# JavasCrypt

JavasCrypt is a proof of concept software system, built in HTML/CSS and JavaScript, which is meant to be used for building diagrams of cryptographic algorithms which are actually functional. This software is built on the joint.js JavaScript library, made by Rappid, which is a comprehensive solution for building flowcharts and flow diagrams using only your web browser.

The JavasCrypt project has a number of files, here are their details:

|  |  |
| --- | --- |
| **Filename** | **Description** |
| backbone.js | This file forms the backbone of the joint.js JavaScript library. This file was never edited for the purposes of this project. |
| joint.js | This is the actual library which is used when making nice, connectable flowchart diagrams. This file was never edited for the purposes of this project. |
| joint.shapes.logic.js | This file contains the specifications for all of the logic blocks. This file was retrieved from the logic gate demonstration found on the Rappid website. This file was edited heavily to include many new 'logic blocks' of my own creation. |
| jquery.js | The always lovable jQuery library. |
| lodash.js | Lodash is another javascript library which joint.js takes advantage of. |
| fullaes.html | This file contains the full, 10-round implementation of AES. |
| fulldes.html | This file contains the full 16-round implementation of DES, excluding the initial and final permutations. |
| hello.html | This file was downloaded from the Rappid website and was where I got the code for how to write logic blocks with HTML elements. |
| logic.html | This was the initial proof of concept. This page shows how an addition block would work, an output block, and how the numeric value of a block can be dynamic. |
| logic2.html | This is the implementation of one round of AES. |
| logic3.html | This is the implementation of one round of DES. |
| logic.css | This file included the styles of all the wires and html elements being used. It was later edited and specialized for the purposes of AES and DES. |
| logic2.css | This file is used to set the styles for the AES logic blocks. |
| logic3.css | This file is used to set the styles for the DES logic blocks. |

# How logic blocks are added to JavasCrypt

## Logic Blocks and Not Gates

The basic idea behind how new logic blocks are made in joint.shapes.logic.js is through a form of inheritance, called extension. For example, this is the code for creating a not gate:

joint.shapes.logic.Not = joint.shapes.logic.Gate11.extend({

defaults: \_.defaultsDeep({

type: 'logic.Not',

attrs: { image: { 'xlink:href': '**REMOVED FOR BREVITY**' }}

}, joint.shapes.logic.Gate11.prototype.defaults),

operation: function(input) {

return !input;

}

});

joint.shapes.logic is a bag of objects. By saying joint.shapes.logic.Not = *blahblahblah* we are adding an object called Not to joint.shapes.logic. This Not object is an extension of the Gate11 object, which is a generic block for logic gates with one input and one output.

Creating logic gates with an arbitrary number of inputs and outputs is possible by creating the appropriate Gate object first, similarly to how Gate11 and Gate21 are created, and then extending this gate into a specific type of functional logic block.

Within the extend() function there are some default values specified as well as an operation. The default values include the type of gate (which should be the last two specifiers after joint.shapes) as well as the attrs attribute, which is where the SVG for the gate is placed. This is a base64 encoded SVG, there are online applications for converting an SVG into base64 for use in this code, or explicit links to SVG files can be used as well. For the sake of brevity, the base64 encoding was removed, as it was multiple pages long. For ease of use and readability, a path to a png file can be specified, instead of a base64 string which encodes it. Png files are used throughout this project, as they were easy to create and edit using MS-Paint, and made the code easier to manage.

joint.shapes.logic.Gate11.prototype.defaults is used to specify the rest of the default settings for this Not gate.

Finally, there is an operation attribute, this is where the actual function which the gate computes is performed. In general, for DES, we should expect the input to be any integers. For AES, however, these inputs may actually be arrays of numbers. A modification of logic3.html was necessary in order for these inputs to be able to be arrays, as the code was initially designed with the intent of only passing around integer values.

## Addition and Gate21

Here is an example of addition, which is an extension of Gate21:

joint.shapes.logic.Sum = joint.shapes.logic.Gate21.extend({

defaults: \_.defaultsDeep({

type: 'logic.Sum',

attrs: { image: { 'xlink:href': '**REMOVED FOR BREVITY**'

}}

}, joint.shapes.logic.Gate21.prototype.defaults),

operation: function(input1, input2) {

return input1 + input2;

}

});

As you can see, the code is almost identical to that for the Not gate, but the operation now takes two inputs.

The code for Gate21 is as follows:

joint.shapes.logic.Gate21 = joint.shapes.logic.Gate.extend({

markup: '<g class="rotatable"><g class="scalable"><image class="body"/></g><circle class="input input1"/><circle class="input input2"/><circle class="output"/></g>',

defaults: \_.defaultsDeep({

type: 'logic.Gate21',

attrs: {

'.input1': { ref: '.body', 'ref-x': -2, 'ref-y': 0.3, magnet: 'passive', port: 'in1' },

'.input2': { ref: '.body', 'ref-x': -2, 'ref-y': 0.7, magnet: 'passive', port: 'in2' },

'.output': { ref: '.body', 'ref-dx': 2, 'ref-y': 0.5, magnet: true, port: 'out' }

}

}, joint.shapes.logic.Gate.prototype.defaults)

});

Which is a bit more complex, but still fairly straightforward. As you can see, the markup has to be specified, which corresponds to the HTML generated. This markup will need to be extended according to the number of inputs and outputs. Further, we can see that the inputs and outputs are specified here by giving them names and attributes in the attrs section. First the position of the ports are specified using ref, ref-x, ref-dx, ref-y and ref-dy (not shown here). Next, the 'magnet type' is specified. This specifies whether the port is input ('passive') or output (true). Lastly, the ports are given names.

If you take a look at the code for Sum, you can see that the operation takes two inputs input1 and input2. These are exactly the inputs which are created in the Gate21 code.

Now take a look at Gate11:

joint.shapes.logic.Gate11 = joint.shapes.logic.Gate.extend({

markup: '<g class="rotatable"><g class="scalable"><image class="body"/></g><circle class="input"/><circle class="output"/></g>',

defaults: \_.defaultsDeep({

type: 'logic.Gate11',

attrs: {

'.input': { ref: '.body', 'ref-x': -2, 'ref-y': 0.5, magnet: 'passive', port: 'in' },

'.output': { ref: '.body', 'ref-dx': 2, 'ref-y': 0.5, magnet: true, port: 'out' }

}

}, joint.shapes.logic.Gate.prototype.defaults)

});

There is only one input, called input. This is exactly the input that the operation for the Not gate is using.

## How Signals Are Propagated

There are at least three different sections of code in logic\*.html, fulldes.html and fullaes.html, in which signals can be propagated from inputs to outputs.

The first section is the initializeSignal() function. This function is invoked at the start of the program and every time a port is connected to or disconnected from. In logic.html the initializeSignal() function has a block of code towards the bottom which looks like this:

\_.each(graph.getElements(), function(element) {

// broadcast a new signal from every input in the graph

(element instanceof joint.shapes.logic.Input) && broadcastSignal(element, signal);

(element instanceof joint.shapes.logic.Four) && broadcastSignal(element, 4);

(element instanceof joint.shapes.html.Element) && broadcastSignal(element, element.get('input'));

});

This code loops through the entire graph and looks for Input elements and broadcasts the signal to their output ports. By adding the next two lines I was able to broadcast custom signals from my logic blocks. The middle line always broadcasts a value of 4 from the output of my Four block, and the last line broadcasts the value which is contained in the input HTML element of the HTML logic block.

Another very useful method for broadcasting signals involves overriding the onSignal property of these logic blocks. An example of this can be seen in the Signal block in logic.html:

// Signal displays the actual signal on the wire

joint.shapes.logic.Signal.prototype.onSignal = function(signal) {

this.attr({

text: {text: signal}

});

}

These functions are automatically invoked whenever an input signal changes. In this code we are merely updating a CSS attribute of the Signal block in order for it to display the signal on the wire, but it is possible to go further and perform an operation on the signal and make use of the broadcastSignal() function in order to propagate the answer to the output ports of the logic block.

An example of this can be seen in fulldes.html where the signal handler for the Round block is overridden in order to generate DES round keys. These overrides were generally used due to technical difficulties: if logic blocks required non-numerical inputs to function (such as an array of ints), then the operation specified in joint.shapes.logic.js would not function appropriately. This was eventually solved by editing a line in the final section of code where signals are broadcast.

The last section of code where signals are propagated from is also the section where the logical operations specified in joint.shapes.logic.js are applied. This section of code exists at the bottom of these HTML files and specifies what happens when there is a change in the signal. When a signal changes on a wire the logic gate which this wire is connected to (if any) then has its logic operation applied to the new signal. Initially these signals were expected to only be numeric, and so a Math.max() operator was being applied to these signal values in order to round the signal to a 0 or a 1, as is desirable when performing Boolean logic. By removing this Math.max() operator, unadulterated signals, including arrays of ints, were able to be forwarded along to the logical operations in joint.shapes.logic.js.

## How the Logic Blocks Are Added to the HTML Canvas

Take a look near the bottom of the logic.html file and you'll see this code

var gates = {

four1: new joint.shapes.logic.Four({ position: { x: 100, y: 90 }}),

four2: new joint.shapes.logic.Four({ position: { x: 100, y: 220 }}),

sig: new joint.shapes.logic.Signal({ position: { x: 440, y: 155 }}),

sum: new joint.shapes.logic.Sum({ position: { x: 240, y: 150 }}),

el1: new joint.shapes.html.Element({

position: { x: 550, y: 55 },

size: { width: 120, height: 60 }

})

};

This is where the gates are specified. The gates must be given a name (e.g. four1, four2, sig, etc.) and then a new logic gate is instantiated using our previously created logic block code. There is no need to make a constructor or anything, just specify the structure of the logic block in joint.shapes.logic.js and you'll be good to go. The only thing the constructor normally takes is an intial position of the logic gate on the canvas.

Now, there's one more catch. If you look at the code above, you'll see that not every logic block is contained in joint.shapes.logic. The last block, el1 (which is our numeric input block), is of type joint.shapes.html. This is because this block is fancier than a standard logic block, it has an active html element built into it which defines its value.

## How HTML Logic Blocks Are Created

The following code is found in fullaes.html and specifies the structure of the RoundKey generation block:

joint.shapes.html.RoundKey = joint.shapes.logic.IO.extend({

markup: '<g class="rotatable"><g class="scalable"><image class="body"/></g><circle class="input"/><circle class="output"/></g>',

defaults: joint.util.deepSupplement({

type: 'html.RoundKey',

attrs: {

'.wire': { 'ref-dx': 0, d: 'M 0 0 L 23 0' },

'.input': { ref: '.body', 'ref-x': -5, 'ref-y': 50, magnet: 'passive', port: 'in' },

'.output': { ref: '.body', 'ref-dx': 35, 'ref-y': 100, magnet: true, port: 'out' } }

}, joint.shapes.logic.IO.prototype.defaults)

});

This initial portion of the code specifies the RoundKey element much like you would specify a logic gate in joint.shapes.logic.js. We include some markup here so that input and output ports will be available for this HTML block, since it is not extending a Gate.

After this code is a large block of code which specifies the view of the HTML element, which is called joint.shapes.html.RoundKeyView. This view includes an attribute called template which specified the HTML code which the HTML block will use:

template: [

'<div class="html-element">',

'<input class="roundkey" type="number" value="0" min="0" max="10"/>',

'<label class="roundkey-text">Roundkey</label>',

'</div>'

].join('')

This is where the various input elements, labels, and HTML styles are specified. After the template attribute is an attribute called initialize. The initialize element is where signals from the interactive HTML elements are propagated from the view to the model. There is code of the following form:

this.$box.find('.roundkey').on('change', \_.bind(function(evt) {

this.model.set('roundkey', $(evt.target).val());

}, this));

This code selects the value from the roundkey input element and sets the 'roundkey' attribute of the model to have this value. This allows us to retrieve the HTML input values later on when performing logical operations on them.