MDEV ISSUE

Dr. Dobb's Journal

JUNE 2011

Next

.NET Development

A Plugin-Based Graphical App in C# Using MEF and Embedded IronPython

ALSO INSIDE

UBM TechWeb

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

June 2011

COVER STORY

More on DrDobbs.com

The jQuery Plugin Mechanism

The jQuery library has many merits and has changed the way we code client web pages. One of its best selling points is the integration of the CSS query syntax with the page DOM. Another nice move is the chain syntax, which allows developers to concatenate multiple calls in a single line. A natural extension of the chain syntax is the plugin mechanism, through which you can add new capabilities to the core library.
http://drdobbs.com/web-development/229403172

Survey Shows Unethical Behavior Rampant Inside IT Development Teams

Ethically doubtful actions and dysfunctional behaviors make it difficult for developers to be trusted by users and other IT staff.
http://drdobbs.com/architecture-and-design/229402654

Endless Projects: Where's the Fun Anymore?

How can you create a non-trivial project that can be completed in a reasonable amount of time?
http://drdobbs.com/architecture-and-design/229402673

Another Fine Jolt

This month, we've kicked off a new season of Jolt awards. In this incarnation, there will be six product categories with awards announced on a bi-monthly cycle: architecture & design, books, coding tools, libraries, testing, and utilities.
http://drdobbs.com/229403175

Making Two-Way Tables in iOS

Although iOS has several Cocoa Touch classes that specialize in displaying text, the class of choice for creating tables that present a two-way flow of information is UITextField.
http://drdobbs.com/mobility/229402045

Mailbag

Virtual Machines and Endless Projects

Using More VMs

In response to the editorial "Use More VMs on Your Laptop," http://drdobbs.com/tools/229402296, in which editor Andrew Binstock argued that virtual machines should be much more widely used on desktops that the limited use cases currently found in development.

All Programs Should Run in VMs

You are right in saying that for development there is absolutely no reason not to utilize VMs, but I would desperately like to see the day when the development of a program integrates the architecture of a VM as a foundation of its function. I mean that each program is automatically sandboxed within a VM-(obfuscated to the end user by a host operating system that is little more than a display of the available VMs — read "apps"). Essentially automating the process and practice you and others do for the web sphere and applying it to every program.

It may mean programs are larger than we are used to seeing, and most may end up being composed out of a DSLinux system, but for the most secure and most stable system where every program is sure to run, and a virus only means that you close that instance of the running program and open it again…

— Jason Greer

Your Entire System Should Run in a VM

I agree with all your points with respect to the use of VM's. This is why I have have been using VM's for development right from the early days on laptops and desktops. One added benefit of using VM's is with respect to hardware/system upgrades. Buy a new laptop or desktop and all you have to do is install the VM software and then just copy the VM files over from the older machine. No fuss and no muss. It's also easy to do backups also by just copying the VM files to backup storage.

Keep up the good work.

— Stevyn R. Dembo

Andrew Binstock responds: "I am not sold on this approach, although I agree with the benefits your list. There are two problems: The backup you suggest is non-specific. A small change in one file requires you to back up the entire VM. Secondly, this approach precludes some of the advantages listed in the editorial, because you cannot run a VM within a VM (or where you can, you shouldn't). So, where virtualization delivers benefits, you have to sit on the sidelines."

Endless Projects: Where's The Fun Anymore?

This editorial, http://drdobbs.com/229402673, opining on how large even small projects have become, received a flood of com-

miseration and suggestions. Most comments proposed using new languages, including Delphi (from Lane Campbell), FoxPro (Anthony Scarpelli), Rebol (Robert Münch), CoffeeScript (Michael Swaine), Clojure (by a commenter named Adde), and the Grails framework (T. Valesky). In addition, several commentators stated that Binstock was on the right path looking at Groovy and Ruby.

Andrew Binstock responds: "Thanks for these thoughtful suggestions. As I paused to consider my options, I decided to take 30 days to dive into Clojure to see whether functional languages might provide either the tools or the insights I could use. Follow my progress, if you like, on Twitter (@platypusguy). As to the larger question, I fear that the next letter stating that it's not principally a language issue is probably right."

It's Not the Language

I feel the same, unfortunately. I don't think you will find your peace of mind with Groovy, Ruby, or any other current language/framework. What is really hurting you (I guess) is repeating yourself, well, maybe not yourself explicitly, but following yourself (your steps).

After several years of writing simple routines, programs, algorithms, I wish they would be finally written for good, and I could focus on the bigger picture. How many times should I deal with problems with enabled buttons, with systray icons, with compressing files, with hashing, cloning data? — oh, this one is good.

We (developers) fell into a trap of needing more and more sophisticated achievements, yet there are not enough developers to make complete, reliable, robust languages and frameworks. With every new, "revolutionary" product, we face the same problems we hoped would be solved by now.

— Maciej Pilichowski

Have a correction or a thoughtful opinion on Dr. Dobb's content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of Dr. Dobb's.

COMING SOON TO DRDOBBS.COM

Four-Part Series: Xtext Tutorial
The next installment of our series on Cloud Programming

Real-Time Java

Microsoft's Fickle APIs

Silverlight faces Redmond's penchant for new APIs.

Microsoft is a company built by developers. Unlike IBM, Oracle, Apple, and other competitors, Microsoft came to the operating system and application business via selling software development tools. Programming is wired into its DNA.

These roots are reflected in how Redmond provisions products. Developers enter into a closed world where internal connections are directed at other Microsoft products. It's a world where the entire ecosystem comes exclusively from Microsoft. To reach this goal, the company has at times undermined competing platforms and stretched standards. To wit, Microsoft paid Sun Microsystems $1.6 billion to settle a lawsuit claiming Microsoft undermined Sun's Java platform. Microsoft dropped its J# product. In other arenas, it uses marketshare to apply torsion to standards. No clearer example of this is Internet Explorer, which in versions 5.0 through 8.x, made running standard HTML next to impossible. All Web developers know the frustration fixing valid HTML so that it renders correctly in IE.

This tradition of creating new standards that fit the company's strategic objectives has led to numerous new APIs and pressure on developers to migrate to them. Even in the early days of Windows, the company's fondness for new APIs cost us dearly. The transition from 16-bit Windows to Win32 initially required a choice between one of three API subsets. Ultimately, two variants faded away, leaving just Win32. Then Microsoft published the Microsoft Foundation Classes, an entirely new framework that dominated Windows programming until .NET.

Migrating code to .NET was difficult and the new languages Microsoft provided generally weren't backwards compatible. To use the new features, working applications had to be substantially modified or rewritten. This was particularly painful for Visual Basic programmers. The conversion from Visual Basic 6 to VB.NET meant brooking numerous syntactical language changes for which Microsoft provided limited assistance.

Making it across the river Jordan to .NET, however, didn't spare us from a new series of APIs. We still had to choose which platform to use for projects: WinForms, Windows Presentation Foundation, and Silverlight being the principal client-side choices.

"More likely, Microsoft will ride the RIA wave with continued investment in Silverlight and sufficient commitment in HTML5 to be part of that market"

This history of constant forced migration raises the question of whether Microsoft will abandon Silverlight now that HTML5 is increasingly the center of the rich Internet application universe. Given that IE 9 provides extensive HTML5 support, this concern is particularly apposite.

While it's a fool's errand to predict corporate behavior, we know something of Microsoft's Silverlight plans. Version 5.0 is in a lengthy beta test and is expected to ship mid-year. Longer term, Silverlight looks like a keeper: Microsoft has invested substantially in making it run on Mac OS X and on the handheld version of Windows. It also invested in a Linux version, called Moonlight, jointly developed with Novell. It seems unlikely that Microsoft would walk away from this investment, especially now that RIA applications are a hotbed of activity.

More likely, Microsoft will ride the RIA wave with continued investment in Silverlight and sufficient commitment in HTML5 to be part of that market. HTML5 — a specification controlled by an independent group — isn't historically where Microsoft would go. The company tends to provide programming layers over which it can exert control. I believe that if HTML5 becomes the standard everyone is expecting, Microsoft will leverage it via Silverlight, thereby preserving its technology while integrating with the larger world.

— Andrew Binstock is Executive Editor for Dr. Dobb's and can be contacted at alb@drdobbs.com.

A Plugin-Based Graphical App in C# Using MEF and Embedded IronPython

Using .NET 4.0's MEF to facilitate a plugin architecture turns out to be fairly easy. So is embedding IronPython. Doing both has its challenges

By Gigi and Saar Sayfan

Graphr is a Windows Presentation Foundation-based GUI program implemented in C# that allows you to generate a dataset based on a rule (expression) such as "(x-3) * (x-5)" for a given range of values. Graphr then displays a graph on the screen. Graphr is interesting because it is a polyglot application. It is a C# application that hosts an embedded IronPython engine. It allows you to write your rules in Python and even use any function from Python's math library, such as *sin()* or *cos()*. The best part is that you don't need to parse and evaluate the expression. Python will do this for you. The other interesting aspect of Graphr is its architecture. Graphr is a plugin-based system. It utilizes the Managed Extensibility Framework (MEF) to load graph plugins dynamically and use them in a totally decoupled fashion. You can add additional graphs and Graphr will be happy to automatically use them without touching Graphr itself. This is a useful practice that can be applied in many situations.

In this article, we will demonstrate Graphr, explain in detail how to embed IronPython in any application, and explore MEF and its use in a plugin architecture. In a separate article, which you can access immediately at http://drdobbs.com/windows/229402979, we've done a walkthrough of Graphr's architecture. Finally, we'll talk a little about the future of Graphr.

Graphr in Action

Before we delve into the code, let's have some fun playing with Graphr. You need Visual Studio 2010 (http://www.microsoft.com/express/Downloads/) and IronPython 2.7 (http://ironpython.net/download/). Both are free (Visual Studio 2010 has non-free versions, too, but they are not needed to build Graphr).

When you launch Graphr, you see a window divided into two panes. The left pane contains the rule and range text boxes and a "Visualize"

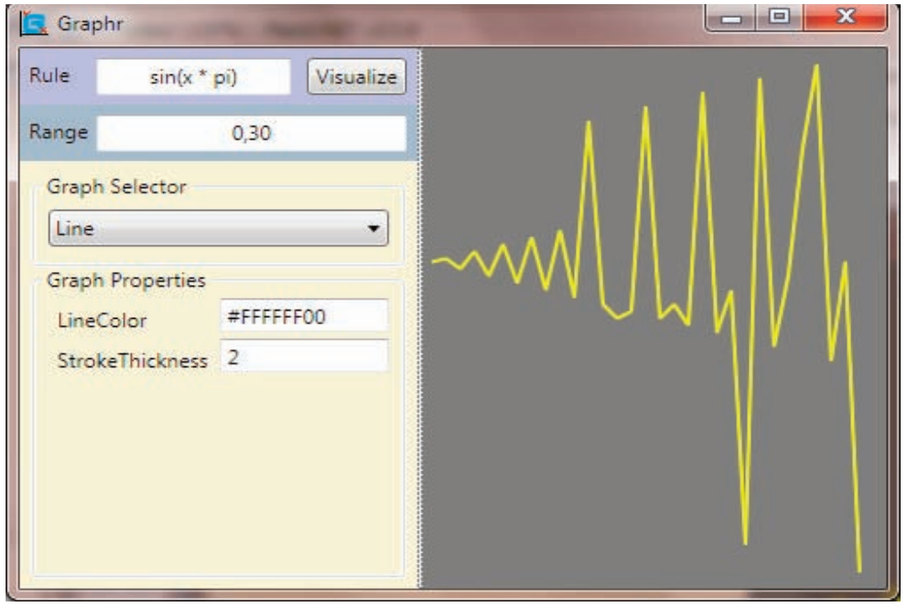

Figure 1: A typical example of Graphr in use.

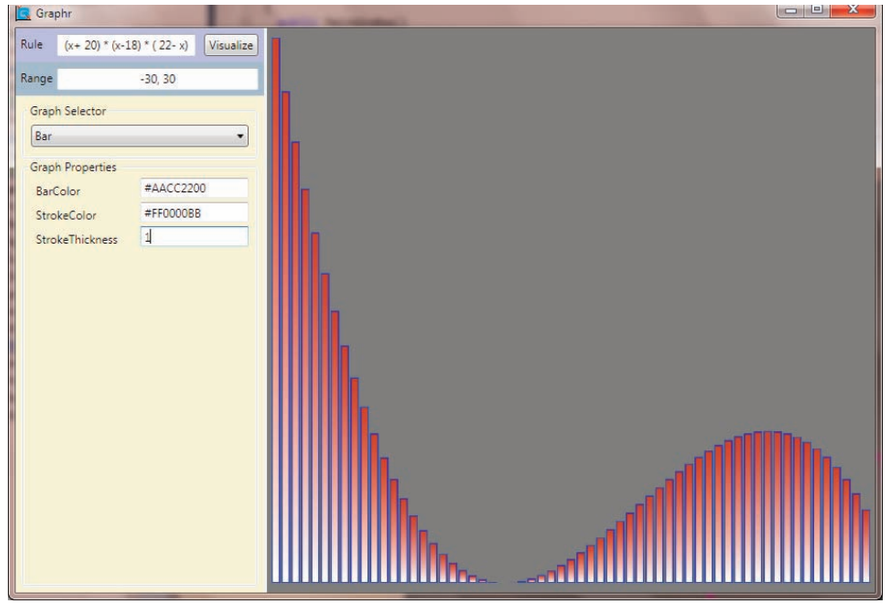

Figure 2: Graphr displaying a bar graph.

button on top, and a graph selector and graph properties on the bottom. The right pane (empty initially) contains the graph display area. Click the "Visualize" button and a yellow line graph is displayed (Figure 1).

The graph properties area allows you to customize the appearance of the graph. The LineColor text box contains four double-digit hex numbers (*A, R, G, B*). The first two digits control transparency (*00* fully transparent, *FF* fully opaque), the other digits control the red, green, and blue values of the color. The LineThickness text box controls the thickness of the line.

You can also change the rule and the range. When you change the rule and/or the range, you must click the "Visualize" button again. The graph selector dropdown box allows you to select a different graph type. Figure 2 shows a bar graph that uses a different rule and a range that includes negative numbers. Note that each graph type has its own graph properties.

The bar graph uses a gradient brush to paint the bars, so the bar color changes gradually from the selected color at the top to white at the bottom of the bar.

Graphr is also fully resizable. You can resize the entire window and the graph will adjust and fit into the new size. You can also change the ratio between the left and right pane by dragging the splitter left and right.

Graphr also supports a pie graph, which displays slices that are proportionate to the values of the rule. The color of each slice is selected randomly, and you can only control the border color. If you play with the pie graph, it is recommended that you use a range of up to 50 values because it takes longer to render (especially if you resize the window).

Embedding IronPython

Graphr is able to parse and evaluate complicated expressions. Implementing a custom parser and evaluator in C# is possible, but would take a long time. Instead, we opted to take advantage of Python, which provides out of the box dynamic evaluation capability.

IronPython is an open source implementation of Python for the .NET platform. IronPython 2.7, which is what we used in Graphr can be downloaded at http://ironpython.codeplex.com/releases/view/54498.

Dynamic Evaluation with IronPython

IronPython makes it easy to dynamically evaluate any valid Python expression and, in particular, mathematical expressions. The key is the built-in *eval()* function, which takes a Python expression as a string, executes it dynamically, and returns the resulting Python object. Here is a short interactive session that demonstrates it:

```
>>> eval('5 + 5')
10
>>> x = 3
>>> eval('x + 2')
5
>>> x = 'hello'
>>> eval('x + " world"')
'hello world'
>>>
```

Graphr needs to evaluate an expression that contains an *x* variable multiple times, where *x* is assigned different values from the range. The Evaluatr.py Python module provides this functionality. It imports all the functions in Python's math library so they can be evaluated as part of the expression. It contains two functions: *evaluateSingleValue()* and *evaluate()*. The *evaluateSingleValue()* function gets an expression (rule) and an *x* value. It substitutes all the *x*'s in the expression with the value of *x* and evaluates it, as shown here:

```
import os
from math import *

def evaluateSingleValue(expr, x):
  # replace the 'X' in the expression with the current value
  e = expr.replace("x", str(x))

  # evaluate the substituted expression string using the eval() function
  return eval(e)
```

The *evaluate()* function is used directly by Graphr. It accepts an expression and a list of values. It iterates over the values and evaluates each one using *evaluateSingleValue()*. Note that *evaluate()* is a Python generator, which means that values are computed only as they are needed. The *yield* keyword returns control to the caller, which allows efficient streaming in case of large ranges that don't need to be fully evaluated (such as a graph that displays a window of 100 values regardless of the entire range), as shown next:

```
def evaluate(expr, values):
   """Evaluate an expression with an X variable over a range of X values

   expr - a string that represents a Python expression with a variable
   values - a list of floating point values

   return a generator that yields pairs of (x, eval(expr))
   """
   expr = expr.lower()
   for x in values:
     try:
       yield (x, evaluateSingleValue(expr, x))
     except:
       yield (None, None)
```

Embedding IronPython in a C# Program

Now, that we have a good Python evaluator that can dynamically evaluate Python expressions, it's time to embed it in your C# program. IronPython is built on top of the DLR (Dynamic Language Runtime). The

DLR exposes hosting APIs designed to support this use case. We isolated all embedding code into a single C# class called *IronPythonEvaluator*. The *IronPythonEvaluator* relies on the Evaluatr.py module to perform the actual computation. This raises the question of how to deploy the Evaluatr.py module with the Graphr app. The most straightforward approach is just to provide it as an additional file, but we chose a more integrated approach where the source code for the module is embedded as a resource in the Graphr.exe assembly (Figure 3).

When the *IronPythonEvaluator* is initialized it loads the resource and saves it to a file. This file is used later by the IronPython runtime engine.

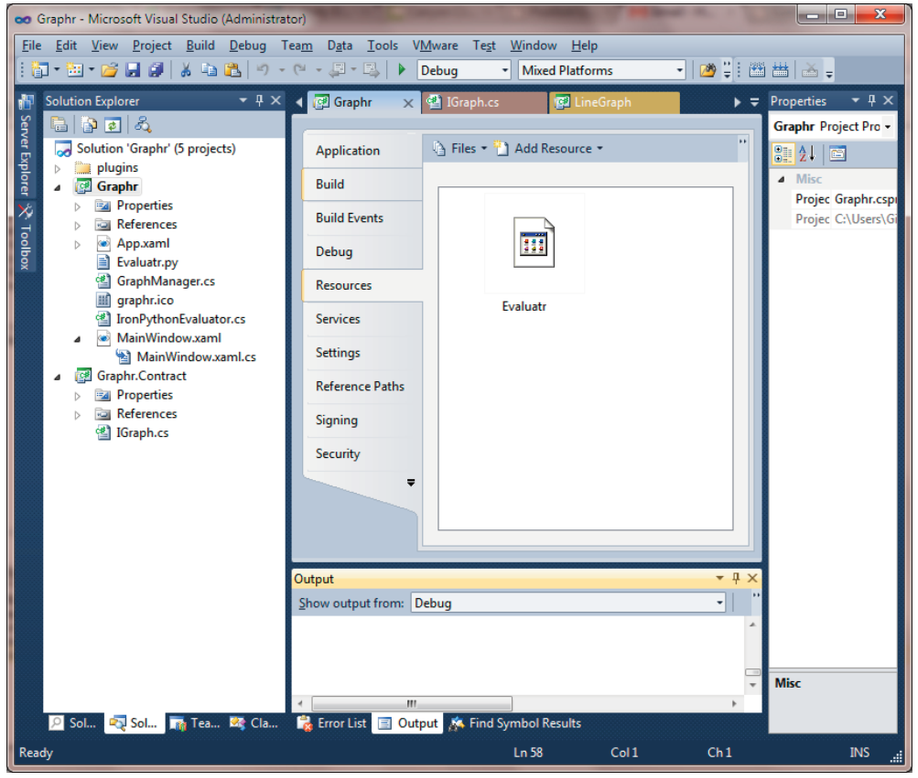

Figure 3: Embedding the IronPython module in the C# program.

Before we dive into the *IronPythonCalculator*, here are the assemblies it uses:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

// The following assemblies are needed to host IronPython
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;

using Graphr.Properties;
using System.IO;
```

The *IronPythonEvaluator* has a constructor and two overloaded *evaluate()* methods. The constructor creates an IronPython engine with a "Debug" option. This is useful because it allows stepping from C# into IronPython code when debugging in VisualStudio. Also, the debugger will present you with a unified call stack from C# to Python if an exception is raised in the Python code (Figure 4). This polyglot debugging is a unique feature of the .NET platform.

The constructor then adds the IronPython Lib directory to the engine's search path. That allows IronPython to import any standard library module. The next step is to write the embedded Evaluatr.py module to a temporary file, then load this file into the engine's runtime and assign the resulting Python module into the *evaluatr* dynamic member. Once the file has been loaded into the runtime, it can be deleted. Note, that this code is not industrial strength and doesn't handle exceptions so the temporary file will not be deleted if an exception is thrown.

```
namespace Graphr
{
  class IronPythonEvaluator
  {
    public IronPythonEvaluator()
    {
      //Creating a new script runtime
      Dictionary<string, object> options = new Dictionary<string, object>();
```

```
options["Debug"] = true;
var env = Python.CreateEngine(options);

// Set the default search path for IronPython (not set by default)
var searchPaths = env.GetSearchPaths();
```

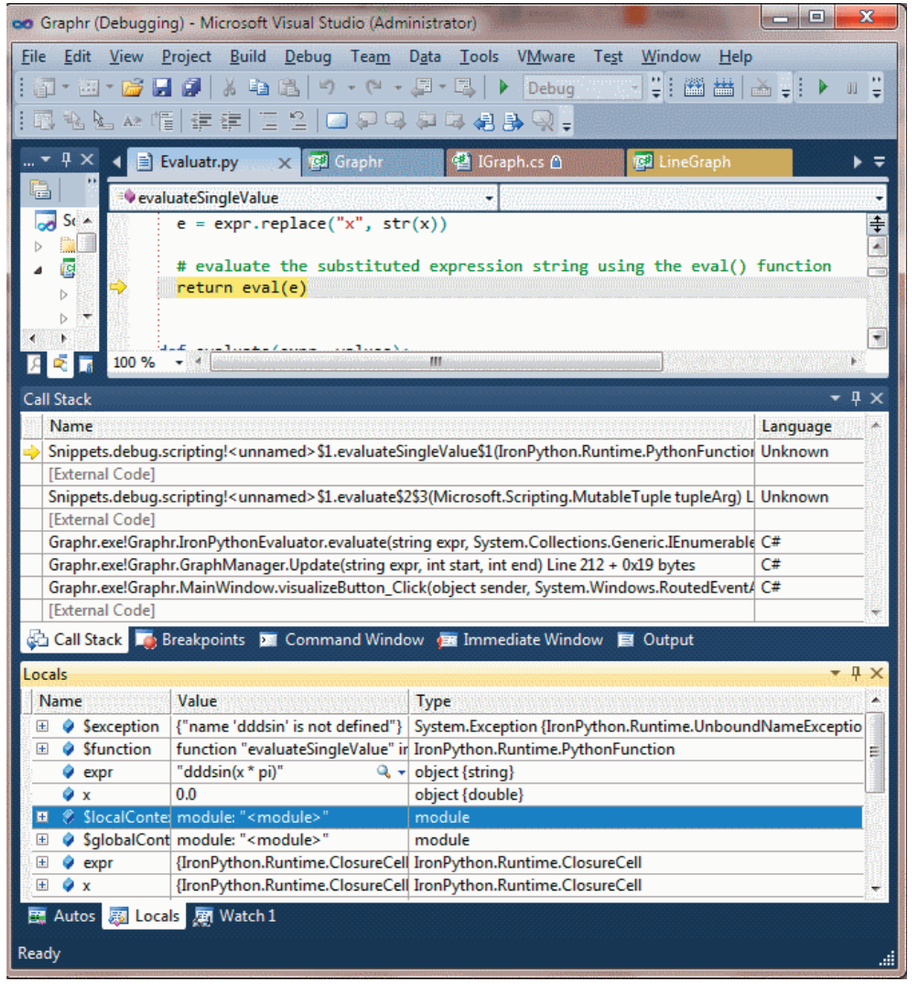

Figure 4: Visual Studio presents a unified call stack from C# to Python, when IronPython is embedded.

```
// Note: this is hard-coded path to the IronPython 2.7 lib directory
var p = "C:/Program Files (x86)/IronPython 2.7/Lib";
if (!Directory.Exists(p))
{
    p = "C:/Program Files/IronPython 2.6 for .NET 4.0/Lib";
}

searchPaths.Add(p);
env.SetSearchPaths(searchPaths);
var filename =  Path.GetTempPath() + "Evaluatr.py";
// Write the Evaluatr.py script (loaded from resource)
using (var fs =
  new FileStream(filename, FileMode.Create, FileAccess.ReadWrite))
{
  BinaryWriter bw = new BinaryWriter(fs);
  bw.Write(Resources.Evaluatr);
}

evaluatr = env.Runtime.UseFile(filename);

// Delete the file
File.Delete(filename);
}
```

The main *evaluate()* method has the following signature:

```
public IList<KeyValuePair<double, double>>
evaluate(string expr,
         IEnumerable<double> values)
```

It accepts a mathematical expression as a string and a list of doubles (the *X* values). It returns a list of pairs of doubles (the *X,Y* pairs).

The *evaluate()* method starts by declaring an empty result list and calling the *evaluate* method of the embedded IronPython *evaluatr* module:

```
var result = new List<KeyValuePair<double, double>>();

//Invoke the method
var r = evaluatr.evaluate(expr, values);
```

The return value *r* is a dynamic object, which actually contains an *IronPython.Runtime.PythonGenerator* object. This is not the type of object we want to expose to the rest of the program (the idea is to hide

the implementation). So, we iterate over the generator and push every pair to the result list.

If the expression is invalid, then an IronPython exception is raised (in Python, you raise exceptions instead of throwing them), and we translate that to a C# exception, which we then throw, as shown next:

```
foreach (var s in r)
{
  try
  {
    double x = s[0];
    double y = s[1];
    var pair = new KeyValuePair<double, double>(x, y);
    result.Add(pair);
  }
  catch (Exception)
  {
    throw new Exception("Invalid expression: '" + expr + "'");
  }
}

return result;
```

The other *evaluate()* method is pretty much the same but it accepts integer values. Internally it converts them to *double*s and calls the main *evaluate()* method:

```
public IList<KeyValuePair<double, double>>
       evaluate(string expr, IEnumerable<int> values)
{
  // Convert integers to doubles
  var xValues = values.Select(i => (double)i);

  // Call the evaluate() method that operates on doubles
  return evaluatr.evaluate(expr, xValues);
}
```

Managed Extension Framework (MEF)

MEF was released as part of the .NET framework 4 and Silverlight 4. It provides a standard way for applications to discover and load extensions and to expose application services to extensions. In addition, the application extensions may depend on each other.

The main parts of MEF are the catalog and the composition container. The catalog is responsible for locating composable parts, and the container is responsible for creating parts and their dependencies (which are other parts). Parts have import and export contracts and they interact with other parts using these contracts.

MEF provides an attribute-based programming model, which is very easy to use and yet provides good error messages when things go wrong. For further MEF information, we recommend http://mef.codeplex.com/documentation.

Plugin-based Applications

We prefer the term "plugin" instead of "application extension," so that's what we'll use from now on. My definition of a plugin is a piece of code that has an interface and is loaded by some application dynamically at runtime; then, the application interacts with the plugin through the interface. The application may provide certain services to the plugin though its interface.

The application and the plugins are loosely coupled and interact through interfaces. The application discovers and loads the plugin through some standard mechanism that (often) can be reused by other applications. We usually prefer to have a well-known directory that contains plugins. Developers and/or administrators may add or remove plugins from this directory. The benefit of such a programming model is that it is very easy to evolve an application and develop major parts of its functionality in isolated plugins that are easy to test and deploy and don't require modifying the application code itself. It also makes it very easy to create custom apps with different functionality

(just pick a custom set of plugins). Finally, application startup time can be dramatically reduced if plugins are loaded only when they are needed.

Graphr is a plugin-based application. It has a *MainWindow* class that's responsible for interacting with the user, the embedded IronPython evaluator, and configuring the active graph properties. It has a *GraphManager* class that's responsible for loading the graph plugins and managing the active graph. And it has graph plugins that are responsible for displaying the current graph data.

All the components interact through interfaces and promote decoupling. The interfaces are defined in their own assembly (Graphr.Contract) and are referenced by Graphr itself and by each plugin. Graphr and the plugins don't reference each other. Their main interface is IGraph. There is an additional interface, IGraphMetadata, that is used by MEF.

The IGraph interface

The IGraph interface has three methods: *PopulateCanvas()*, *DoGraphLayout()*, and *ConfigSpec*.

The *PopulateCanvas()* method is called whenever the data is modified. It accepts a *Canvas* object (where the graph is displayed), the data as a list of *x,y* values, and a *config* object, which is a dictionary of graph specific items like color and stroke thickness. The graph plugin that implements the interface should construct the visual elements that represent the current data and the graph properties. For example, a bar graph will create a bar object for each *x,y* pair,and set its border color and fill color based on the graph properties.

The *DoGraphLayout()* method is called every time the graph needs to be redrawn. This happens whenever the canvas is resized (when the whole window is resized or the splitter between the left and right pane is moved). It accepts the canvas and the data. You could argue that the plugin could store the canvas and the data when *PopulateCanvas()* is called — then the *DoGraphLayout()* method could be simpler with no arguments. This is true, but since the *MainWindow* that calls *DoGraphLayout()* must have the canvas and data anyway, it is better not to store the same information in each plugin, too.

The *ConfigSpec* property is a dictionary that maps names to type and object. Each graph plugin supports a different set of named configuration items of different types. A dictionary of typed objects is a generic way to represent this set. The *ConfigSpec* contains the initial values of these items (e.g., *StrokeThickness = 2*). The graph properties pane displays these initial values and the user may modify them and change the appearance of the displayed graph.

Here is the entire IGraph interface:

```
public interface IGraph
{
  void PopulateCanvas(
    Canvas c,
    IList<KeyValuePair<double, double>> data,
    IDictionary<string, object> config);

  void DoGraphLayout(
    Canvas c,
    IList<KeyValuePair<double, double>> data);

  // The config spec is a triplet of config items
  // name: string
  // type: Type
  // initial value: object of the item's type
  IDictionary<string, Tuple<Type, object>> ConfigSpec { get; }
}
```

The IGraphMetadata Interface

This is a very simple interface that just has a single property called *Name*:

```
public interface IGraphMetadata
{
  string Name { get; }
}
```

This is an interface used by MEF to annotate plugins with metadata that can be queried without instantiating the plugin. You will see later how this metadata is used by Graphr.

GraphManager and MEF

GraphManager is a class that encapsulates the plugins and the interaction with MEF and also keeps track of the current active graph plugin. It begins by loading the graph plugins, which are implemented as MEF parts. The *GraphManager* has the following data member:

```
[ImportMany]
public IEnumerable<Lazy<IGraph, IGraphMetadata>> Helpers { get; set; }
```

The *Helpers* collection is decorated with MEF's [*ImportMany*] attribute. It is an enumerable collection of *Lazy* pairs of *IGraph* and *IGraphMetadata*. The *Lazy<T, Metadata>* is an MEF extension of the .NET 4 *Lazy<T>* template. It allows accessing the metadata without instantiating the plugin itself.

In order for a graph plugin to be discoverable this way, it must comply with the following requirements:

1. Implement IGraph
2. Have an [*Export]* attribute with a type of *IGraph*
3. Have an [*ExportMetadata*] attribute with a "*Name*" property that matches the *IGraphMetadata* interface.

The following code is a snippet from the LineGraph plugin:

```
[ExportMetadata("Name", "Line")]
[Export(typeof(IGraph))]
class LineGraph : IGraph
{
  ...
```

MEF can discover and load all compliant plugins into the *Helpers* collection. To locate the plugins, we use MEF's *DirectoryCatalog*. Because we want to scan both the current directory and a "plugins" directory, we use an *AggregateCatalog* and add both the current directory and the "plugins" directory (if we can find it). Once the catalog is ready, we create a *CompositionContainer* with the catalog and call its *compose-Parts*() method. That tells MEF to do its magic: Scan the directories in the catalog, scan each assembly in these directories for types that comply with the graph plugin requirements, and load them (without instantiation) into the *Helpers* collection.

```
    public GraphManager(Canvas c, Grid g)
    {
      this.canvas = c;
      this.propertyGrid = g;
// Discover and load graph plugins via MEF magic. All the plugins
// will automagically populate the Helpers collection

      var catalog = new AggregateCatalog();
      catalog.Catalogs.Add(new DirectoryCatalog
         (Directory.GetCurrentDirectory()));

      // Find the plugins directory
      var cd = Directory.GetCurrentDirectory();
      var d = cd;
      var pluginsDir = Path.Combine(d, "plugins");
      var root = Directory.GetDirectoryRoot(d);
      while (d != root)
      {
        d = Path.GetDirectoryName(d);
        pluginsDir = Path.Combine(d, "plugins");
        if (Directory.Exists(pluginsDir) && pluginsDir != cd)
        {
          catalog.Catalogs.Add(new DirectoryCatalog(pluginsDir));
          break;
        }
      }
      var container = new CompositionContainer(catalog);
      container.ComposeParts(this);
    }
```

When the user selects a new graph type in the graph selector drop-down box (or when initially loading the default graph type), the *GraphManager* is responsible for the switch. The *SwitchGraphHelper()* method is called, which selects from the *Helpers* collection the graph plugin whose *Metadata.Name* property (defined in the [*ExportMetadata*] attribute) matches the name argument. Then it populates the graph properties pane using the current helper's *ConfigSpec* and repopulates the canvas if there is data, as shown in this code:

```
    public void SwitchGraphHelper(string name)
    {
      this.helper =
          Helpers.Single(h => h.Metadata.Name == name).Value;

      PopulateGraphProperties(helper.ConfigSpec);
      if (this.data != null)
      {
        PopulateCanvas(this.data);
      }
    }
```

The *PopulateGraphProperties()* method populates the graph properties grid in the left pane dynamically based on *ConfigSpec*:

```
public void PopulateGraphProperties(IDictionary<string,
            Tuple<Type, object>> configSpec)
    {
      config = new Dictionary<string, object>();
      foreach (var kv in configSpec)
      {
        config.Add(kv.Key, kv.Value.Item2);
      }
```

```
    var g = this.propertyGrid;

    g.ColumnDefinitions.Clear();
    g.RowDefinitions.Clear();
    g.Children.Clear();

    // Define the Columns
    ColumnDefinition colDef1 = new ColumnDefinition();
    ColumnDefinition colDef2 = new ColumnDefinition();
    g.ColumnDefinitions.Add(colDef1);
    g.ColumnDefinitions.Add(colDef2);

    int row = 0;
    foreach (var pair in configSpec)
    {
      var s = pair.Value;
      RowDefinition rd = new RowDefinition();
      rd.Height = new GridLength(25);
      g.RowDefinitions.Add(rd);

      var label = new Label();
      label.Content = pair.Key;
      g.Children.Add(label);
      var editor = _createEditor(pair.Key, s.Item1, s.Item2);
      g.Children.Add(editor);

      Grid.SetRow(label, row);
      Grid.SetColumn(label, 0);

      Grid.SetRow(editor, row);
      Grid.SetColumn(editor, 1);

      ++row;
    }
  }
```

The *_createEditor()* method creates an editor for the particular item type. The supported types are: *string, Color, Int32,* and *Double*. In the current implementation, it is always a *TextBox*, but in general it can be any *UIEllement*. Different item types are handled differently. The *_createEditor()* method attaches a type-specific handler for each item type:

```
  private UIElement _createEditor(string name, Type t, object v)
  {
    var e = new TextBox() { Name = name, Text = v.ToString() };
    if (t.Name == "string")
     e.TextChanged +=
                new TextChangedEventHandler(_onTextChanged);
    else if (t.Name == "Color")
     e.TextChanged +=
                new TextChangedEventHandler(_onColorChanged);
    else if (t.Name == "Int32")
     e.TextChanged += new TextChangedEventHandler(_onIntChanged);
    else if (t.Name == "Double")
   e.TextChanged +=
                new TextChangedEventHandler(_onDoubleChanged);
    else
    {
      throw new Exception("Unknown type");
    }

    return e;
  }
```

Details about Graphr code (unrelated to MEF or IronPython, but including descriptive details about the plugins) are available at http://www.drdobbs.com/windows/229402979. Complete code and build files for the project are available at http://is.gd/blplMd.

Conclusion

Graphr is a fun project that demonstrates the power and ease of use of WPF and explores the polyglot programming world by embedding IronPython in C#. It also shows how simple it is to add plugins to an application with MEF. If you like it you may find it interesting to further extend it.

Related Links

Using IronRuby in .NET Programs: http://drdobbs.com/224600662

— Gigi and Saar Sayfan are regular contributors to Dr. Dobb's, most recently authoring a two-part exploration of the PolyArea Project (http://drdobbs.com/windows/226700093 and http://drdobbs.com/open-source/227700264).

Clever Hax: Broadcasting Build Notifications from TFS

Team Foundation Server makes it easy to notify your team about build failures in clever ways: flashing red lights or an SMS message to every team member.

By Marcin Kawalerowicz and Craig Berntson

A build notification tells you the results of a build. Did it succeed? Did it fail? And if it failed, why? And which module caused the failure? Continuous integration (CI) servers have a rich notification repertoire. Windows tray icons, e-mails, instant messaging, and IDE plug-ins are just some of the possibilities. This variety of options is a proof for the thesis that build notification is one of the most important parts of CI.

In fact, it's so important that some CI practitioners sacrifice an additional monitor or even a whole machine to provide constant monitoring for the team. It's usually an old computer with an old monitor that stands in a visible place in the developers' room or some place where everybody can see it. The sole purpose of this machine is to provide the team with up-to-date information about the build processes. The dashboard page may refresh every few minutes, or special custom software may monitor the CI server. If your team can benefit from something like this, set it up. Or get geeky and use the following LED message board to provide a broken build notification.

Providing Feedback via an LED Message Board

An LED message board is a gadget that comes from a big family of crazy USB toys from the China Seas area. The one this example uses is a matrix of 7×21 LED lights sealed in a small black plastic casing (sometimes called a human interface device, or HID). If you aren't a USB geek, the only thing you have to know is that the HID driver makes it easy to interact programmatically with an HID-enabled device. If you want to buy one, search for "USB LED message board" or "scrolling USB LED message board."

Let's write a simple program that checks the state of the builds on a TFS server. If it finds a broken build, it'll display a blinking red circle on an LED message board. This display should be hard to miss if the board is in the developers' room.

First, let's find out whether the last build in a given build definition was broken. The following listing shows the details.

Listing One: Sniffing around the last broken build in TFS 2010round the last broken build in TFS 2010

```
NetworkCredential Credentials =
    new NetworkCredential("marcin", "password");
TeamFoundationServer tfs =
    new TeamFoundationServer(
        "http://tfs1:8080/tfs/CiDotNet", Credentials);
IBuildServer buildServer =
    (IBuildServer)tfs.GetService(typeof(IBuildServer));
IBuildDetailSpec buildDetailSpec =
     buildServer.CreateBuildDetailSpec(
         "CiDotNet.Ch4.Tfs", "Ci.CiDotNet.Ch4.Tfs");
buildDetailSpec.MaxBuildsPerDefinition = 1;
buildDetailSpec.QueryOrder =
    BuildQueryOrder.FinishTimeDescending;
IBuildQueryResult results =
    buildServer.QueryBuilds(buildDetailSpec);
if (results.Failures.Length == 0
   && results.Builds.Length == 1)
{
    IBuildDetail buildDetail = results.Builds[0];
    if (buildDetail.Status == BuildStatus.Failed)
    {
        MyLedNotify();
    }
}
```

This example uses the Microsoft.TeamFoundation API to sniff for the latest build output. You use the *Microsoft.TeamFoundation*, *Microsoft.TeamFoundation.Client*, and *Microsoft.TeamFoundation.Build.Client* namespaces to first connect to the TFS server and team collection using network credentials (lines 1-7). Then you get the build server from the TFS instance and query it for a given team project and build definition (lines 8-10). You take only the last build result (lines 11-13) and check its state. If it fails, you turn on the big red dot (14-23); see Listing Two.

Listing Two: Using an LED message board to notify you about a broken build.

```
HidDevice MessageBoard;
HidDeviceList = HidDevices.Enumerate(0x1d34, 0x0013);
if (HidDeviceList.Length > 0)
{
    MessageBoard = HidDeviceList[0];
    MessageBoard.Open();
    Thread.Sleep(1000);
    byte[] Packet0 = new byte[] { 0x00, 0x00, 0x00,
    0xff, 0xfe, 0xff, 0xff, 0xfd, 0x7f };
    byte[] Packet1 = new byte[] { 0x00, 0x00, 0x02,
    0xff, 0xfb, 0xbf, 0xff, 0xf7, 0xdf };
    byte[] Packet2 = new byte[] { 0x00, 0x00, 0x04,
    0xff, 0xfb, 0xbf, 0xff, 0xfd, 0x7f };
    byte[] Packet3 = new byte[] { 0x00, 0x00, 0x06,
    0xff, 0xfe, 0xff, 0x00, 0x00, 0x00 };
    for (int i = 0; i < 10; i++)
    {
        MessageBoard.Write(Packet0);
        MessageBoard.Write(Packet1);
        MessageBoard.Write(Packet2);
        MessageBoard.Write(Packet3);
        Thread.Sleep(50);
    }
    MessageBoard.Close();
}
```

This example uses a generic .NET HID device library from Mike O'Brien (https://github.com/mikeobrien/HidLibrary). The library provides a way to connect and use any HID-enabled device. All you have to do is to get a manual for the interface used in this device. The manufacturer of our LED message board was kind enough to provide one when asked.

To connect an HID device, you have to find it using a unique identifier provided by the manufacturer (lines 1-2). After you find the device, you must connect to it (line 6) and wait a while for the hardware to snap in. Next, you define the big red dot using a report formatted according to the manufacturer's interface description (7-16). For now, you'll have to believe us that the lights form a big red dot on the LED message board. You then send the light definition to the device (17-23), after which the device will look like the one shown in Figure 1. The packets sent to the LED message board light up the device for only a few milliseconds, so you have to refresh the signal to light it up periodically.

It doesn't cost much to provide a new way to notify your team about a problem. Buying a flashing emergency vehicle roof light and installing it in the developers' room is an even better idea (of course, including the siren!). But because the Taiwanese LED device is a lot cheaper (around $10-$20), you can start with it. It's that important to react immediately to a broken build.

What if the blinking lights, emails, and sirens aren't doing the job? How about something more intrusive? Send an SMS message to every team member.

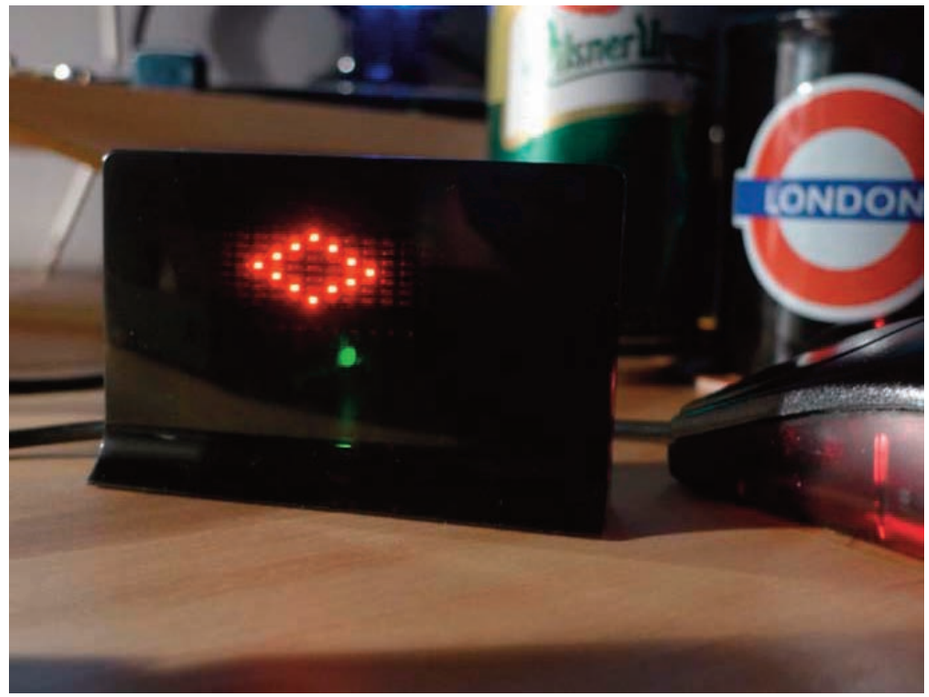

Figure 1. LED message board blinking with a red eye to indicate a failed build

Providing Feedback via SMS Notifications

It's a little scary idea to send someone an SMS message with a build notification. But what if you're on vacation climbing Mount Kilimanjaro and you want to know if your team is dealing with the broken build fast enough? No problem. You can take the easy route and send yourself an SMS message using Skype and an online computer in your office. Here's how.

Skype provides a COM library to automate some of its tasks. One of the methods provided by this API is SendSms, which we'll use here. This method requires you to have Skype installed and some money in your Skype account because, unfortunately, SMS isn't free.

You can download the Skype library from https://developer.skype.com/. Check the Tools and SDK area. To do the build-state sniffing, you can use a variation of the program shown earlier in Listing One. The hitch is to detect only the change in the state of the build from successful to

broken, and then send one SMS message. After such an event, it's a matter of implementing the following code to send the SMS message:

```
Skype Skype = new Skype();
if (Skype.Client.IsRunning)
{
    Skype.Client.Start();
}
Skype.SendSms(PhoneNumber, Message);
```

From now on, you can sleep well, knowing that an SMS message will alert you if something goes wrong with your build. Isn't that comforting?

Summary

Access to immediate and accurate information about the state of your build process is vital to your CI quality. The faster you get the information about a change in the quality of your source code, the more quickly you can react to fix the problem. The faster you fix the problem, the better your team will work. You'll know where you journey is taking you and whether your project is starting to veer off the road. In this article, we discussed how to extend your build notifications.

— Marcin Kawalerowicz works for European customers in the automotive and financial sectors, among others. He blogs at iprogrammable.com. Craig Berntson is a Microsoft MVP. He blogs at www.craigberntson.com/blog.

This article was adapted with permission from the authors' recently released book, *Continuous Integration in .NET* (http://is.gd/dRiGLt), published by Manning Publications.

From the Vault

Creating Better Cryptographic Hash Functions

As encryption technologies are broken by hackers and researchers, the need to for better cryptographic hashes attains new urgency. Here is how these hashes are created, with a look at the algorithm innovations that have emerged to counter security threats.

— *DDJ*

By Jesse Walker, Michael E. Kounavis, Shay Gueron, and Gary Graunke

Hash functions are one of cryptography's most fundamental building blocks, even more so than encryption functions. For example, hash functions are used for digital fingerprinting and commitment schemes, such as message authentication and random number generation, as well as for digital signature schemes, stream ciphers, and random oracles.

Recently, Antoine Joux, one of the leading cryptographers of our time, discovered multi-collision attacks against the general framework in which hash functions are constructed [1], and Xiaoyun Wang created an attack that breaks the collision-resistance property of the most widely deployed hash functions [2], including MD5 and SHA-1. In 2005, Arjen Lenstra and Wang demonstrated how to forge two digital certificates, based on the MD5 hash function, by using different keys but the same signature — something that was hitherto thought to be impossible [3]. These attacks that were discovered, and other vulnerabilities that were exposed by researchers have stimulated a resurgence of research into hash functions. This research has also spawned an international competition, sponsored by the National Institute of Standards and Technology (NIST), an agency responsible for standards used by the U.S. Government, to create a next-generation hash function design.

In this article we highlight some recent work in hash function development. We begin by providing some background on hash functions and look at the problems that hash functions address. We then sketch how to build a hash function. Moving on, we outline recent seminal work in the field of hash functions. We describe two radically different designs entered in the NIST competition, the Skein and Vortex designs, created in part with Intel participation. This is followed by a discussion of the design rationale for each where we also compare and contrast the basic design decisions. We end with a summary of our findings.

Hash Functions

A hash functions is usually defined as a function *H* satisfying three properties [4]:

- Collision resistance. It is computationally infeasible to find two distinct bit strings $s \neq s'$ such that $H(s) = H(s')$.
- Pre-image resistance. Given a hash value *t* in the range of *H*, it is computationally infeasible to find a string *s* for which $H(s) = t$.
- 2nd Pre-image resistance. Given a string *s* and hash value *t* such that $H(s) = t$, it is computationally infeasible to find a second string *s'* such that $H(s') = t$ as well.

A hash function maps the set of all bit strings into a "message digest" of defined length, called the hash function's "block size." Since there are many more strings than message digests, at least one digest output by the hash function must be the image of more than one input string. It is therefore remarkable that it is possible to build a function *h* that has the three properties previously noted. These properties imply that h essentially acts like a randomly selected compression function.

It is instructive to describe some typical use cases:

- Digital fingerprinting. Hash functions construct effective digital fingerprints. If *d* represents a digital data structure, such as a document, then the message digest $h(d)$ is its fingerprint; the 2nd-pre-image-resistance property says it is infeasible to find a second data structure *d'* with the same fingerprint $h(d') = h(d)$.
- Digital signatures. Digital signatures extend digital fingerprinting by encrypting the hash value $h(d)$ of a data structure *d* under a private key.
- Message authentication. A hash function h used with a secret key *K* can be used to authenticate a message *m*. The idea is to create a tag $t = h(K \| m)$, where "||" denotes string concatenation, which is sent with the message and verified by the receiver. (This does not quite work in practice, because the cascade construction introduces vulnerabilities hashing the last message block. Instead the tag is essentially computed as $h(K \| h(K \| m)))$. Because of pre-image resistance, it is infeasible for an attacker to create the same tag tunless he or she knows the key *K*, and because of collision resistance, the tag could have been created only by concatenating the message m to the key *K*.
- Pseudo-random number generation. One standard way to build a random number generator is to take a key *K*, usually called a "seed," and to compute $h(K \| 0)$, $h(K \| 1)$, $h(K \| 2)$, with each digest representing a different random number. If *h* and *K* are carefully selected, it is infeasible to distinguish this, by any statistical test, from a stream of genuine random numbers.
- Stream ciphers. A stream cipher can be built by taking a hash function *h*, and encrypting message m_i by $m_i \oplus m_i \oplus h(K, i)$, where "$\oplus$" denotes XOR, and decryption is $mi \oplus h(K, i) \oplus (mi \oplus h(K, i)) \oplus h(K, i) = m_i$.
- Random oracles. Random oracles can be thought of as specialized random number generators. They are used widely in cryptography, such as for randomizing public and private key encryption, to make them secure from arcane attacks.

Designing Hash Functions

The standard approach to building a hash function is to first construct a compression function that operates on the input strings of a fixed

length, and then to use the cascade construction to extend the compression function to strings of arbitrary length [5, 6].

Compression functions are usually built out of block ciphers. Recall that a block cipher is a pair of *D* and *E* functions, for decrypting and encrypting, respectively, that operate on strings of a particular length, called the "block size." If the block size is *n*-bits, then the encryption of an *n*-bit string *s* is *E (s),* and its decryption is *D (s)*. Every string can be encrypted or decrypted, and $E(D(s)) = D(E(s)) = s$, meaning *E* (and *D*), is a permutation of the set of all *n*-bit strings. Cryptographers say a block cipher is secure if both $s \rightarrow E(s)$ and $s \rightarrow D(s)$ are indistinguishable from a randomly selected permutation. To meet the indistinguishability property, block ciphers are keyed, so a block cipher represents a family of permutations. *A* particular block cipher instance is selected by choosing a key *K*. The resulting encryption and decryption instances are denoted E_K and D_K. That is, for each choice of a key *K*, $s \rightarrow E_K(s)$ (and $s \rightarrow D_K(s)$) behave like different randomly selected permutations.

Compression Functions

The compression functions for all the hash functions commonly used today are built in the following way:

1. Select a block cipher scheme *(E, D)*.
2. Define a compression function $c(iv, s) = E_s(iv) \oplus iv$.

Here *s* denotes a message of exactly *n*-bits, and *iv* denotes an initialization vector. This recipe for *c* says to use s as the encryption key and iv as the data to be encrypted, and then to use XOR *s* with the encrypted result $Es(iv)$. The mapping $(iv, s) \rightarrow Es(iv) \oplus iv$ is called the Davies-Meyer construction [7] for *E*. It is easy to show that a block cipher used in Davies-Meyer mode is collision-resistant, pre-image resistant, and 2nd pre-image resistant. *c* is called a compression function because it compresses <*iv,s*> into a new string *iv'* of exactly *s*'s length. Other compression function constructions also exist: both Vortex and Skein use the Matyas-Meyer-Oseas [8] construction, $c(iv, s) \rightarrow Eiv(s) \oplus s$, which is identical to Davies-Meyer, except it reverses the role of *iv* and *s*.

The Cascade Construction

The cascade construction builds a hash function *h* from a compression function *c* with block size *n* as in this equation:

cascade (*s*)

pad (*s*); $s_1\ s_2 \ldots s_b \leftarrow s$; $iv_1 \leftarrow iv$; **do** $i = 1$ **to** $b \Rightarrow iv_{i+1} \leftarrow c(iv_i, s_i)$ **od**;

output iv_{b+1}

Every hash function based on a block cipher must define a padding scheme, because compression functions only operate on strings *s* of length *n* bits exactly. Most padding schemes pad *s* with a single 1 bit followed by as many 0 bits as are necessary to bring the length to a multiple of *n*. The length of the unpadded string *s* is then encoded as an *n*-bit integer and appended to defend against extension attacks.

Once padded, partition *s* into $b = |s|/n$ blocks, each consisting of *n* bits ($|s|$ denotes *s*'s length in bits): $s_1\ s_2\ s_b \leftarrow s$.

Finally, beginning with a hash-function-specific initialization vector *iv*, serially compute $c(iv_i's_i)$ for each block s_i.

The cascade construction extends the collision resistance, pre-image resistance, and pre-image resistance properties from a compression function to a function operating on strings of arbitrary length [9]. The

cascade construction is sometimes called the Merkle-Damgard construction after its inventors. This construction is intrinsically serial, as it needs to be able to detect problems, such as two blocks being exchanged.

Hashing Today

We just summarized the state of the art during the early part of this decade, prior to two significant publications. The first was by Antoine Joux, who introduced the multi-collision attack. The second was by Xiaoyun Wang, where she described an attack, based on differential cryptanalysis, against all of the hash algorithms broadly used today.

Suppose a hash function is built out of a compression function by using the cascade construction. Also suppose that someone has broken the collision resistance of the hash function; that is, they have discovered two distinct strings $s \neq s'$ so that $h(s) = h(s')$. Joux observed that it is easy to find many more collisions for little additional cost [1]. The source of the problem is that the cascade construction maintains too little state as it progresses from one invocation of the compression function to the next. Joux's result says that by itself the cascade construction is too weak to serve as an adequate building block for constructing hash functions.

Wang's attack [2], based on differential cryptanalysis, has a different flavor. Differential cryptanalysis is a technique to analyze block ciphers. Essentially, differential cryptanalysis follows a bit slice through the block cipher being analyzed, to characterize how it gets diffused. The goal of differential cryptanalysis is to identify bits leading to unusually high or low levels of diffusion. When such bits are identified, they can be used to recover bits of the encryption key. This can dramatically shrink the size of the key space, making brute force search realistic. As an example, differential cryptanalysis reduced the cost of key recovery attacks against the DES cipher from 256 encryptions to about 241.

Wang showed that a differential attack could produce collision in message digests, thereby breaking the collision-resistance of the hash function producing them. Wang first demonstrated her attack against MD4, MD5, RIPE-MD, and SHA-0. This was viewed as a stunning result, but then she demonstrated that a collision can be produced in SHA-1 at a cost of about 261 operations. This caused upheaval in the cryptographic community, raising the question as to whether we even understand what a hash function is.

The cryptographic community has vigorously debated hash design principles in the intervening years. The only clear consensus emerging from this debate is that we need a worldwide, focused project whose goal is to create a new generation of hash functions that defend against the new attacks. A lesson previously learned by the community is that contests have great efficacy in galvanizing technical consensus building. In 2007, NIST initiated an international competition to create a new hash standard [10]. Candidate submissions were due on October 31, 2008. 55 algorithms were entered. In February of 2009, NIST whittled down the list of candidates to 40, and from this it selected the first-round candidates in August 2009. Then NIST selected five SHA-3 finalists to advance to the third (and final) round of the competition on December 9, 2010, which ended the second round of the competition.

A one-year public comment period is planned for the finalists. NIST also plans to host a final SHA-3 Candidate Conference in the spring of 2012 to discuss the public feedback on these candidates, and select the SHA-3 winner later in 2012 (http://is.gd/9YJOxa).

NIST is widely influential in the creation of cryptographic standards worldwide, so it is a good sponsor for the competition. One of NIST's

most important contributions to cryptography standards has been the creation of requirements for algorithms submitted to the competition. The competition requires that candidate algorithms provide the collision-resistance, pre-image-resistance, and 2nd-pre-image-resistance properties — and be free of any known intellectual property. Algorithms must support output block sizes of 128, 160, 224, 256, 384, and 512 bits. The rules encourage support for features outside the core properties, especially for parallelization. Submissions must be accompanied by a security rationale, to help establish confidence in the algorithms.

Some New Designs

Two of the candidates submitted to the NIST hash competition, Skein and Vortex, include contributions by Intel personnel. Skein is among the entries remaining in the competition.

Skein

Skein was designed by Mihir Bellare (University of California, San Diego), Jon Callas (PGP Software), Niels Ferguson (Microsoft), Tadayoshi Kohno (University of Washington), Stefan Lucks (Bauhaus University-Mannheim), Bruce Schneier (British Telecom), Doug Whiting (Hi-Fn), and Jesse Walker (Intel Corporation). Skein produces message digests of any length from 1 to 296 bytes. Skein has three major components: a new block cipher named Threefish, a replacement for the cascade construction named Unique Block Iteration (UBI), and an argument system extending Skein's domain of use beyond hashing.

The First Skein Component: The Threefish Block Cipher

Threefish is a "tweakable" block cipher [11], which means that a randomizer called a tweak is passed to the cipher with the key and data to encrypt. Skein uses the block offset from the start of the message as the Threefish tweak. The tweak addresses many deficiencies in the cascade construction and represents the major innovation in Skein.

Threefish has three flavors: a 256-bit, a 512-bit, and a 1024-bit block size. The Threefish encryption key is the same size as the block size. The tweak is always 128 bits.

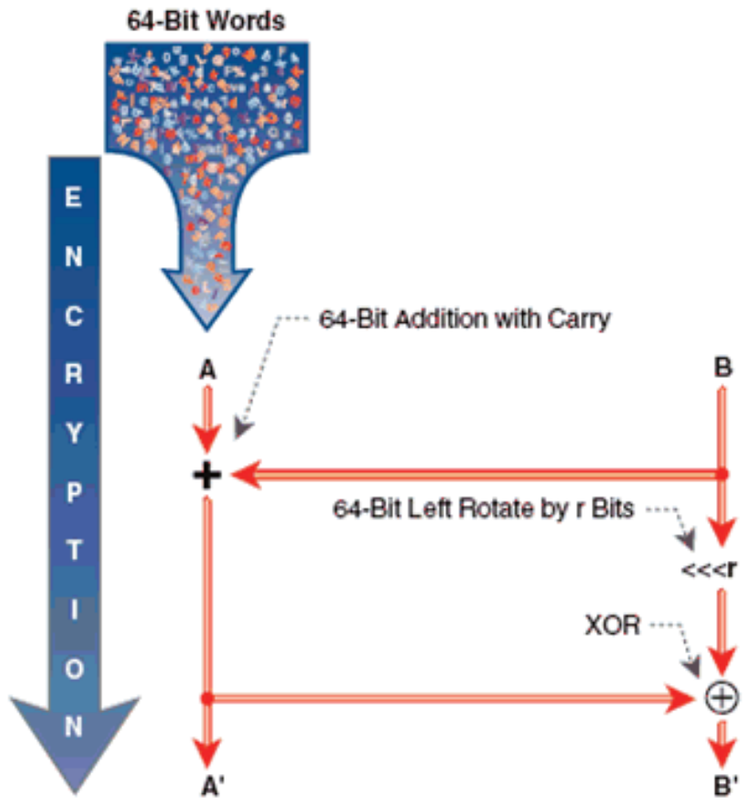

Figure 1: Threefish MIX (Source: Intel Corporation, 2009).

Threefish is a product cipher, meaning it is composed of rounds. Each round is a simple but weak encryption function. Threefish obtains security by piling round upon round: 72 rounds for Threefish-256 and for Threefish-512 and 80 rounds for Threefish-1024. The number of rounds represents a tradeoff between performance and security.

A Threefish round consists of a number of parallel MIX functions followed by a permutation, so that different blocks are mixed for different rounds. The MIX functions are made up of just three instructions — 64 bit addition, left rotate, and XOR — to combine two 64-bit words *A* and *B*, as depicted in Figure 1.

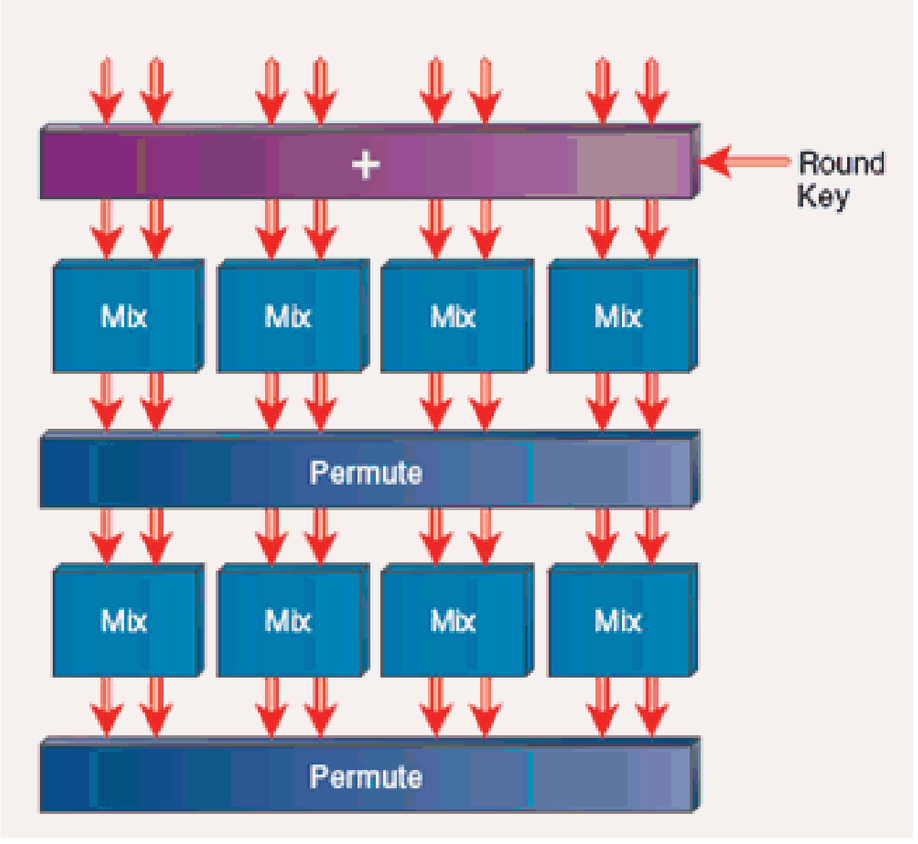

Figure 2: Two Rounds of Threefish-512 (Source: Intel Corporation, 2009)

Threefish-256 splits its input into four 64-bit words, so each round consists of two parallel MIXes: Threefish-512 uses eight words with four parallel MIXes, and Threefish-1024 uses sixteen words with eight parallel MIXes. Figure 2 depicts the Threefish round structure for Threefish-512. The parallel MIXes efficiently exploit the super-scalar properties of modern processors. The rotation constants *r* were selected by a hill-climbing algorithm that maximized diffusion over randomly selected sets of rotation constants.

Threefish adds a round key every four rounds. Figure 2 depicts one of these additions. The Threefish round keys come from a key schedule inspired by Skipjack's key schedule [12]. Each Threefish round key is the same size as the plaintext data block, and each key depends on all the bits of both the encryption key and the tweak.

The Second Skein Component: Unique Block Iteration

Unique Block Iteration (UBI) mode replaces the cascade construction in Skein. UBI consists of four parts. First, UBI uses the Matyas-Meyer-Oseas construction, $(iv, s) \rightarrow Eiv(s) \oplus s$, to build a compression function *c* out of any block cipher. Second, UBI padding appends enough 0 bits to bring the length of the message being hashed to a multiple of the block size. Third, UBI constructs and passes the tweak to the block cipher. The UBI tweak is composed of two flags and of the message block offset from the beginning of the message in bytes. One of the flags is set on the first block, and the second flag is set for the final block. Finally, UBI computes its output just like the cascade construction, the only difference being the construction of the tweak, as shown in the following equation:

UBI(*iv*, *s*)

pad(s)

$t \leftarrow 0 \oplus$ start-flag; $iv_2 \leftarrow c(iv, s_1, t)$; t = block-size;

do $i = 2$ ***to*** $b-1 \Rightarrow iv_{i+1} \leftarrow c(iv_i s_i, t)$; t t+block-size ***od***;

$t \leftarrow t \oplus$ end-flag; ***output*** $c(iv_b, s_b, t)$

UBI uses the Matyas-Meyer-Oseas construction instead of Davies-Meyer. This converts attacks against a hash function from related key attacks to chosen plaintext attacks against the block cipher: The community understands more about defending against the latter than it does about defending against the former.

The Third Skein Component: Skein Argument System

The Skein argument system extends the algorithm beyond normal hashing to application-specific or personalized hashing, message authentication, key derivation, pseudo-random number generation, stream ciphers, and tree (that is, parallelized) hashing.

Putting it Together in Skein

Skein instantiates UBI mode with the Threefish block cipher. The design's initialization vector is computed as the UBI-Threefish output of the configuration string "SHA-3." Skein first hashes a string *s* under UBI mode and *iv* to obtain an intermediate value. Skein uses the intermediate value as an *iv* to hash the integers 0, 1, 2, under UBI, again to obtain the final output. Classical theory justifies the claim that Skein-*n* (*n* = 256, 512, or 1024) achieves *n/2* bits of security against collisions, and n-1 bits of security against 1st and 2nd pre-image attacks — the best that can be achieved, theoretically. The double hashing under UBI mode also allows Skein to make additional, unusually strong claims, as follows:

- If Threefish is a pseudo-random permutation, then Skein can be used as a pseudo-random function, a secure key derivation function, a secure message authentication code, a secure stream cipher, and a secure pseudo-random number generator.
- If Threefish acts like an ideal cipher, then Skein cannot be differentiated from a random oracle.

The first claim says that Skein can be used naively in a broad range of applications that usually require great sophistication when constructed from classical hash functions. The second claim is a non-trivial result: it claims that Skein is structurally sound when Threefish is viewed as a black box; that is, the attacker is not allowed to utilize any knowledge about the internals of Threefish. This structural property means the security of Skein depends on the security of the underlying block cipher only. The best known attack against Threefish at this time breaks a 34-round variant (out of 72 rounds for full Threefish), which is superior to AES, whose 8-out of 10-round variant falls to attack.

In software Skein is one of the fastest unbroken algorithms ever devised: it runs at 6.1 clocks/byte on an Intel Core Duo processor and requires no special hardware acceleration, such as an AES round instruction. This is twice as fast as the best software implementations of the current hashing standard. Skein also maintains a very small footprint for its in-memory state, allowing implementation in even constrained environments such as smart cards.

Vortex

Vortex is a family of hash functions developed by Michael Kounavis and Shay Gueron of Intel. A main strength of the Vortex design is that this hash function can achieve an ideal performance of 2.2-2.5 cycles per byte by using the AES round [14] and carry-less multiply instructions [15]. Such instructions have been announced for future Intel processors. Vortex is one of the fastest collision-resistant hashes known when running on future IA processors, outperforming SHA-1 (approx. 7 cycles/byte) by 3.18X, and outperforming SHA256 (approx. 19 cycles/byte) by 8.63X.

The Vortex family produces message digests of 224, 256, 384, and 512 bits, respectively. The main idea behind Vortex is to use well-known algorithms with very fast diffusion in a small number of steps. These algorithms also balance the cryptographic strength, that comes from iterating block cipher rounds with S-box substitution and diffusion, against the need to have a lightweight implementation with as small a number of rounds as possible. Vortex is built upon the following algorithms:

- The Rijndael round function, which performs very fast mixing across 32 bits, as a standalone operation, and 128 bits or 256 bits, if combined with at least one more round.
- A variant of Galois Field multiplication that mixes bits of different sets in a manner that is cryptographically stronger than many other simpler schemes.

Vortex uses a variable number of Rijndael rounds with a stronger key schedule. The number of rounds is a tunable parameter. Rijndael rounds are followed by a variant of Galois Field multiplication to cross-mix between 128-bit or 256-bit sets. This transformation is not simple carry-less multiplication; rather, it combines bit reordering operations, XORs, and additions with carries. In this way, this variant of Galois Field multiplication achieves better diffusion than the straightforward carry-less multiplication between the 128-bit or 256-bit inputs; it is also a non-commutative operation, protecting against chaining variable swapping attacks.

Vortex uses the Enveloped Merkle-Damgrd (EMD) construction to lift collision resistance, pre-image and 2nd pre-image resistance, pseudo-random oracle preservation, and pseudo-random function preservation from the underlying compression function to the hash function. To achieve its properties, the EMD construction first hashes the input string s under one initialization vector to get an intermediate value, and then it hashes the intermediate value under a second initialization vector to obtain a final result.

For Vortex-256, Gueron and Kounavis demonstrate that the number of queries required to find a collision with a probability greater or equal to 0.5 is at least 1.18 * 2122.55 [13].

In summary, the Vortex compression function uses the Rijndael round function. Vortex-224 and Vortex-256 use Rijndael-128 rounds. Vortex-384 and Vortex-512 use Rijndael-256 rounds. AES uses the Rijndael-128 round function. For the remainder of this section, $K(X)$ denotes a block cipher based on the Rijndael round function that encrypts X by using key K. $V_M^{(A)}(A, B)$ is a multiplication-based merging function.

The Vortex-block algorithm is the Vortex compression function. This algorithm incorporates two repetitions of an algorithm called Vortex-sub-block. The first repetition of Vortex-sub-block accepts as input the chaining variable $A_i \,||\, B_i$ and two least-significant input block words

W_{4i}, W_{4i+1} of the message being hashed. It returns an intermediate value for the chaining variable $A \,||\, B$. The second repetition of Vortex-sub-block accepts as input the intermediate value of the chaining variable $A \,||\, B$ and two most-significant input block words W_{4i+2}, W_{4i+3}. It returns an update on the chaining variable $A_{i+1} \,||\, B_{i+1}$.

With the exception of the last sub-block (discussed later), the algorithm for processing a Vortex-sub-block is as follows:

Vortex sub-block (A, B, W0, W1)

// W_0 is the first word of the current sub-block to be processed

$A \leftarrow \tilde{A}_A(W_0) \oplus W_0 ; B \leftarrow \tilde{A}_B(W_0) \oplus W_0 ; A \,//\, B \leftarrow V_M^{(A)}(A, B)$

// W_1 is the second word of the current sub-block to be processed

$A \leftarrow \tilde{A}_A(W_1) \oplus W_1 ; B \leftarrow \tilde{A}_B(W_1) \oplus W_1 ; A \,//\, B \leftarrow V_M^{(A)}(A, B)$

output $A \,//\, B$

The structure of the Vortex sub-block is shown in Figure 3. There are four instances of the transformation $_K(x)$ in the Vortex sub-block. Each instance is wrapped by using a feed-forward provided by the Matyas-Meyer-Oseas construction to make the transformation non-reversible. The first two instances process input word W_0. The other two instances process the input word W_1. W_0 is the least-significant word of the current sub-block to be processed. Instances of $_K(x)$ that accept the same input word process a different variable from among A, B. Each instance treats its input variable A or B as a key and treats its input word, which is one from W_0 or W_1 as plaintext, as that is the norm in a Matyas-Meyer-Oseas construction.

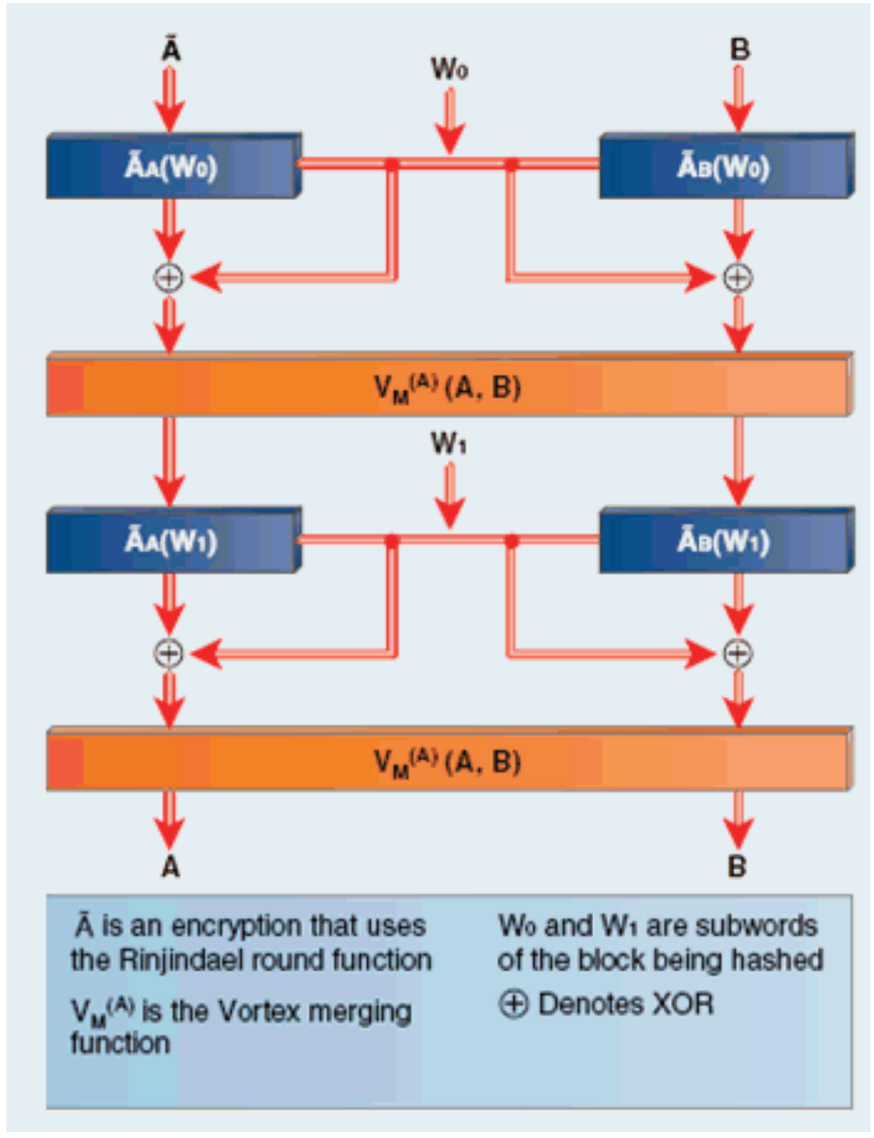

Figure 3: Vortex Sub-Block Structure (Source: Intel Corporation, 2009).

The Vortex merging function $V_M^{(A)}(A, B)$ operates as in the follwing equation:

$$V_M^{(A)}(A, B)$$
$$A_1\, A_0 \leftarrow A\,;\, B_1\, B_0 \leftarrow B$$
$$O \leftarrow A_0 \otimes B_1\,;\, I \leftarrow A_1 \otimes B_0$$
$$I_1\, I_0 \leftarrow I\,;\, O_1\, O_0 \leftarrow O$$
$$\text{output } B_1 \boxplus I_1 \,//\, B_0 \boxplus O_0 \,//\, A_1 \oplus O_1 \,//\, A_0 \oplus I_0$$

where $\boxplus$ is ordinary 64-bit addition, and $\oplus$ denotes carry-less multiplication.

The Vortex merging function (as shown in Figure 4) ensures that the bits of *A* impact the bits of *B* and vice versa. In fact, each bit of one variable affects a significant number of the bits of the other variable in a non-linear manner. This makes the design better than a straightforward XOR or other simple mathematical operation.Carry-less multiplcation is the default configuration of Vortex. The reason why Vortex uses carry-less multiplication by default is because it is easier to make analytical assertions about the collision resistance and pre-image resistance of the hash. In another configuration, Vortex uses integer multiplication. An integer multiplier increases the performance of the hash (not all processor architectures have a carry-less multiplier) and also increases the non-linearity of merging; however, it also makes the security of the scheme more difficult to prove.

The last Vortex sub-block is different. It repeats the sequence of Matyas-Meyer-Oseas transforms and merging several times. The total number of times every bit is diffused over all bits of the hash is deter-

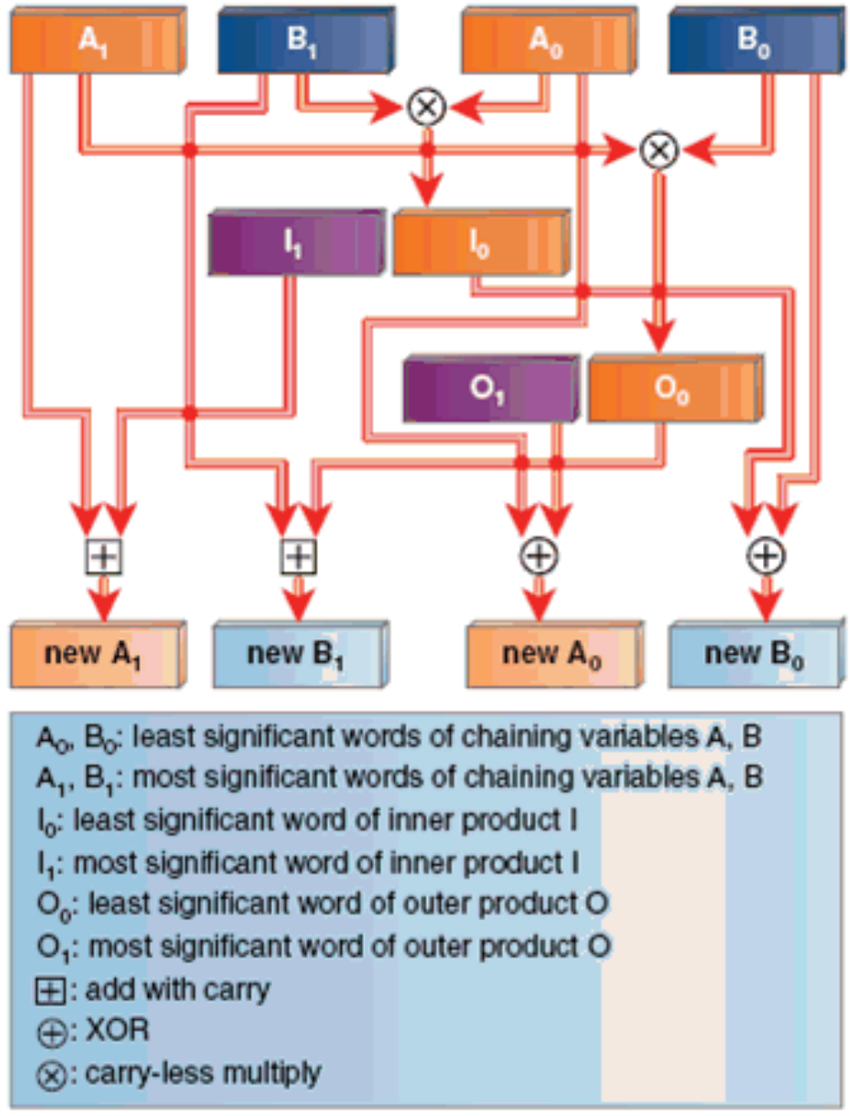

Figure 4: Vortex Merging Function (Source: Intel Corporation, 2009).

mined by the number of sequences of Rijndael rounds and merging found in the last Vortex sub-block; this is another tunable parameter of the hash.

Design Rationale and Comparison

In this section, we summarize and compare the design rationale for both algorithms.

Skein Design Rationale

The Skein team had a number of goals:

- Design for simplicity. Simple designs, rather than complex ones, are easier to optimize and analyze for security flaws.
- Maximize security per clock. Performance trumps security in practice, so extract as much security as possible from each cycle. In particular, it is important to use only the simplest instructions that have been highly optimized on every platform. Exploit the super-scalar behavior of modern processors and maximize the diffusion from each operation. Through a series of experiments the Skein team discovered that simpler round functions with more rounds optimize both performance and security.
- Achieve high performance and easy implementation on all processors. Building in a performance advantage for one processor family over another is a disadvantage in a public competition.

The Skein team made a number of critical design decisions when designing the algorithm:

- Base the design on a block cipher. This is the most conservative security choice. The community knows better how to analyze designs based on block ciphers. Moreover, there is a well-established theory on how to turn a block cipher into a hash function.
- Build your own block cipher. The Skein team obtained better performance and a simpler security analysis with a new cipher of the correct block size instead of adapting an existing narrow-block cipher.
- Make the block cipher tweakable. This introduces great flexibility in designing a new mode at no cost.
- Replace the cascade construction with something better. The cascade construction has many known defects; the tweak allows UBI to offer provable security, and it suffers from none of the cascade construction's problems.
- Use Matyas-Meyer-Oseas construction instead of Davies-Meyer. Attacks against Matyas-Meyer-Oseas-based compression functions are chosen plaintext attacks; attacks against Davies-Meyer-based compression functions are related key attacks. The community has more experience defeating chosen plaintext attacks than related key attacks.
- Do not use table lookups. Most cipher designs use a table called an S-box. Lookups in the S-box enable side-channel attacks on software implementations, where the encryption key can be read by monitoring thepower, timing, or EMI of the processor. Threefish is not subject to these attacks since it incorporates no table lookups.
- Build in three different internal state sizes and allow output of any size. This allows flexibility in the level of security available across different use cases.

- Change the design when necessary. The design should be changed when doing so allows for simpler security proofs.

Vortex Design Rationale

The Vortex team had a number of goals:

- Design for performance. Vortex uses algorithms that are implemented by using dedicated instructions in future IA processors. These are instructions for AES round computation (AES-NI) and carry-less multiplication (GFMUL-NI). The Vortex team argues that such instructions will become a trend in the industry.
- Maximize security per clock. Like Skein, Vortex extracts as much security as possible from each cycle. In particular, it uses independent AES round operations in each block. Such operations can potentially be completed in a single clock in future processors. It also introduces parallelism in the design of its compression function (two AES rounds and their key schedules can be executed in parallel); and finally, it maximizes the diffusion from each operation. Diffusion is supported by the AES round algorithms (S-box substitution, ShiftRows, MixColumns) as well as by the multiplication stage that follows.
- Achieve high performance and easy implementation on future processors. The Vortex team believes that instructions for AES round computation and carry-less multiplication will become a trend in the industry. This is because (i) several hardware vendors including IBM and Sun are either implementing or researching them; (ii) even processors for embedded systems now include AES hardware; and (iii) there is a precedence in the industry that good instruction sets are widely adopted (for example, SSE instructions).

The Vortex team made a number of critical design decisions when designing the algorithm:

- Base the design on a well-known and secure block cipher. Vortex uses an AES round as a building block. AES is a well-studied block cipher, and the AES round operation offers very good mixing across 32 bits, as a standalone operation, and 128 bits if repeated several times.
- Strengthen the AES key schedule. Hashing is a one-way operation so additions with carries are permitted in the design of a hash function. Vortex strengthens the AES key schedule by adding round constants with carries and performing S-box substitution across all round key bytes.
- Combine the outputs of two parallel AES transformations by using carry-less or integer multiplication. Multiplication is a highly nonlinear operation and can be used for destroying bit differentials.
- Replace the cascade construction with something better. Vortex uses the EMD construction and a tweak value to preserve the pseudo-random function and the pseudo-random oracle properties.
- Use Matyas-Meyer-Oseas construction instead of Davies-Meyer. This is similar to the rationale of the Skein team. The community has more experience defeating chosen plaintext attacks than related key attacks.
- Use Rijndael S-boxes instead of table lookups. Rijndael S-boxes have a special structure that allows them to be implemented by using combinatorial logic as opposed to table lookups. Thus, side-channel attacks can be averted.

Comparison of the Skein and Vortex Designs

It is now possible to highlight some similarities and differences between the Skein and Vortex designs. We begin with the similarities.

Similarities Between the Skein and Vortex Designs

Both designs are based on block ciphers, and they both use the Matyas-Meyer-Oseas construction to convert the related key attacks against the deployed hash function designs into chosen plaintext attacks. The cryptographic community has more experience defeating chosen plaintext attacks than related key attacks.

Both designs use a flavor of the cascade construction to paste together the hash of different blocks output by a compression function, and they both double hash the final output; that is, the output from the cascade construction is rehashed to become the final output. Both do this to address vulnerabilities that arise from processing the final block with a construction such as Davies-Meyer or Matyas-Meyer-Oseas, within the cascade construction.

Differences Between the Skein and Vortex Designs

The differing design approaches reflect the differing skill sets of the two teams. The Skein team members were skilled in designing block ciphers. In contrast, the Vortex team members did not design a new block cipher, but instead used an existing one. The Skein design allows its security claims to be derived from the security of a block cipher. The Vortex design effort emerged from the need to demonstrate a secure hash function by using the new AES round and carry-less multiplication instructions for future IA processors, which the Vortex team members designed.

Skein uses significantly more rounds (72/72/80) than Vortex, stemming from the different design decisions made by each team. The Skein team's experiments indicated diffusion per clock is maximized by numerous simple rounds. The Vortex designers instead were motivated by performance; they believed the best performance is achieved by using fewer rounds, based on very powerful diffusion primitives.

Skein's support for a hash value of any length from 1 to 264 bytes allowed the team to prove the property that Skein cannot be differentiated from a random oracle if Threefish acts like an ideal cipher. There are two ways to think about any algorithm: (1) as a monolithic black box, where you have no knowledge of any of the algorithm's internal parts, and (2) as presented in this article, where we know the details of the algorithm's internal structure. By saying that Skein cannot be differentiated from a random oracle, therefore, we mean that it is impossible, even in principle, for Skein to construct any statistical test that exploits differences in the two views. This means that Skein is structurally sound and that its security depends only on the security of Threefish. Vortex's final output is of a fixed length, but the second hash allows it to act like a fixed-length random oracle.

None of the Skein components use table lookups such as S-boxes, so it is more difficult to launch side-channel attacks on Skein than on specific implementations of Vortex. Vortex avoids these attacks by relying on a hardware logic implementation for the AES S-boxes.

Summary

In this article, we reviewed the theory of hash functions, the state of knowledge about them, and some of Intel's contributions to this field of research. Hash functions are basic building blocks that are central to cryptography's mission, but recent attacks by Joux and Wang have

undermined our confidence in classical constructions. Because of this insecurity, there has been a wave of new research into hash functions, and Intel has been at the forefront, with two independent and radically different submissions to the international NIST hash competition.

References

[1] A. Joux. "Iterated Collisions on Iterated Hash Functions." Crypto 2004, *Lecture Notes in Computer Science (LNCS) 3621*, Springer-Verlag, Berlin, 2005.

[2] X. Wang, Y. Yin, and H. Yu. "Finding Collisions in the full SHA-1." *Crypto 2005, LNCS 2947*, Springer-Verlag, Berlin, 2004

[3] A. Lenstra, X. Wang, and B. de Berger. "Colliding X.509 Certificates." http://eprint.iacr.org

[4] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[5] R. Merkle. "Secrecy, authentication, and public key systems." Stanford University Ph.D. thesis, 1979. Available as *"Technical Report No. 1979-1,"* Information Systems Laboratory, Stanford University, Palo Alto, California, 1979.

[6] I. Damgrd. "A Design Principle for Hash Functions." *Crypto 1989, LNCS 435*, Springer-Verlag, Berlin, 1989.

[7] R. Winternitz. "A secure one-way hash function built from DES." *Proceedings of IEEE Symposium on Security and Privacy*, 1984.

[8] S. Matyas, C. Meyer, and J. Oseas. "Generating strong one-way functions with cryptographic algorithms." *IBM Technical Bulletin, 27,* 1985.

[9] J. Black, P. Rogaway, and T. Shrimpton. "Black box analysis of block-cipher-based hash functions from PGV." *Crypto 2002, LNCS 2442*, Springer-Verlag, Berlin, 2002.

[10] The National Institute of Standards and Technology. US Government agency sponsor for competition on next-generation hash design. http://csrc.nist.gov

[11] M. Liskov, R. Rivest, and D. Wagner. "Tweakable Block Ciphers." *Crypto, LNCS 2442*, Springer-Verlag, Berlin, 2002.

[12] National Security Agency. Skipjack and KEA Specifications. May 29, 1998. http://csrc.nist.gov

[13] M. Kounavis and S. Gueron. "Vortex: A new Family of One-Way Hash Functions Based on Rijndael Rounds and Carry-less Multiplication." http://eprint.iacr.org

[14] S. Gueron. "Advanced Encryption Standard (AES) Instructions Set." http://software.intel.com

[15] S. Gueron and M. Kounavis. "Carry-Less Multiplication and its Usage for Computing the GCM Mode." http://software.intel.com

This article and more on similar subjects may also be found in the *Intel Technology Journal*, June 2009 Edition, "Advances in Internet Security."

— When Dr. Dobb's first published this article in 2009, Jesse Walker was an Applied Cryptographer in Intel's Communication Technology Laboratory; Michael E. Kounavis was a Senior Research Scientist working with Intel Labs.; Shay Gueron was an Intel Principal Engineer; and Gary Graunke was a Senior Staff Architect for cryptography at Intel.

This Month on DrDobbs.com

Interesting items posted on www.drdobbs.com over the past month that you may have missed

VISUALVM FOR JAVA DEVELOPMENT

Eric Bruno is continually surprised to see so many Java developers unaware of, or just unwilling to use, the many tools available to them.
http://drdobbs.com/blogs/java/229403052

GETTING STARTED WITH GOOGLE APPS AND OAUTH

In this second installment of coding for the cloud, Allen Holub examines the first step in accessing a cloud app: authorization.
http://drdobbs.com/web-development/229401853

FAST, PARALLELIZED CRC COMPUTATION USING THE NEHALEM CRC32 INSTRUCTION

A fast and efficient method of computing a Cyclic Redundancy Check (CRC) on Intel processors for the fixed (degree-32) iSCSI polynomial, using the CRC32 instruction introduced in Intel Core i7 processors.
http://drdobbs.com/high-performance-computing/229401411

JAVA 7 AND NETBEANS 7

The Javasphere is settling into two recent announcements by Oracle: The features list for Java EE 7 has been finalized and version 7.0 of the NetBeans IDE has shipped.
http://drdobbs.com/open-source/229401884

WORLD'S LARGEST DATABASE: THE WEB OF DATA

While "Big Data" has been a hot button for venture capitalists and the trade press, there have been several years of progress in the "Biggest Data" arena. The Semantic Web meme, with some prodding by the World Wide Web Consortium (W3C), has morphed into the Linked Data meme. And the notion of building the world's largest database has gained traction and continues to move forward.
http://drdobbs.com/blogs/database/229402861

QT APPLICATION DEVELOPMENT FOR SYMBIAN

A full native application development environment for Symbian smart phones with Qt.
http://drdobbs.com/mobility/229301197

DESIGNING FOR RELIABILITY

Whether you are designing a data center or a manned space tourism vehicle, you will probably design in redundancy to ensure reliability.
http://drdobbs.com/tools/229401344

C/C++ PROGRAMMING IN A UNIX ENVIRONMENT

How long did it take you to learn C++?
http://drdobbs.com/blogs/cpp/229402491

IN THIS ISSUE

Dr.Dobb's

Andrew Binstock Executive Editor, Dr. Dobb's
alb@drdobbs.com

Deirdre Blake Managing Editor, Dr. Dobb's
dblake@techweb.com

Amy Stephens Copyeditor, Dr. Dobb's
astephens@techweb.com

Sean Coady Webmaster, Dr. Dobb's
scoady@techweb.com

Jon Erickson Editor in Chief Emeritus , Dr. Dobb's

DR DOBB'S
UBM TECHWEB
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

Dr.Dobb's Business Contacts

DR. DOBB'S

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Account Executive, Shaina Guttman
(212) 600-3106, sguttman@techweb.com

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

CMO, Scott Vaughan
(949) 223-3662, svaughan@techweb.com

VP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz
(212) 600-3015, mschwartz@techweb.com

Sales Assistant, Group Sales, Kelly Glass
(212) 600-3327, kglass@techweb.com

Publisher's Assistant, Esther Rodriguez
(949) 223-3656, erodriguez@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Western Regional Director, Matt Stovall
(415) 947-6245, mstovall@techweb.com

District Sales Manager, Rachel Calderon
(516) 562-5338, rcalderon@techweb.com

Inside Sales Manager, Vesna Beso
(415) 947-6104, vbeso@techweb.com

Sales Assistant, Ian Doyle
(415) 947-6105, idoyle@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, skupiec@techweb.com

Account Manager, Shoshana Freisinger
(415) 947-6349, sfreisinger@techweb.com

Sales Assistant, Matthew Cohen-Meyer
(415) 947-6214, mmeyer@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Jenny Hanna
(516) 562-5116, jhanna@techweb.com

District Manager, Michael Greenhut
(516) 562-5044, mgreenhut@techweb.com

Account Manager, Cori Gordon
(516) 562-5181, cgordon@techweb.com

Inside Sales Manager East, Ray Capitelli
(212) 600-3045, rcapitelli@techweb.com

Sales Assistant, Elyse Cowen
(212) 600-3051, ecowen@techweb.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
(212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
(516) 562-5651, jgambino@techweb.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kjurina@techweb.com

Dr. Dobb's

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Account Executive, Shaina Guttman
(212) 600 3106, sguttman@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, wng@techweb.com

Director of Marketing, Sherbrooke Balser
(949) 223-3605, sbalser@techweb.com

Marketing Manager, Monique Luttrell
(949) 223-3609, mluttrell@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000

UBM TECHWEB

Tony L. Uphoff CEO

John Dennehy CFO

David Michael CIO

Bob Evans Sr. VP and Global CIO Director

Joseph Braue Sr. VP, Light Reading Communications Network

Scott Vaughan CMO

Ed Grossman Executive Vice President, InformationWeek Business Technology Network

John Ecke VP and Group Publisher, Financial Technology Network, InformationWeek Government, InformationWeek Healthcare

Martha Schwartz VP, Group Sales, InformationWeek Business Technology Network

Beth Rivera Senior VP, Human Resources

David Berlind Chief Content Officer, TechWeb, and Editor in Chief, TechWeb.com

Fritz Nelson VP, Editorial Director, InformationWeek Business Technology Network, and Executive Producer, TechWeb TV

UNITED BUSINESS MEDIA LLC

Pat Nohilly Sr. VP, Strategic Development and Business Admin.

Marie Myers Sr. VP, Manufacturing