a server, and that, in addition, a specific method of encryption should be used for garbling. The source code of the garbled circuit software can be modified to accommodate this integration, but this would significantly increase development costs and is likely to introduce bugs into the source. Ideally, developers would be provided with an *abstract programming interface* (API) so that they can customize the software without modifying the source code. This includes customizing the client and the server, the OT protocol, garbling, encryption, and so on. Also, modules should be designed in such a way that developers can use them to build a protocol with any level of security (we have implemented the curious-but-honest protocol). Another objective would be to allow developers to replace parts of their circuits with improved implementations by other developers, making modular not only the software, but also the circuits themselves. Finally, developers must be able to test circuits for correctness automatically, without having to run the secure protocol.

1.1 Our Results

We present VMCrypt - a fully customizable Java library for secure computation. VMCrypt introduces novel algorithms that, regardless of the circuit being garbled or its size, have a very small memory footprint and use no disk storage. VMCrypt comes with a *Developers Manual* [19] and provides tools for debugging and validation.

To describe VMCrypt, we use the example of the database

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

search protocol, where the server has records $\langle x_i, p_i \rangle$, and the client, who holds y, wants to learn all p_i for which $x_i = y$. We denote the circuit for this protocol by `DBSearch`.

Unlike Fairplay programs, VMCrypt circuits (called *components*) are simply Java classes. They are compiled once and for all and receive their parameters at run time, via the constructor. For example, `DBSearch` is instantiated at run time by executing `Component c = new DBSearch(`ℓ_x, ℓ_p, N`)`, where ℓ_x and ℓ_p denote the bit length of x and p, and N is the number of records in the database. In a realistic scenario where the number of records in the database changes daily, components allow us to start a VMCrypt protocol on the fly, whereas with Fairplay we need an expensive offline phase for recompilation (we compare with Fairplay because TASTY combines homomorphic encryption and garbled circuits. However, as we later show, the same applies to TASTY).

The second difference between VMCrypt and Fairplay relates to how circuit descriptions are maintained in memory. In earlier versions of VMCrypt, constructing a component (e.g., `new DBSearch(`ℓ_x, ℓ_p, N`)`) would have immediately created an instance of this object in memory. In the case of `DBSearch`, the size of the object is linear in $N(\ell_x + \ell_p)$. Thus, instantiating `DBSearch` is worse than holding the entire database in memory. This is clearly infeasible. Worse, it excludes secure computation from consumer devices such as laptops and cell phones. Fairplay suffers from the same issue.

VMCrypt solves this problem. Intuitively, it has an underlying virtual machine that loads and destructs hardware descriptions dynamically, during execution. This mechanism is transparent to the developer. It works for circuits of any size, and guarantees that the client and the server hold only a small part of the circuit in memory. See Section 3 for details.

To describe the third difference between VMCrypt and Fairplay, recall that in Yao's protocol the client evaluates the function after receiving the garbled circuit from the server. This requires storage (e.g., a hard drive) for wire labels or lookup tables, which, as our tests show, can reach gigabytes of data. Writing this much data to the hard-disk will paralyze the operating system and slow down execution.

VMCrypt eliminates the need for storage, which also improves running time. To achieve this, we introduce a new variant for Yao's protocol. Informally, our protocol divides the number n of input wires into segments of a constant size s (e.g., $s = 100$). In iteration i, the server issues the i-*th* segment of s wire labels to the client, following which it garbles the part of the circuit corresponding to this segment. The lookup tables are sent directly into the network. The client reads the lookup tables from the network and evaluates the i-th part of the circuit. This continues until the entire circuit is evaluated. To achieve this, we garble, evaluate, and execute OT at the same time (rather than sequentially). Interleaving OT in Yao's protocol is highly non trivial, especially because we execute OT in bulk, which is several orders of magnitude faster. Our new protocol is also attractive because it can be multi-threaded, which may further improve performance. More details, and a security proof, are given in Section 2.

We stress that all VMCrypt mechanisms (e.g., storing only part of the circuit in memory, or garbling the circuit in chunks) are completely transparent to developers. VMCrypt developers only need to describe the circuit (what we call *a component*) they want to securely compute and the input to the circuit.

1.2 Performance

Taking `DBSearch` again as an example, Fairplay compilation alone takes 50 seconds and 0.7 GB memory (23% of a 3 GB RAM) when the database size is 40 records. The Fairplay compiler crashes after 3 minutes when the number of records is increased to 55.

With VMCrypt, when the database size is $10,000$, the entire protocol terminates after 19 seconds. Moreover, each party (and the Java virtual machine that executes it) uses only 0.15 GB memory (4.8% of 3 GB). Even if we increase the database size to 1 million (which increases the size of the circuit to 100 million gates), the parties still use the same amount of memory.

VMCrypt has gone through rigorous performance analysis. To show that it scales on circuits with a wide structure we tested it on the `DBSearch` circuit. To show that VMCrypt scales regardless of wiring patterns, we implemented the set intersection circuit, which has a highly connected structure. To show that VMCrypt scales on wide and deep circuits, we implemented the minimum circuit. To show VMCrypt modularity, all circuits have been developed and tested without modifying VMCrypt source code. To show that performance is truly linear, each circuit was tested on inputs of increasing length (as opposed to breaking the input into small pieces and modifying VMCrypt source code to evaluate several smaller circuits). To show scalability we evaluated circuits with hundreds of millions of gates. As our tests show, VMCrypt would scale on any circuit with any number of gates (e.g., billions). See Section 5.

1.3 Related Work

Secure computation has been used recently to implement systems for privacy-preserving face recognition, fingerprint matching, and DNA processing (c.f., [13, 7, 28, 24, 1, 8]). Other optimizations for garbled circuits, such as the free XOR technique [15] and others [17, 14, 27, 26, 6] has been proposed. Techniques for secure computation based on homomorphic encryption (c.f., [25, 5, 10, 30, 29]) have also been studied.

Fairplay [21, 2] demonstrated the feasibility of two-party secure computation based on garbled circuits. Automation and benchmarking of cryptographic protocols were studied in, e.g., [3, 4, 18, 23].

The comparison of VMCrypt to Fairplay applies also to TASTY [11]. TASTY combines the benefits of homomorphic encryption and garbled circuits. It uses a high level language, which requires recompilation and rules out executing secure protocols on the fly. TASTY does not provide an API (indeed, this was not a design requirement), and therefore integrating it into other applications requires modifying TASTY source code. TASTY would run out of memory on any large circuit. In fact, TASTY does not deal with large circuits, except for one example where memory consumption climbs as the circuit reaches 4.2 million gates, and crashes thereafter.

Prior to this work, we showed that faster execution times can be achieved by constructing circuits from Java classes [9]. The software from [9] was not designed to provide an API, and it does not include any of the innovations presented

in this paper. Similarly, it does not scale and it requires the use of storage. Specifically, it crashes on large circuits and exhibits non-linear running time, problems which we avoided in [9] by manually instantiating many small circuits and rewriting the source code for each circuit.

2. OVERVIEW OF VMCRYPT

This section gives a high level overview of VMCrypt. Components are described in detail in the next section.

Once a component has been written, VMCrypt provides the developer with a test tool to validate that the component indeed computes the correct function. This tool, implemented in class `TestModule`, creates an instance of `StandardInput`, which holds the input bits. Similarly, it creates an instance of `StandardOutput`, and assigns the output bus of the component to point at it. After the component computes, its output is stored in the `StandardOutput` object. If the component implements interface `Testable`, then the test tool can automatically validate the component for correctness. The test tool iterates over all inputs to obtain a full functional coverage of the component. The advantage of our tool over Fairplay is that a component can be validated on all inputs, without having to run the secure protocol.

The preceding discussion gives the impression that the component is a passive description of a circuit, and that the test tool somehow processes this description. This impression is wrong. The entire work is done by the component, and the same applies to garbling and evaluation. We describe this process in more detail.

Each VMCrypt binary gate has a method `notify` that takes three arguments: *a port, an object*, and *a function*. In the case of the test module, the function is class `Calculate`, and the object, which represents an input, is a bit (0 or 1). The port, which is 0 or 1, tells the gate which input wire is receiving the object. Once both inputs arrive, the gate invokes the `compute` method of the function, passing these inputs as arguments. Next, the gate notifies its output wire with the output of this function. This process continues until the entire circuit is computed. The process of garbling and evaluating a circuit uses the same `notify` method, except that different functions and objects are passed as arguments.

VMCrypt presents a new variant for Yao's protocol, in which the circuit is garbled in chunks. We use a `Notifier` (see Figure 1), which has a very simple task: to obtain a segment (shown as a bracket) of pairs of wire labels, and for each pair in the segment, notify the component with this pair. More precisely, the notifier notifies the component with an object and a function that gates pass execution to once they have all their inputs ready. On the server side the function is `Garble` and the object is a `WireLabelPair`. On the client side the function is `Eval` and the object is `WireLabel`. Both functions implement abstract class `Function`.

Once the notification process begins, gates receive objects on all of their input wires and pass execution to the function. In the case of the garble function, gates receive wire label pairs, lookup tables are written into the network, and gates notify their output wires with a wire label pair. In the case of the evaluation function, gates receive wire labels, lookup tables are decrypted, and gates notify their output wires with wire labels. The notifiers repeat this, segment by segment, until the entire component is notified. Immediately after the server sends the last lookup table, the client computes the output. The component builds and destructs parts of its description during the notification process, which we describe in the next section.

Oblivious transfer is not executed before or after the garbling; it is interleaved in the garbling. What makes this possible is abstract class `WireLabelTransport`. VMCrypt provides a server side implementation for this class called `WLTPServer` (*wire label transport protocol, server side*) and a client side called `WLTPClient`. The role of the wire label transport protocol is to guarantee that when the notifier on the server side receives a segment of wire labels, the notifier on the client side already has them. This enables the server to stream lookup tables, knowing that the client will be able to decrypt them on the fly and discard them immediately. The obvious implication of this streaming is that no storage is necessary for either wire labels or lookup tables. This is a significant advantage because the garbled circuit can be very large. Moreover, time consuming read/write operations are eliminated, thus reducing running time. A

We briefly sketch the proof of security. The idea is identical to the classic proof [16]. The simulation begins when the simulator $\mathcal{S}$ sends the evaluator $\mathcal{E}$ a garbled circuit with a random output. Of course, the circuit is sent in chunks, and OT is interleaved in the garbling. After all OT operations are over, $\mathcal{S}$ learns the input of $\mathcal{E}$. It then queries the ideal functionality, learns the real output, and sends $\mathcal{E}$ a permutation that maps the random output to the real output. Since the ideal and the real views of $\mathcal{E}$ are computationally indistinguishable, the protocol is secure in the honest but curios setting.

In the rest of this section we describe the wire label transport protocol parameters, starting with the OT sub protocol. VMCrypt provides two OT implementations [22, 12], but of course any OT protocol can be passed as an argument. The second parameter is an implementation of interface `WireLabelGenerator`. The role of this interface is to provide wire label pairs for the wire label transport protocol. A standard implementation of this interface would simply return two random strings as a `WireLabelPair`, but VMCrypt implements the "free XOR" idea [15] and therefore our implementation of this interface produces a pair $\langle r, r \oplus R \rangle$, where r is a freshly chosen random string, and R is a fixed random string. To use standard garbling, all that one needs to do is provide the standard implementation for `WireLabelGenerator`, with the corresponding implementations of `Function` for garbling and evaluation, and pass them as arguments to the respective classes.

The third parameter to the wire label transport protocol is a class that implements interface `CircuitInput`. For any i, this interface provides a method that answers whether wire i corresponds to a server input or not, and another method that returns the value on this wire (0 or 1). The wire label transport protocol uses the `CircuitInput` interface to find which input wire belongs to which party, and what is the value of this input. Implementations of `CircuitInput` play another important role: if a party has an input that is too large to hold in memory (e.g., a database), then `CircuitInput` can read it from its origin segment by segment, as opposed to loading it all at once in the beginning.

To avoid expensive OT per wire, we execute OT in bulk. However, recall that we interleave OT in the garbling, which means that we do not know in advance how many (and which) wire labels require OT. To overcome this, we implement what we call *an OT bucket*. Specifically, in the wire

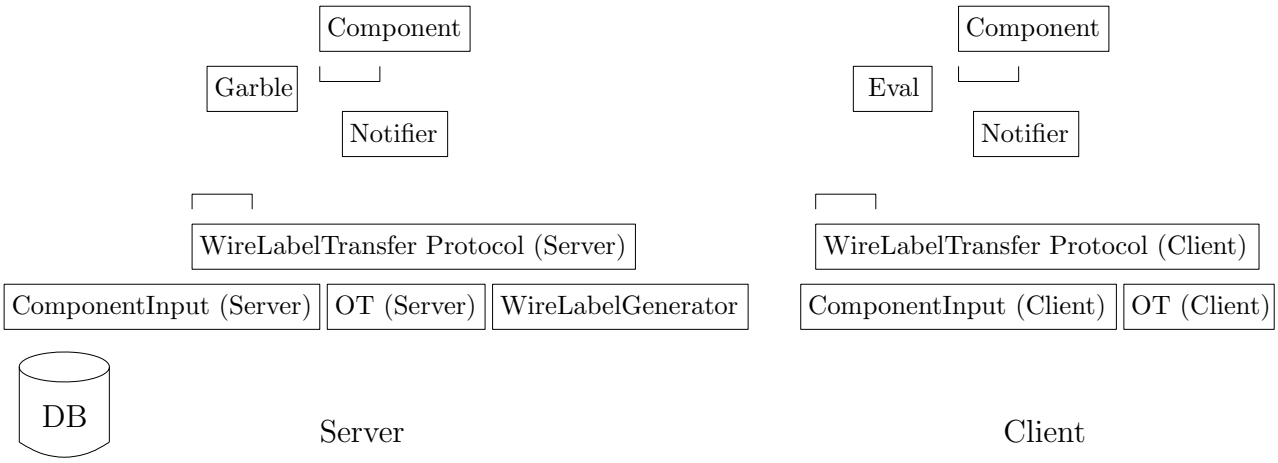


Figure 1: Overview of main VMCrypt modules

label transport protocol (server side) the OT bucket contains pairs of random strings $\langle r_{j,0}, r_{j,1}\rangle$, as opposed to wire label pairs. On the client side it contains one string $r_{j,b}$ per pair, where $b \in \{0,1\}$ is the client input at wire index j. Wire labels are transferred as follows. If $\langle W^{i,0}, W^{i,1}\rangle$ is a wire label pair corresponding to a server input b, then $W^{i,b}$ is sent to the client. Otherwise, $\langle W^{i,0} \oplus r_0, W^{i,1} \oplus r_1\rangle$ is sent, where $\langle r_0, r_1\rangle$ is the pair of masks at the top of the OT bucket. This works because the wire label transport protocol examines future segments to find what the client will need when future wires are reached. When the bucket is empty, the OT protocol is invoked to refill it.

Earlier in this section we described the test tool, which validates that a component indeed computes the function at hand. This may not be sufficient for debugging, because a developer may want to see how a signal passes through a circuit. The developer may also want to collect statistics, like network traffic, or number of gates. To support this, VMCrypt provides class `Monitor`, which is a tool for visualizing the execution of VMCrypt modules. The monitor allows developers to see what is happening inside components and protocols in real time. Modules report events by calling monitor method `report`. They pass the event itself and a string that includes values of relevant variables. The monitor displays the `ID` of the component with each report. The `ID` is a pair of integers `(depth,index)`. When sub components are built, they are assigned `depth` that is one higher than their enclosing component, and a unique `index` within this component. By providing the `ID`, VMCrypt helps developers identify which part of the component is reporting. Events are implemented in classes `ComponentEvent` and `ProtocolEvent`. They describe useful information such as when components build and destruct, or when protocols start and end. There are currently thirty one events in VMCrypt, and they can be extended of course. Events also provide runtime statistics, such as network traffic, execution times, and execution progress.

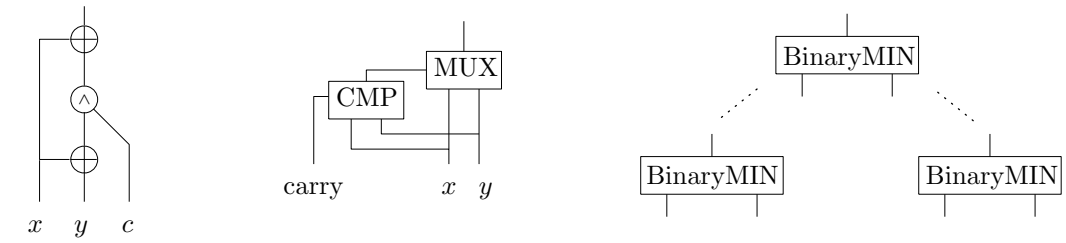
3. THE COMPONENT MODULE

The component module is the heart of VMCrypt. It has been redesigned four times from scratch to guarantee that performance does not depend on circuit topology. We start with the original design and explain how it evolved to the current version.

First version. The first version of the component module was inspired by the minimum function, which plays a central role in privacy-preserving systems [7, 28, 24, 1]. We defined two types of components: `Gate` and `Circuit`. We also provided wires to connect them. Our idea was that developers would be able to construct small circuits, and then use them as building blocks for larger circuits. For example, a bit multiplexer (Figure 2 a) would be built in a low-level manner, from gates and wires. A string multiplexer, on the other hand, would be built from the bit multiplexer. Next, a multiplexer and a comparator can be used to build a circuit for finding the minimum of two numbers (Figure 2 b), which is then used to build a circuit for finding the minimum of N numbers (Figure 2 c). This approach is similar to programming in $C++$ in the sense that developers have both the low-level power of C and the modularity provided by an object-oriented language. Another advantage of this approach is that developers can share components or replace them with better implementations.

To instantiate a circuit for finding the minimum of N numbers whose bit length is $arity$, VMCrypt developers use the following standard Java syntax: `Component c = new MIN`$(N, arity)$. The problem is that, even for modest values $N = 1,000,000$ and $arity = 64$, the number of wires needed to connect the `BinaryMIN` circuits is $3 * arity * (N-1) = 194$ million wires (excluding wires and gates inside the `BinaryMIN`). Moreover, wires are implemented as lists (class `Vector` in Java) to allow fan out degree higher than 1. Thus, the memory needed to instantiate the `MIN` circuit is prohibitive. The code of this version was used in [8].

Second version. The objective of the second version was to provide components for wireless circuit design. We added a new component, called a `Switch`, that can route signals to its sub components without wires. The idea is simple: when the switch receives a signal and a wire index (called *a port*), it compares the in-degree of the first sub component with the port. If the port is smaller, then the sub component receives the signal. Otherwise, the in-degree is subtracted from the port, and the switch iterates on the next sub component (of course, like all other mechanisms,



(a) BitMUX (b) BinaryMIN (c) MIN

Figure 2: Components of the MIN circuit due to [14, 28]

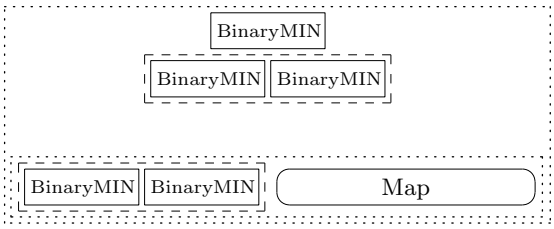


Figure 3: The `MIN` component.

this is automatically taken care of by the component and is transparent to the developer).

The switch brings up a difficult dilemma that is general to all the components. That is, should we equip a switch with output wires or not? Without output wires components will not be able to connect to each other. With them we create the same amount of wires that we wanted to eliminate in the first place. Our solution was to replace output wires of components other than gates with one pointer, and allow input wires only in circuits. Thus, gates will still be able to send signals to multiple components, but non-gate components will only be able to send signals to one component. In the rare case where this would be too restrictive, we provide a `Splitter`, which enables routing to multiple components. This dilemma represents a conflict between utility, usability, and efficiency, that predominated throughout the entire development process.

The other problem with the switch was that routing a signal requires linear time (as opposed to constant when using wires). This becomes quadratic per component, and increases the running time considerably. To solve this problem we introduced the `UniSwitch`. This component contains multiple copies of the same sub component. Thus, it can route a signal in constant time by dividing the value of the port by the in-degree of this sub component. The `UniSwitch` turned out to be very useful as it enables constructing components that mimic a `for` loop.

Let us use these components to build the `MIN` component without wires. As shown in Figure 3, this component has a `BinaryMIN` circuit at the root, and a `UniSwitch` containing `BinaryMIN` at each layer, except the base, which is a `Switch` containing two sub components: a `UniSwitch` and a component called a `Map`. The map requires almost no memory as it contains nothing inside. When it receives input on port i, it simply outputs it on port $m(j)$, where m is the function implemented by the map (the identity in this case). The map is used here to pass the signal one level up in case that the tree is not perfect. Notice that all the components reside inside a `Switch` whose input layer is the base layer.

Having built `MIN`, let us revisit the java statement `Component c = new MIN`$(N, arity)$, which motivated all the new components in the second version. This statement no longer creates millions of wires. However, since all the `BinaryMIN` circuits are made of wires and gates, the `MIN` circuit still requires an unacceptable amount of memory.

Third version. The third version of the component module introduced the `Bus`. If VMCrypt has one most important module, the bus would be it. The bus has a very simple task: it counts the number of signals leaving the component. When this number reaches zero, the bus invokes the `destruct` method of the component. This removes the component from memory. All VMCrypt components have a bus, except for gates, which self destruct.

Clearly, there is no point in having a bus if the entire component resides in memory; the bus has a value only if components are built *after* their instantiation. Thus, all VMCrypt components (except gates) build when they receive their first signal, and destruct immediately when they output their last signal. We call this *the notification principle*. To see the power of this principle, notice that even when a component is built, its sub components will not be built until they receive input, and this of course applies recursively. Consequently, VMCrpyt components have a very lean memory footprint.

Let us see how much memory the `MIN` component (Figure 3) requires now. The first signal will build the base layer, which contains two components, but only the `UniSwitch` will build because the `Map` received no signal yet (assuming the signal comes from the left). Inside the `UniSwitch`, only the leftmost `BinaryMIN` circuit is built because the other ones received no inputs yet. Because the `BinaryMIN` is also built from sub components, only those that receive this signal will be built, and so on. Eventually, the signal will hit a gate, who will store it and wait for the other signal (assuming a binary gate). When this signal arrives, the gate computes, destructs, notifies its output wire, and the process continues. When the `BinaryMIN` fires its last output, it is destructed by its bus. In the case of a `UniSwitch`, even the reference to this object is removed from memory. Hence, the `UniSwitch` from the base layer becomes empty. Since each layer notifies the one above it, the `Switch`, which contains all these layers, requires $\log(N)$ references to components of type `UniSwitch` that are either empty or contain one `BinaryMIN`. Notice that

this is not a flaw in VMCrypt; it is due to the underlying recursive algorithm we are using. To further improve the notification process we added a `Buffer` to components of type `Circuit`. The buffer will build the circuit and flush only after all signals have arrived. Overall, this reduces the amount of time components (both inside and outside the circuit) reside in memory as well as the number of these components.

The notification principle confronted us with the following problem. Consider the `MIN` circuit from Figure 2 c. The `BinaryMin` circuits (Figure 2 b) that make up this circuit take a carry bit (either 0 or 1) as input. Thus, the moment this value becomes available, all of them will be notified and hence built. The amount of memory required for this build is like having the entire `MIN` circuit in memory, which is clearly unacceptable. In our component for set intersection we faced the same problem. There, we have multiple copies of a sub component for set membership, and all of them are notified with the same set. The issue with our design is that ports have no control over when they receive notifications.

Fourth version. The fourth version introduces *pulling* and *pushing* ports. A component with a pulling port does not need to be connected to another component in order to receive its signals. Instead, it pulls them from a global *pulling-pushing table*, called `PTable`.

The most important aspect of pulling ports is that they can control when they want to be notified. For example, in the scenario mentioned above with the `MIN` circuit (Figure 2 c), the `BinaryMin` circuits will not build when the carry bit becomes available. Instead, they decide when to pull this value from the table. Of course, pulling is useless without pushing and therefore any output port can push a value into the table. In the `MIN` circuit scenario, for example, the carry bit will be initially pushed into the table, and components will pull it from the table.

The table presents several design challenges. Consider, for example, a component that has only pulling ports (we call it a *pulling component*). According to the notification principle, this component will never get notified and hence never built. The question is therefore who will notify this component and when. We decided that this will be specified by the enclosing component. That is, a component specifies pulling ports, pulling sub components, and a counter. The component increases its counter on each notification. If the counter equals to the in-degree of the component, then all pulling ports and pulling sub-components are notified. Thus, pulling can occur before, after, or while the component is receiving notifications. There is another issue. Recall that the role of the bus is to destruct a component once the component fired its last output. If some signals go to the `PTable` instead, the bus would not know about them and therefore never destruct the component. This problem has a recursive nature because components are nested. We solved this problem by allowing these *pushing signals* to continue to notify buses. Components, however, can ignore them.

We did not conceive situations where a port pulls a value that has not been pushed yet, but to allow this flexibility the table stores references to components making such requests and notifies them as soon as the value becomes available. We decided that only components (but no other object) will push values into the table. This preserves modularity because, by including pulling and pushing information in the component, developers can freely exchange components.

To see how a `PTable` works in practice, consider the VMCrypt component for set intersection from Figure 4. Intuitively, this component pushes the set $X = \{x_1, \ldots, x_{m-1}\}$ of the server into the table, and then each y_i from the client set $Y = \{y_1, \ldots, y_n\}$ is tested for membership in X by pulling X from the table. In detail, the component contains two sub components of type `UniSwitch`. The first `UniSwitch` pushes X into the table using m copies of the identity map (recall that only components can push. It turns out that `Map` is ideal for this purpose). The second `UniSwitch` contains n copies of a `Switch` for set membership. This `Switch` contains three sub components. The first is a map that pushes y_i into row m of the table. The second is a `UniSwitch` that contains m copies of the equality component `EQ` (of type `Circuit`). The j-th `EQ` circuit pulls to its leftmost ports the value stored in the table at row m (initially y_0) and to its rightmost ports the value stored in the table at row j (which is x_j). That is, it checks whether y_i and x_j are equal. Thus, the `UniSwitch` containing the `EQ` circuits outputs a sequence of m bits that are all 0 if and only if $y_i \notin X$. Because this output is fed into an `OR` component (of type `Circuit`), the i-th `Switch` for set membership will output 1 if and only if $y_i \in X$.

How much memory did the `PTable` save? Suppose that we compute set intersection on databases of size $N = 1,000,000$ and that for efficiency we hash set elements into 20 byte strings (that is, $arity = 160$ bits). This requires $N(N * arity) + N * arity$ wires. The minimum memory cost for a wire is 8 bytes (a reference to the wire plus a reference in the wire itself). Thus, we would need a fantastic amount of 320 terabytes RAM.

Let us examine what happens in memory when the set intersection component computes. The component is first notified with the bits of x_0 and therefore only the `Map` exists fully in memory. There is nothing inside the map, but notice that now we also need to store information about which ports are pushing. Since all ports are pushing, this requires an array of size $arity$, which is the bit length of set elements. Once the map fires its last signal it is destructed by its bus and the `UniSwitch` in which this map resided becomes empty again. This repeats until $x_0, \ldots, x_{m-1}$ are pushed. Now the table contains the entire set X. This is not a flaw in VMCrypt; it is a consequence of the underlying algorithm we are using for set intersection. Next, we notify the component with the bits of y_0. As before, this causes the map to build and push y_0 into the table. After the map destructs, the `UniSwitch` containing the `EQ` circuit is built. Next, the first equality circuit builds, computes, destructs, and the `UniSwitch` containing it becomes empty again. The output of the `EQ` notifies the `OR` component. Since the `OR` was implemented as a circuit and circuits have a buffer, the output of `EQ` is stored in this buffer and the `OR` is not built yet. After the last `EQ` circuit produces its output, the `UniSwitch` containing the `EQ` destructs and the buffer of the `OR` is full. This causes the buffer to build the `OR` and flush all the signals into it. The `OR` computes, destructs, and sends its output to its enclosing component, the `Switch`. Since the out-degree of the `Switch` is also 1, it also destructs. Now we are at the same state as in the beginning because all the components inside the set intersection component are empty. This entire process repeats with $y_1, \ldots, y_{n-1}$, and finally the set intersection component destructs.

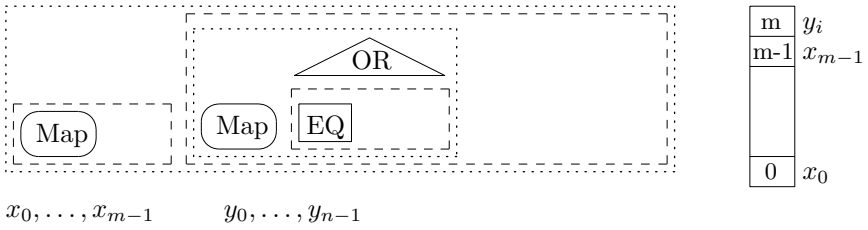


Figure 4: The set intersection component implemented in VMCrypt.

4. CREATING A NEW COMPONENT

This section describes how to create a VMCrypt component. For lack of space, we only provide an overview. See [20, 19] for details.

We start with an overview of the component hierarchy (Figure 5). The root of this hierarchy is `BaseComponent`. Objects of this class have an `ID`, which is a pair of integers `(depth,index)`. The only role of the `ID` is in validation. Class `Gate` is extended by two classes: `BinaryGate` for binary gates and `UniGate` for unary gates. All gates receive their type (e.g., XOR, OR, NOT) during instantiation, and their output wire (initially empty) is also created during instantiation. The classes `Map, UniSwitch, Switch` and `Circuit` are derived from `Component`. The main difference between a `Component` and `Gate` is that the former has a `Bus`, which also implies that it is built only when notified, whereas the later has an output wire and is built when instantiated.

Components are created in VMCrypt by writing a new Java class. Whether created from existing components or from scratch, developing new components is 100% modular - it requires no access to any source code. To create a new component, we need to choose a sub class of `Component` to extend, and the type of sub components. Notice that we can also add new building blocks to VMCrypt by deriving directly from `Component`, whereas in Fairplay the language is final and cannot be extended.

Class Map. A map is a component that maps an input from one port to another. A map only needs to implement abstract method `map`. For example, the identity map code is as follows:

```
class IDMap extends Map {
  IDMap(Bus bus, int inDegree) {
    super(bus, inDegree);
  }
  int map(int port) {
    return port;
  }
  public String name() {
    return "IDMap";
  }
}
```

The `name` method returns the name of the component and is used only for testing purposes. The `notify` method, which handles notifications from other components, is implemented in class `Map`. Notice that components like `Map` are not the only notifiable modules; a wire, a bus, and other modules also have a `notify` method.

Like all sub classes of `Component`, the constructor of `Map` passes three arguments to the constructor of `Component`: `inDegree, outDegree` and a `bus`. Those are data members of `Component`. In the case of `Map`, the value of `inDegree` is passed both as `inDegree` and `outDegree`, which is why our constructor for `IDMap` only takes `inDegree` as an argument.

Class Component. The most important data member of `Component` is the bus. We illustrate the role of the bus using the minimum component (Figure 3). The base layer of this component is a `Switch` with two sub components: a `UniSwitch` and an `IDMap`. Figure 6 shows this `Switch` with its two subcomponents (the `BinaryMIN` inside the `UniSwitch` are not shown). Each component has a `Bus` shown as a grey rectangle. As we mentioned above, all sub classes of `Component` must pass to the constructor of `Component` an instance of class `Bus`. This instance stores a reference (called `out`) to a notifiable object that will receive the output of the component. In the case of the minimum circuit, for example, we first instantiate the layer above the base layer (call it L), then we create a new bus that will point at it by writing `Bus bus = new Bus(`L`)`, and finally we invoke the constructor of the `Switch` with `bus` as argument. This bus is shown in Figure 6 at the top left corner of the `Switch`. In addition to directing the output, the bus is responsible for destructing a component once it has fired its last signal. For this, the bus maintains a counter. The only role of the `outDegree` variable is to set this counter. Consider now the sub components of the `Switch`. Their output is intended to L, yet their busses point at the bus of the `Switch`. Why? because the bus of the `Switch` must be able to count all the signals leaving the component. If some signals are missing, then it will never destruct the `Switch`.

Let us see how the bus works in practice. Suppose that the first signal the `Switch` from Figure 6 receives is on port 0. It will pass it to the `UniSwitch`, who will process it and fire a new signal on, say, port 0. If the `outDegree` of the `UniSwitch` is, say, 8, then the counter in the bus of the `UniSwitch` is now $8 - 1 = 7$. Similarly, if the `outDegree` of the `Switch` is, say, 16, then the bus of the `Switch` updates its counter to $16 - 1 = 15$. Of course, it also passes the signal from the `UniSwitch` out (to the layer above), on the same port it was received, which is 0. The type of bus that we use in the `UniSwitch` would not work in the `IDMap`. To see why, consider what happens when the `Switch` routes signals to the `IDMap`. Suppose that the `inDegree` of the `UniSwitch` is 16 and that the `Switch` receives a signal on port 16. Since 16 does not fall in the range $0 - 15$, the `Switch` will pass it to the `IDMap` on port $16 - 16 = 0$. Since the map is the identity function, the bus of the `IDMap` will pass this signal to the bus of the `Switch` on port 0, which will override outputs of the `UniSwitch`. Thus, when we instantiate the `IDMap`, we pass to it an `OffsetBus`, which offsets the port.

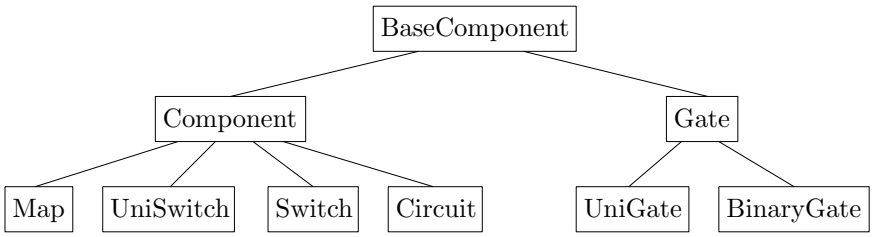


Figure 5: Class hierarchy of components.

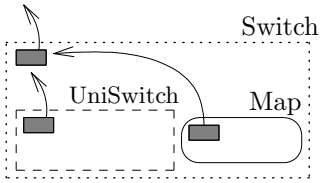


Figure 6: The role of the Bus.

In this scenario we want the output of the `IDMap` appearing immediately to the right of the output of the `UniSwitch`, and since the `outDegree` of the `UniSwitch` is 8, we instantiate this bus by writing `OffsetBus offsetBus = new OffsetBus(`$bus, 8$`)`, where bus is the bus of the `Switch`. Any output of the `IDMap` on port i will now be given to the bus of the `Switch` on port $8 + i$.

Unlilke `outDegree` and `Bus`, the role of `inDegree` depends on the component and the sub component. In a `Map`, the `inDegree` and `outDegree` are always the same, but `inDegree` is never used. Similarly, `Switch` and `UniSwitch` never use their own `inDegree`. However, the `Switch` uses the `inDegree` of its sub components for routing. This is why all classes derived from `BaseComponent` have an `inDegree()` method. The `UniSwitch` takes two parameters: `inputTab` and `outputTab`. When a signal is received on port i, it is given to the sub component at index $index = \lfloor i/\texttt{inputTab} \rfloor$. If this sub component outputs a signal on port j, then the bus of the `UniSwitch` will output it on port $index * \texttt{outputTab} + j$. Class `Circuit` is the only component that uses its `inDegree`: when a circuit is built, it instantiates `inDegree` objects of class `Wire` and stores them in an array of size `inDegree`.

Circuits. Class `Circuit` is the only component with input wires. Like in a `Switch`, the sub components of `Circuit` can be of any type derived from `BaseComponent`. To create a circuit in VMCrypt, we need to implement two methods: one that instantiates sub components and another that connects them. As an example, we implement the bit multiplexer from Figure 2 a. The skeleton of our class is:

```
class BitMUX extends Circuit<BinaryGate> {
  BitMUX(Bus bus) {
    super(bus, 3,1);
  }
  public String name() {
    return "BitMUX";
  }
}
```

The numbers 3 and 1 are the `inDegree` and `outDegree` of the component, respectively. Class `Circuit` has an array called `components`. When a circuit receives its first input, its `define_sub_components()` method is called. In the case of the `BitMUX`, for example, this method will initiate `components` to be `XOR` and `AND` gates. Next, class `Circuit` will initialize its `inputWires` array to size `inDegree` (which is 3 in our case), create empty wires, and invoke method `connect_wires()` to connect wires to gates.

Once building blocks like `BitMUX` are built, higher level circuits can be built more easily. For example, our `MUX` on strings of length $arity$ instantiates $arity$ elements of `BitMUX` in `define_sub_components()`, and connects their input and output wires in `connect_wires()`.

Classes Switch and UniSwitch. Creating a component from classes `Switch` and `UniSwitch` is even simpler than class `Circuit`. In the case of `UniSwitch`, we only need to implement a method returning an instance of the sub component. In a `Switch`, on the other hand, each sub component can be from a different class, so like in `define_sub_components()`, we instantiate all the sub components, and connect them using a `Bus`, or a `Splitter` when necessary.

5. PERFORMANCE

We analyze VMCrypt performance on circuits which have been developed without modifying VMCrypt source code. Each circuit has different structural properties and was tested on inputs of increasing length to show that performance does not depend on size or structure. We are not aware of such rigorous analysis in prior work.

All tests were executed on a Thinkpad X301 laptop with 3 GB RAM and a 1.6 GHz Intel Core2 Duo processor running Ubuntu Linux. Two Java Virtual Machines (JVM) were run on this computer; one for each party. To guarantee that performance is measured in a standard setting, we did not configure the JVM (e.g., by increasing the size of the heap). Parties communicated through the loopback network interface. Running the parties side by side means that the same machine was stressed with twice the amount of work, for both TCP (packet handling, checksums, etc.) and Yao's protocol.

Encryption was implemented with SHA-1 modeled as a random oracle, which provides much better performance compared to number theoretic based pseudo-random generators (PRGs). Wire label length was 120 bits. We remark that our communication complexity is exactly four encryptions (one lookup table) per gate and one OT per wire. This is the absolute minimum. By maintaining party state, we avoid sending metadata (describing what is being sent), gate numbers, wire index, etc. Thus, any other protocol can only

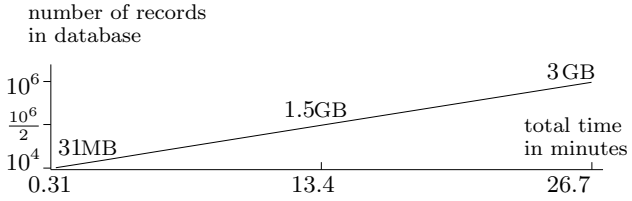


Figure 7: Running time of secure database search.

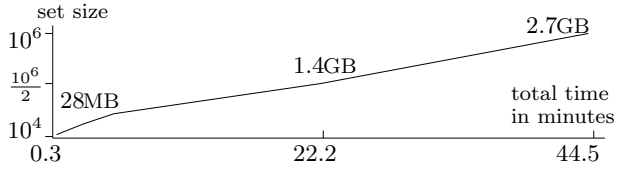


Figure 8: Running time of secure minimum.

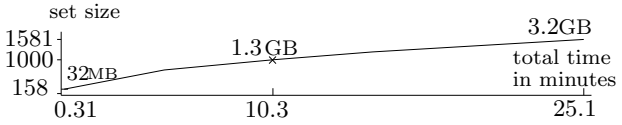


Figure 9: Running time of secure set intersection.

have equivalent or worse communication complexity. Formally, the communication complexity is exactly $4n*L+k*|x|$ bytes, where n is the number of non-XOR gates in the circuit, L is the byte length of a ciphertext (encryption), $|x|$ is the length of the input of the evaluator, and k is a small constant that depends on the OT communication complexity.

We start with the database search function. The input to this function is an array of records $\langle x_i, p_i \rangle$ and a string y, where x_i and p_i are viewed as columns in a database table. The output is all p_i for which $x_i = y$. In VMCrypt it is implemented as component `DBSearch`. In Fairplay [21] it was implemented as *Keyed Database Search (KDS)*.

Since Fairplay circuit parameters are passed at compile time, the compiler must be run in each execution. Thus, we compared the performance of our full protocol with that of the Fairplay compiler [21] alone (ignoring the time and memory it takes to actually execute the Fairplay protocol). Since the time and memory complexity of the Fairplay compiler are not reported in the literature, we carried our own test on a table where the length of each of x_i, p_i and y is 20 bits. When the table size is $N = 20$ records, the compiler runs for 10 seconds. When $N = 40$ the running time is 50 seconds. Obviously, this is not linear. When $N = 55$ the compiler runs for 3 minutes and then crashes with a Java `Out Of Memory` Error. In the case of $N = 40$, memory consumption climbs gradually to 23% of the RAM. When $N = 55$, it reaches 26%.

In VMCrypt, when $N = 10,000$ the running time of the secure database search protocol is 19 seconds. Figure 7 describes the running time (in minutes) and communication complexity for values of N up to 1 million. The component with 1 million records has 100 million gates and 60 million lookup tables (the number of non-XOR gates).

We discuss Figure 7 and other statistics. First and foremost, although the full hardware description of the circuit for database search is linear in N, the client and the server each used the same amount of memory, namely, 4.8% RAM. Secondly, the running time is linear in the database size. This was not the case with earlier versions of VMCrypt, and therefore should not be taken for granted (see discussion in Section 3). Thirdly, the very large amounts of data sent confirm that parallelizing Yao's protocol yields significant savings on disk read and writes. Finally, 84% of the running time was consumed by cryptographic operations and communication (this was measured by comparing with the running time of calculating the component). In other words, VMCrypt overhead was only 16% . Other implementations also have an overhead, but unfortunately such statistics are not provided in the literature.

Our next test was the minimum component, implemented in VMCrypt as class `MIN` (Figure 3). This component takes N integers (of length 20 bits each) and outputs their minimum. In our experiment, half of the integers belong to the client and half belong to the server. We remark that, in practice, parties will have shares of the integers, and the function would first add the shares and then find the minimum.

The running time (in minutes) and communication complexity of the protocol for secure minimum are given in Figure 8. These are linear in N because, as with the `DBSearch` component, the underlying (non-secure) algorithm runs in linear time. On average, parties used 5% of the memory (independently of N), and VMCrypt overhead was 10%. Notice that the running time is almost double that of the `DBSearch` component. This is because half of the inputs required oblivious transfer, and the number of lookup tables was almost double (the `MIN` component with 1 million inputs had 110 million gates and 90 million lookup tables).

Our last test was the set intersection component, implemented as class `SetIntersection` (Figure 4). This component takes two sets of size N and M, and outputs their intersection. We fixed the bit length of set elements to 20 bits. To simplify the presentation we chose $N = M$. The running time (in minutes) and communication complexity of the protocol for secure set intersection are given in Figure 9.

Notice that, unlike previous tests, the size of the set intersection component is quadratic in N due to the underlying (non-secure) algorithm it implements. For comparability with previous tests, we chose values of N ranging from 158 to 1581. Thus, when $N = 158$ we have $N^2 \cong 10^4$ and the component has 1 million gates and 0.5 million lookup tables. Similarly, when $N = 1581$ we have $N^2 \cong 10^6$ and the component has 100 million gates and 50 million lookup tables. On average, parties used 5% of the memory (independently of N), and VMCrypt overhead was 15%.

6. REFERENCES

[1] M. Barni, T. Bianchi, D. Catalano, M. D. Raimondo, R. D. Labati, and P. Faillia. Privacy-preserving fingercode authentication. In *MM&Sec'*, Roma, Italy, 2010. ACM.

[2] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security*, pages 257–266, 2008.

[3] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

[4] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Public Key Cryptography*, pages 160–179, 2009.

[5] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Public Key Cryptography*, pages 119–136, 2001.

[6] I. Damgård and C. Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In *CRYPTO*, pages 558–576, 2010.

[7] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *Privacy Enhancing Technologies*, pages 235–253, 2009.

[8] D. Evans, Y. Huang, J. Katz, and L. Malka. Efficient privacy-preserving biometric identification. In *Proceedings of the 17th conference Network and Distributed System Security Symposium, NDSS 2011*.

[9] D. Evans, Y. Huang, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. To appear in the 20th USENIX Security Symposium.

[10] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[11] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: Tool for automating secure two-party computations. In *ACM Conference on Computer and Communications Security*, 2010.

[12] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, pages 145–161, 2003.

[13] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *IEEE Symposium on Security and Privacy*, pages 216–230, 2008.

[14] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS*, pages 1–20, 2009.

[15] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP '08: Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, pages 486–498, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[17] Y. Lindell, B. Pinkas, and N. P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *SCN*, pages 2–20, 2008.

[18] P. D. MacKenzie, A. Oprea, and M. K. Reiter. Automatic generation of two-party computations. In *ACM Conference on Computer and Communications Security*, pages 210–219, 2003.

[19] L. Malka. VMCrypt 1.4 developers manual. *http://www.lior.ca/publications/VMCrypt_Manual_Rev1.0.pdf*.

[20] L. Malka. VMCrypt - modular software architecture for scalable secure computation. *EPrint report 2010/584*, 2010.

[21] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *SSYM04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[22] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA*, pages 448–457, 2001.

[23] J. D. Nielsen and M. I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *PLAS*, pages 21–30, 2007.

[24] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. Scifi - a system for secure face identification. In *IEEE Symposium on Security and Privacy*, pages 239–254, 2010.

[25] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT'99: Proceedings of the 17th international conference on Theory and application of cryptographic techniques*, pages 223–238, Berlin, Heidelberg, 1999. Springer-Verlag.

[26] A. Paus, A.-R. Sadeghi, and T. Schneider. Practical secure evaluation of semi-private functions. In *ACNS*, pages 89–106, 2009.

[27] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, pages 250–267, 2009.

[28] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *ICISC*, pages 229–244, 2009.

[29] N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography*, pages 420–443, 2010.

[30] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *EUROCRYPT*, pages 24–43, 2010.

[31] A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.