JS Review

1/9

PHP actually has default input validation
which one could turn off

like JSON encode
can't just add slashes
JS as HTML entity

Untrusted code uses JS embed

Even CSS!

(There is very little of a good guide on this)

Google's Caja JS lib
Can also protect <frames>

AntiSamy
(not well supported

(7)

Submit w/o iframe

Have to do w/ JS AJAX
No call back for, form.submit()
So ~~same~~ problem is ~~CSRF~~ Same origin
Actually another poster noticed hidden iframe
and can use onload handle

(Have never read the languages officially)

Same origin

Cross origin resource sharing
Access-Control-Allow-Origin header
└ for the site being requested
Window.postMessage
Can add listeners
Or reverse proxy

---

but <script> tag is not restricted
JSONP

(3)

JSONP

prefered method from diff domain

Use script tag

but can be any JS code!

JS interpeter not JSON parser

CORS is a fix

callback({ json3 })

Tgenerally provide in URL

Could also be a variable

Via script element injection

if jQuery can do

but it can inset any code

want special restricted mine type

(4)

(how can jquery add a function to main code)

- add <script> w/ text </script> tag

Create function objects

Constructors + prototypes

(I should know these patterns better)

Prototype

property that gets ~~define~~ created when define function

Can add constructor when make new

new Gadget ('webcam', 'black')

* objects passed by refrence

Prototype = object from which other obj inherits properties

Prototype property

(5)

Constructor property is set to prototype property at fn creation

ECMA-262

by Dmitry Soshnikov

JavaScript. The core.

Read this article in: Chinese, Japanese, German, Arabic, Russian, Korean, French.

1. An object
2. A prototype chain
3. Constructor
4. Execution context stack
5. Execution context
6. Variable object
7. Activation object
8. Scope chain
9. Closures
10. This value
11. Conclusion

This note is an overview and summary of the "ECMA-262-3 in detail" series. Every section contains references to the appropriate matching chapters so you can read them to get a deeper understanding.

Intended audience: experienced programmers, professionals.

We start out by considering the concept of an *object*, which is fundamental to ECMAScript.

An object

ECMAScript, being a highly-abstracted object-oriented language, deals with *objects*. There are also *primitives*, but they, when needed, are also converted to objects.

An object is a *collection of properties* and has a *single prototype object*. The prototype may be either an object or the `null` value.

Let's take a basic example of an object. A prototype of an object is referenced by the internal `[[Prototype]]` property. However, in figures we will use `__<internal-property>__` underscore notation instead of the double brackets, particularly for the prototype object: `__proto__` (which is a real, but non-standard, feature in some engines, e.g. SpiderMonkey).

For the code:

```
var foo = {
  x: 10,
```

```
  y: 20
};
```

we have the structure with two explicit *own* properties and one implicit `__proto__` property, which is the reference to the prototype of `foo`:

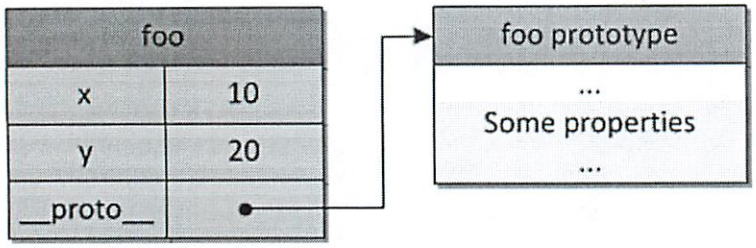


Figure 1. A basic object with a prototype.

What for these prototypes are needed? Let's consider a *prototype chain* concept to answer this question.

A prototype chain

Prototype objects are also just simple objects and may have their own prototypes. If a prototype has a non-null reference to its prototype, and so on, this is called the *prototype chain*.

A prototype chain is a *finite* chain of objects which is used to implement *inheritance* and *shared properties*.

Consider the case when we have two objects which differ only in some small part and all the other part is the same for both objects. Obviously, for a good designed system, we would like to *reuse* that similar functionality/code without repeating it in every single object. In class-based systems, this *code reuse* stylistics is called the *class-based inheritance* — you put similar functionality into the class `A`, and provide classes `B` and `C` which inherit from `A` and have their own small additional changes.

ECMAScript has no concept of a class. However, a code reuse stylistics does not differ much (though, in some aspects it's even more flexible than class-based) and achieved via the *prototype chain*. This kind of inheritance is called a *delegation based inheritance* (or, closer to ECMAScript, a *prototype based inheritance*).

Similarly like in the example with classes `A`, `B` and `C`, in ECMAScript you create objects: `a`, `b`, and `c`. Thus, object `a` stores this common part of both `b` and `c` objects. And `b` and `c` store just their own additional properties or methods.

```
var a = {
  x: 10,
  calculate: function (z) {
    return this.x + this.y + z
  }
};

var b = {
  y: 20,
  __proto__: a
};

var c = {
```

```
  y: 30,
  __proto__: a
};

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80
```

ahh can just do that

Easy enough, isn't it? We see that `b` and `c` have access to the `calculate` method which is defined in `a` object. And this is achieved exactly via this prototype chain.

Makes sense!

The rule is simple: if a property or a method is not found in the object itself (i.e. the object has no such an *own* property), then there is an attempt to find this property/method in the prototype chain. If the property is not found in the prototype, then a prototype of the prototype is considered, and so on, i.e. the whole prototype chain (absolutely the same is made in class-based inheritance, when resolving an inherited *method* — there we go through the *class chain*). The first found property/method with the same name is used. Thus, a found property is called *inherited* property. If the property is not found after the whole prototype chain lookup, then `undefined` value is returned.

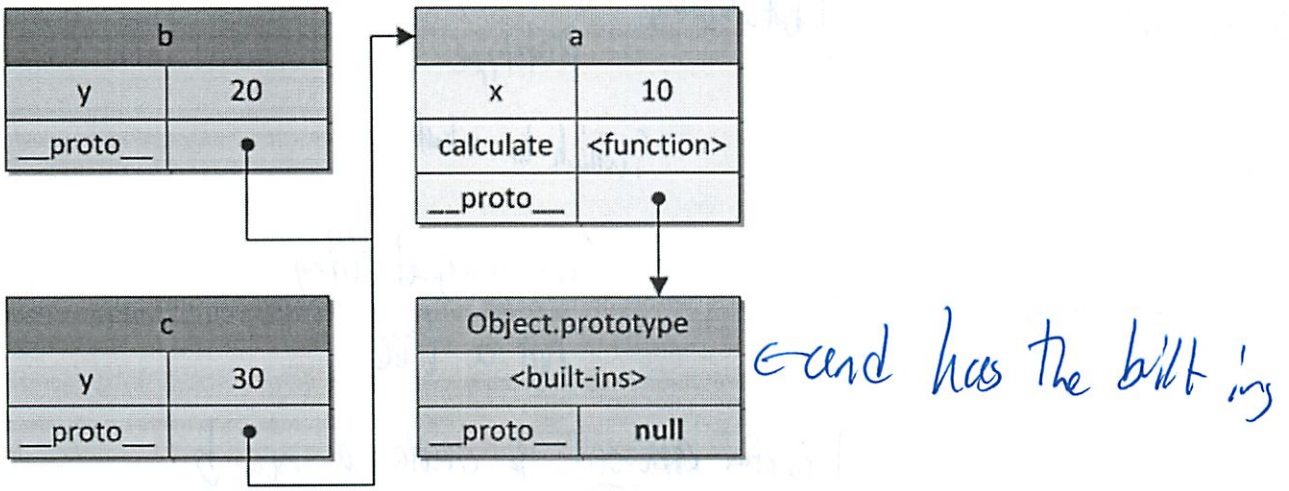
good to see specific terms

Notice, that `this` value in using an inherited method is set to the *original* object, but not to the (prototype) object in which the method is found. I.e. in the example above `this.y` is taken from `b` and `c`, but not from `a`. However, `this.x` is taken from `a`, and again via the *prototype chain* mechanism.

If a prototype is not specified for an object explicitly, then the default value for `__proto__` is taken — `Object.prototype`. Object `Object.prototype` itself also has a `__proto__`, which is the *final link* of a chain and is set to `null`.

Ok that is end of chain

The next figure shows the inheritance hierarchy of our `a`, `b` and `c` objects:



end has the built ins

Figure 2. A prototype chain.

Often it is needed to have objects with the *same or similar state structure* (i.e. the same set of properties), and with different *state values*. In this case we may use a *constructor function* which produces objects by *specified pattern*.

Constructor

Besides creation of objects by specified pattern, a *constructor* function does another useful thing — it

automatically sets a prototype object for newly created objects. This prototype object is stored in the `ConstructorFunction.prototype` property.

E.g., we may rewrite previous example with `b` and `c` objects using a constructor function. Thus, the role of the object `a` (a prototype) `Foo.prototype` plays:

```
// a constructor function
function Foo(y) {
  // which may create objects
  // by specified pattern: they have after
  // creation own "y" property
  this.y = y;
}

// also "Foo.prototype" stores reference
// to the prototype of newly created objects,
// so we may use it to define shared/inherited
// properties or methods, so the same as in
// previous example we have:

// inherited property "x"
Foo.prototype.x = 10;

// and inherited method "calculate"
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};

// now create our "b" and "c"
// objects using "pattern" Foo
var b = new Foo(20);
var c = new Foo(30);

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80

// let's show that we reference
// properties we expect

console.log(

  b.__proto__ === Foo.prototype, // true
  c.__proto__ === Foo.prototype, // true

  // also "Foo.prototype" automatically creates
  // a special property "constructor", which is a
  // reference to the constructor function itself;
  // instances "b" and "c" may found it via
  // delegation and use to check their constructor

  b.constructor === Foo, // true
  c.constructor === Foo, // true
  Foo.prototype.constructor === Foo // true

  b.calculate === b.__proto__.calculate, // true
  b.__proto__.calculate === Foo.prototype.calculate // true

);
```

This code may be presented as the following relationship:

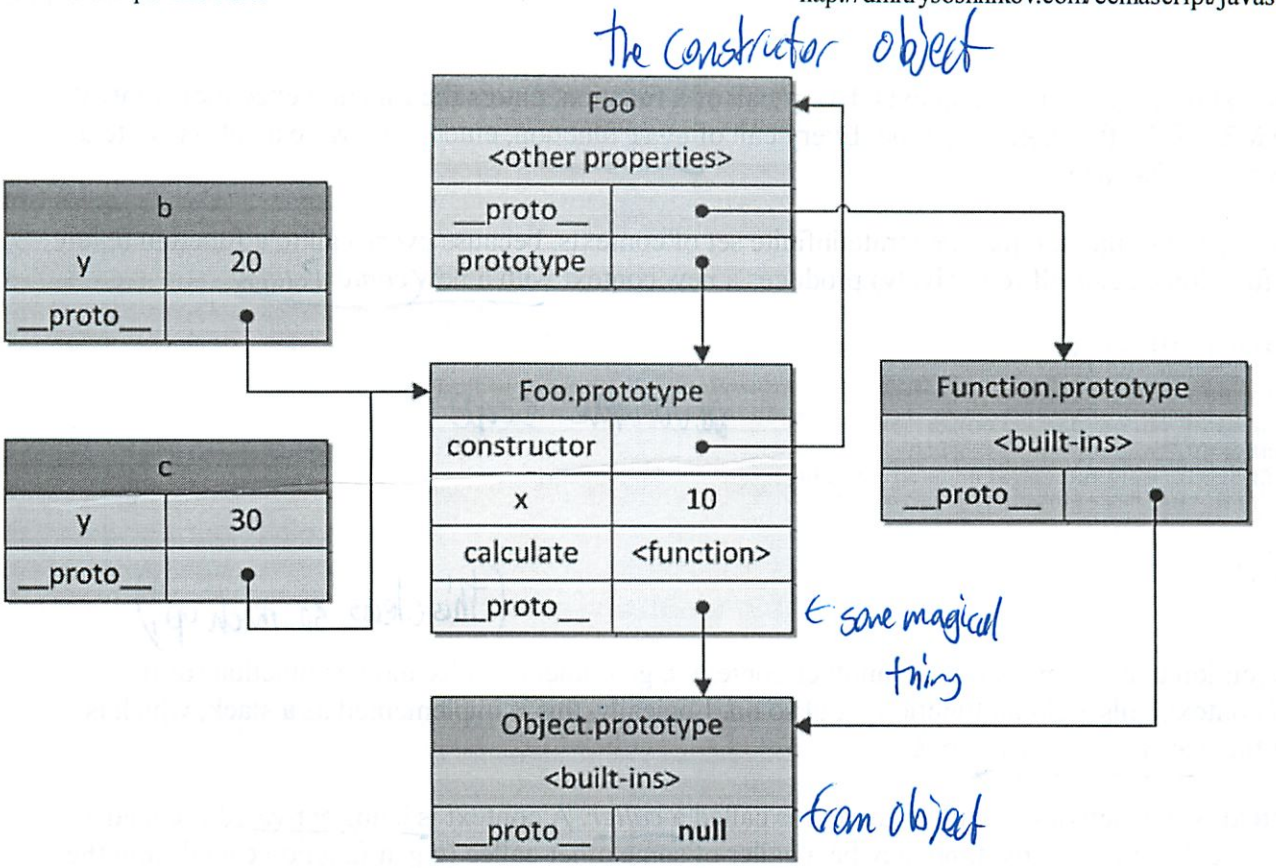


Figure 3. A constructor and objects relationship.

This figure again shows that every object has a prototype. Constructor function `Foo` also has its own `__proto__` which is `Function.prototype`, and which in turn also references via its `__proto__` property again to the `Object.prototype`. Thus, repeat, `Foo.prototype` is just an explicit property of `Foo` which refers to the prototype of `b` and `c` objects.

Formally, if to consider a concept of a *classification* (and we've exactly just now *classified* the new separated thing — `Foo`), a combination of the constructor function and the prototype object may be called as a "class". Actually, e.g. Python's *first-class* dynamic classes have absolutely the same implementation of properties/methods resolution. From this viewpoint, classes of Python are just a syntactic sugar for delegation based inheritance used in ECMAScript.

The complete and detailed explanation of this topic may be found in the Chapter 7 of ES3 series. There are two parts: Chapter 7.1. OOP. The general theory, where you will find description of various OOP paradigms and stylistics and also their comparison with ECMAScript, and Chapter 7.2. OOP. ECMAScript implementation, devoted exactly to OOP in ECMAScript.

Now, when we know basic object aspects, let's see on how the *runtime program execution* is implemented in ECMAScript. This is what is called an *execution context stack*, every element of which is abstractly may be represented as also an object. Yes, ECMAScript almost everywhere operates with concept of an object ☺

Execution context stack

There are three types of ECMAScript code: *global* code, *function* code and *eval* code. Every code is evaluated in its *execution context*. There is only one global context and may be many instances of

function and *eval* execution contexts. Every call of a function, enters the function execution context and evaluates the function code type. Every call of `eval` function, enters the *eval* execution context and evaluates its code.

Notice, that one function may generate infinite set of contexts, because every call to a function (even if the function calls itself recursively) produces a new context with a new *context state*:

```
function foo(bar) {}

// call the same function,
// generate three different
// contexts in each call, with
// different context state (e.g. value
// of the "bar" argument)

foo(10);
foo(20);
foo(30);
```

An execution context may activate another context, e.g. a function calls another function (or the global context calls a global function), and so on. Logically, this is implemented as a stack, which is called the *execution context stack*.

A context which activates another context is called a *caller*. A context is being activated is called a *callee*. A callee at the same time may be a caller of some other callee (e.g. a function called from the global context, calls then some inner function).

When a caller activates (calls) a callee, the caller suspends its execution and passes the control flow to the callee. The callee is pushed onto the the stack and is becoming a *running (active)* execution context. After the callee's context ends, it returns control to the caller, and the evaluation of the caller's context proceeds (it may activate then other contexts) till the its end, and so on. A callee may simply *return* or exit with an *exception*. A thrown but not caught exception may exit (pop from the stack) one or more contexts.

I.e. all the ECMAScript *program runtime* is presented as the *execution context (EC) stack*, where *top* of this stack is an *active* context:

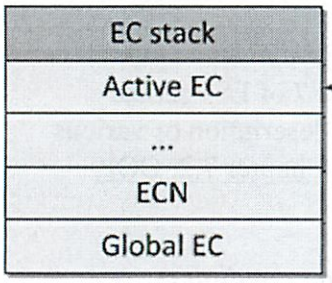


Figure 4. An execution context stack.

When program begins it enters the *global execution context*, which is the *bottom* and the *first* element of the stack. Then the global code provides some initialization, creates needed objects and functions. During the execution of the global context, its code may activate some other (already created) function, which will enter their execution contexts, pushing new elements onto the stack, and so on. After the initialization is done, the runtime system is waiting for some *event* (e.g. user's mouse click) which will activate some function and which will enter a new execution context.

JavaScript, JavaScript…

by Angus Croll

Read 11/9

Function Declarations vs. Function Expressions

Let's start with a short quiz. What is alerted about these 2 ads?) ← opps

Question 1:

What is returned

```
01 function foo(){
02     function bar() {
03         return 3;
04     }
05     return bar();
06     function bar() {
07         return 8;
08     }
09 }
10 alert(foo());
```

3 8

Question 2:

```
01 function foo(){
02     var bar = function() {
03         return 3;
04     };
05     return bar();
06     var bar = function() {
07         return 8;
08     };
09 }
10 alert(foo());
```

3 3

Question 3:

```
01 alert(foo());
02 function foo(){
03     var bar = function() {
04         return 3;
05     };
```

3

```
    return bar();
    var bar = function() {
        return 8;
    };
}
```

Question 4:

```
function foo(){
    return bar();
    var bar = function() {
        return 3;
    };
    var bar = function() {
        return 8;
    };
}
alert(foo());
```

If you didn't answer 8, 3, 3 and [Type Error: bar is not a function] respectively, read on... (actually read on anyway ☺)

What is a Function Declaration?

A Function Declaration defines a named function variable without requiring variable assignment. Function Declarations occur as standalone constructs and cannot be nested within non-function blocks. It's helpful to think of them as siblings of Variable Declarations. Just as Variable Declarations must start with "var", Function Declarations must begin with "function".

e.g.

```
function bar() {
    return 3;
}
```

ECMA 5 (13.0) defines the syntax as
function *Identifier* (*FormalParameterList*$_{opt}$) { *FunctionBody* }

The function name is visible within it's scope and the scope of it's parent (which is good because otherwise it would be unreachable)

```
function bar() {
    return 3;
}

bar() //3
bar  //function
```

What is a Function Expression?

A Function Expression defines a function as a part of a larger expression syntax (typically a variable assignment). Functions defined via Functions Expressions can be named or anonymous. Function

Expressions must not start with "function" (hence the parentheses around the self invoking example below)

e.g.

```
01 //anonymous function expression
02 var a = function() {
03     return 3;
04 }
05
06 //named function expression
07 var a = function bar() {
08     return 3;
09 }
10
11 //self invoking function expression
12 (function sayHello() {
13     alert("hello!");
14 })();
```

ECMA 5 (13.0) defines the syntax as
function *Identifier*$_{opt}$ (*FormalParameterList*$_{opt}$) { *FunctionBody* }

(though this feels incomplete since it omits the requirement that the containing syntax be an expression and not start with "function")

The function name (if any) is not visible outside of it's scope (contrast with Function Declarations).

So what's a Function Statement?

Its sometimes just a pseudonym for a Function Declaration. However as kangax (http://yura.thinkweb2.com/named-function-expressions/#function-statements) pointed out, in mozilla a Function Statement is an extension of Function Declaration allowing the Function Declaration syntax to be used anywhere a statement is allowed. It's as yet non standard so not recommended for production development

About that quiz….care to explain?

OK so Question 1 uses function declarations which means they get hoisted…

Wait, what's Hoisting?

To quote Ben Cherry's excellent article (http://www.adequatelygood.com/2010/2/JavaScript-Scoping-and-Hoisting): "Function declarations and function variables are always moved ('hoisted') to the top of their JavaScript scope by the JavaScript interpreter".

When a function declaration is hoisted the entire function body is lifted with it, so after the interpreter has finished with the code in Question 1 it runs more like this:

```
01 //**Simulated processing sequence for Question 1**
02 function foo(){
03     //define bar once
```

```
    function bar() {
        return 3;
    }
    //redefine it
    function bar() {
        return 8;
    }
    //return its invocation
    return bar(); //8
}
alert(foo());
```

But...but...we were always taught that code after the return statement is unreachable

In JavaScript execution there is Context (which ECMA 5 breaks into LexicalEnvironment, VariableEnvironment and ThisBinding) and Process (a set of statements to be invoked in sequence). Declarations contribute to the VariableEnvironment when the execution scope is entered. They are distinct from Statements (such as **return**) and are not subject to their rules of process.

Do Function Expressions get Hoisted too?

That depends on the expression. Let's look at the first expression in Question 2:

```
var bar = function() {
    return 3;
};
```

The left hand side (*var bar*) is a Variable Declaration. Variable Declarations get hoisted but their Assignment Expressions don't. So when **bar** is hoisted the interpreter initially sets *var bar = undefined*. The function definition itself is not hoisted.

(ECMA 5 12.2 A variable with an *initialzier* is assigned the value of its *AssignmentExpression* when the *VariableStatement* is executed, not when the variable is created.)

Thus the code in Question 2 runs in a more intuitive sequence:

```
//**Simulated processing sequence for Question 2**
function foo(){
    //a declaration for each function expression
    var bar = undefined;
    var bar = undefined;
    //first Function Expression is executed
    bar = function() {
        return 3;
    };
    // Function created by first Function Expression is invoked
    return bar();
    // second Function Expression unreachable
}
alert(foo()); //3
```

Ok I think that makes sense. By the way, you're wrong about Question 3. I ran it in Firebug and

In the next figure, having some function context as `EC1` and the global context as `Global EC`, we have the following stack modification on entering and exiting `EC1` from the global context:

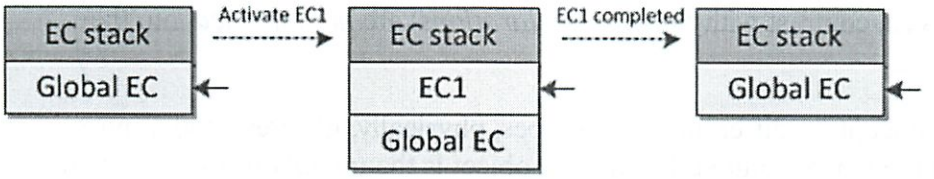


Figure 5. An execution context stack changes.

This is exactly how the runtime system of ECMAScript manages the execution of a code.

More information on execution context in ECMAScript may be found in the appropriate Chapter 1. Execution context.

As we said, every execution context in the stack may be presented as an object. Let's see on its structure and what kind of *state* (which properties) a context is needed to execute its code.

Execution context

An execution context abstractly may be represented as a simple object. Every execution context has set of properties (which we may call a *context's state*) necessary to track the execution progress of its associated code. In the next figure a structure of a context is shown:

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[Variable object + all parent scopes]
thisValue	Context object

Figure 6. An execution context structure.

Besides these three needed properties (a *variable object*, a `this` *value* and a *scope chain*), an execution context may have any additional state depending on implementation.

Let's consider these important properties of a context in detail. what is variable scope in JS?

Variable object

A *variable object* is a *scope of data* related with the execution context. It's a special object associated

with the context and which stores *variables* and *function declarations* are being defined within the context.

Notice, that *function expressions* (in contrast with *function declarations*) are *not included* into the variable object.

A variable object is an abstract concept. In different context types, physically, it's presented using different object. For example, in the global context the variable object is the *global object itself* (that's why we have an ability to refer global variables via property names of the global object).

Let's consider the following example in the global execution context:

```
var foo = 10;

function bar() {} // function declaration, FD
(function baz() {}); // function expression, FE

console.log(
  this.foo == foo, // true
  window.bar == bar // true
);

console.log(baz); // ReferenceError, "baz" is not defined
```

Then the global context's variable object (VO) will have the following properties:

Global VO	
foo	10
bar	<function>
<built-ins>	

Figure 7. The global variable object.

See again, that function `baz` being a *function expression* is not included into the variable object. That's why we have a `ReferenceError` when trying to access it outside the function itself.

Notice, that in contrast with other languages (e.g. C/C++) in ECMAScript *only functions* create a new scope. Variables and inner functions defined within a scope of a function are not visible directly outside and do not pollute the global variable object.

Using `eval` we also enter a new (eval's) execution context. However, `eval` uses either global's variable object, or a variable object of the caller (e.g. a function from which `eval` is called).

And what about functions and their variable objects? In a function context, a variable object is presented as an *activation object*.

Activation object

When a function is *activated* (called) by the caller, a special object, called an *activation object* is created. It's filled with *formal parameters* and the special `arguments` object (which is a map of formal

parameters but with index-properties). The *activation object* then is used as a *variable object* of the function context.

I.e. a function's variable object is the same simple variable object, but besides variables and function declarations, it also stores formal parameters and `arguments` object and called the *activation object*.

Considering the following example:

```
function foo(x, y) {
  var z = 30;
  function bar() {} // FD
  (function baz() {}); // FE
}

foo(10, 20);
```

we have the next activation object (AO) of the `foo` function context:

Activation object	
x	10
y	20
arguments	{0: 10, 1: 20, ..}
z	30
bar	<function>

Figure 8. An activation object.

And again the *function expression* `baz` is not included into the variable/activate object.

The complete description with all subtle cases (such as *"hoisting"* of variables and function declarations) of the topic may be found in the same name Chapter 2. Variable object.

And we are moving forward to the next section. As is known, in ECMAScript we may use *inner functions* and in these inner functions we may refer to variables of *parent* functions or variables of the *global* context. As we named a variable object as a *scope object* of the context, similarly to the discussed above prototype chain, there is so-called a *scope chain*.

Scope chain

A *scope chain* is a *list of objects* that are searched for *identifiers* appear in the code of the context.

The rule is again simple and similar to a prototype chain: if a variable is not found in the own scope (in the own variable/activation object), its lookup proceeds in the parent's variable object, and so on.

Regarding contexts, identifiers are: *names* of variables, function declarations, formal parameters, etc. When a function refers in its code the identifier which is not a local variable (or a local function or a formal parameter), such variable is called a *free variable*. And to *search these free variables* exactly a

scope chain is used.

In general case, a *scope chain* is a list of all those *parent variable objects*, *plus* (in the front of scope chain) the function's *own variable/activation object*. However, the scope chain may contain also any other object, e.g. objects dynamically added to the scope chain during the execution of the context — such as *with-objects* or special objects of *catch-clauses*.

When *resolving* (looking up) an identifier, the scope chain is searched starting from the activation object, and then (if the identifier isn't found in the own activation object) up to the top of the scope chain — repeat, the same just like with a prototype chain.

```
var x = 10;

(function foo() {
  var y = 20;
  (function bar() {
    var z = 30;
    // "x" and "y" are "free variables"
    // and are found in the next (after
    // bar's activation object) object
    // of the bar's scope chain
    console.log(x + y + z);
  })();
})();
```

We may assume the linkage of the scope chain objects via the implicit `__parent__` property, which refers to the next object in the chain. This approach may be tested in a real Rhino code, and exactly this technique is used in ES5 *lexical environments* (there it's named an `outer` link). Another representation of a scope chain may be a simple array. Using a `__parent__` concept, we may represent the example above with the following figure (thus parent variable objects are saved in the `[[Scope]]` property of a function):

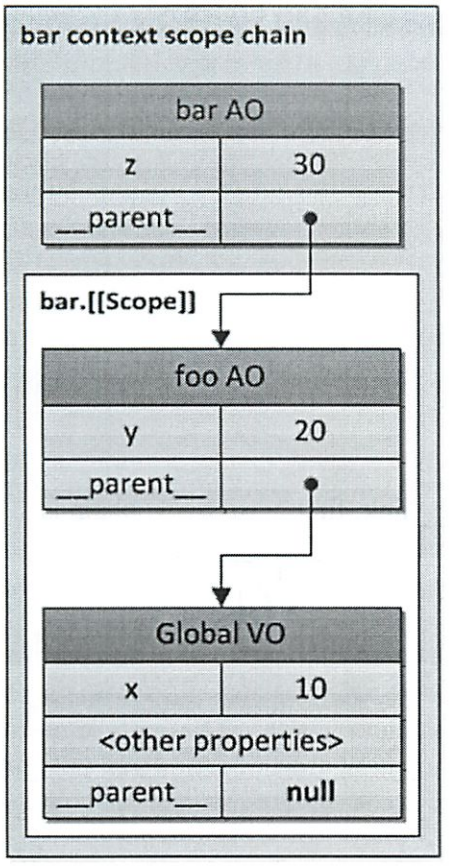


Figure 9. A scope chain.

At code execution, a scope chain may be augmented using `with` statement and `catch` clause objects. And since these objects are simple objects, they may have prototypes (and prototype chains). This fact leads to that scope chain lookup is *two-dimensional*: (1) first a scope chain link is considered, and then (2) on every scope chain's link — into the depth of the link's prototype chain (if the link of course has a prototype).

For this example:

```
Object.prototype.x = 10;

var w = 20;
var y = 30;

// in SpiderMonkey global object
// i.e. variable object of the global
// context inherits from "Object.prototype",
// so we may refer "not defined global
// variable x", which is found in
// the prototype chain

console.log(x); // 10

(function foo() {

  // "foo" local variables
  var w = 40;
  var x = 100;

  // "x" is found in the
  // "Object.prototype", because
  // {z: 50} inherits from it
```

```
  with ({z: 50}) {
    console.log(w, x, y , z); // 40, 10, 30, 50
  }

  // after "with" object is removed
  // from the scope chain, "x" is
  // again found in the AO of "foo" context;
  // variable "w" is also local
  console.log(x, w); // 100, 40

  // and that's how we may refer
  // shadowed global "w" variable in
  // the browser host environment
  console.log(window.w); // 20

})();
```

we have the following structure (that is, before we go to the `__parent__` link, first `__proto__` chain is considered):

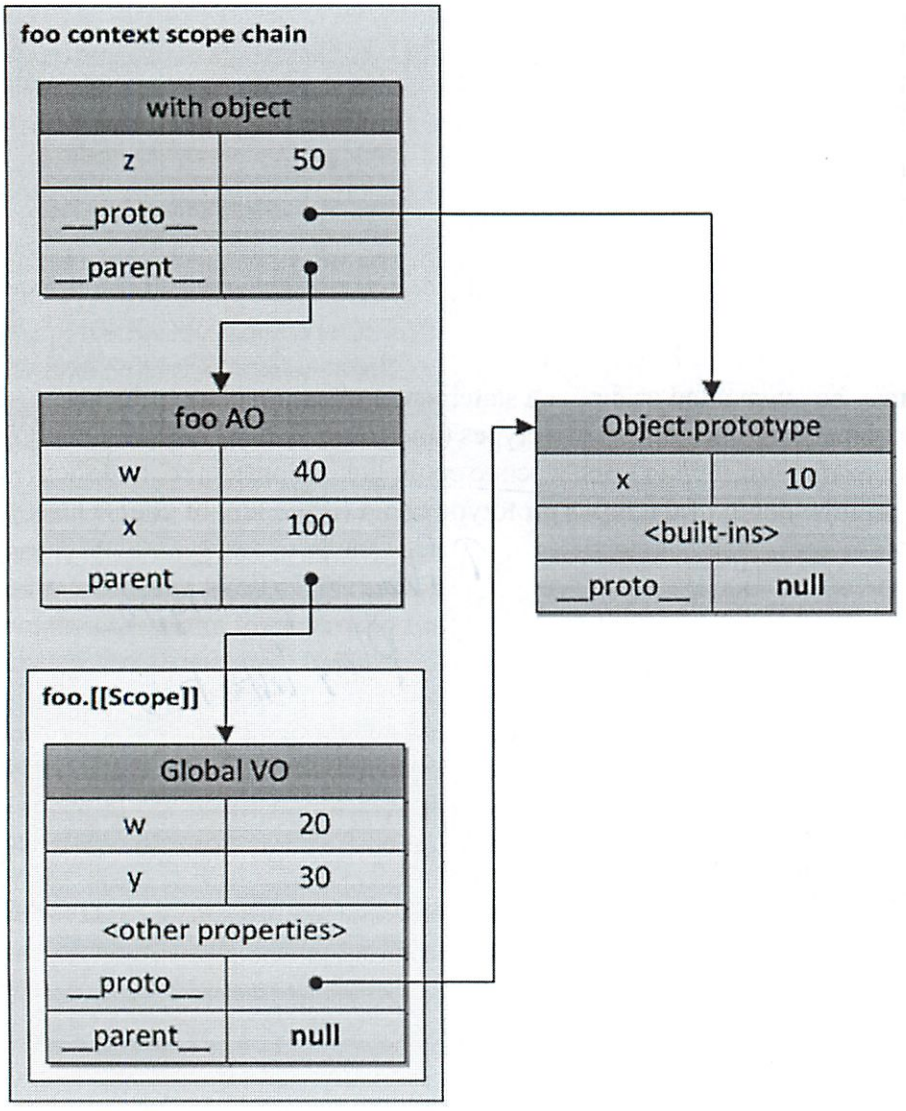


Figure 10. A "with-augmented" scope chain.

Notice, that not in all implementations the global object inherits from the `Object.prototype`. The

behavior described on the figure (with referencing "non-defined" variable `x` from the global context) may be tested e.g. in SpiderMonkey.

Until all parent variable objects exist, there is nothing special in getting parent data from the inner function — we just traverse through the scope chain resolving (searching) needed variable. However, as we mentioned above, after a context ends, all its state and it itself are *destroyed*. At the same time an *inner function* may be *returned* from the parent function. Moreover, this returned function may be later activated from another context. What will be with such an activation if a context of some free variable is already "gone"? In the general theory, a concept which helps to solve this issue is called a *(lexical) closure*, which in ECMAScript is directly related with a *scope chain* concept.

Closures

In ECMAScript, functions are the *first-class* objects. This term means that functions may be passed as arguments to other functions (in such case they are called *"funargs"*, short from "functional arguments"). Functions which receive "funargs" are called *higher-order functions* or, closer to mathematics, *operators*. Also functions may be returned from other functions. Functions which return other functions are called *function valued* functions (or functions with *functional value*).

There are two conceptual problems related with "funargs" and "functional values". And these two sub-problems are generalized in one which is called a *"Funarg problem"* (or "A problem of a functional argument"). And exactly to solve the *complete "funarg problem"*, the concept of *closures* was invented. Let's describe in more detail these two sub-problems (we'll see that both of them are solved in ECMAScript using a mentioned on figures `[[Scope]]` property of a function).

First subtype of the "funarg problem" is an *"upward funarg problem"*. It appears when a function is returned "up" (to the outside) from another function and uses already mentioned above *free variables*. To be able access variables of the parent context *even after the parent context ends*, the inner function *at creation moment* saves in it's `[[Scope]]` property parent's *scope chain*. Then when the function is *activated*, the scope chain of its context is formed as combination of the activation object and this `[[Scope]]` property (actually, what we've just seen above on figures):

```
Scope chain = Activation object + [[Scope]]
```

Notice again the main thing — exactly at *creation moment* — a function saves *parent's* scope chain, because exactly this *saved scope chain* will be used for variables lookup then in further calls of the function.

```
function foo() {
  var x = 10;
  return function bar() {
    console.log(x);
  };
}

// "foo" returns also a function
// and this returned function uses
// free variable "x"

var returnedFunction = foo();

// global variable "x"
var x = 20;

// execution of the returned function
```

```
returnedFunction(); // 10, but not 20
```

This style of scope is called the *static (or lexical) scope*. We see that the variable `x` is found in the saved `[[Scope]]` of returned `bar` function. In general theory, there is also a *dynamic scope* when the variable `x` in the example above would be resolved as `20`, but not `10`. However, dynamic scope is not used in ECMAScript.

The second part of the "funarg problem" is a *"downward funarg problem"*. In this case a parent context may exist, but may be an ambiguity with resolving an identifier. The problem is: *from which scope* a value of an identifier should be used — statically saved at a function's creation or dynamically formed at execution (i.e. a scope of a *caller*)? To avoid this ambiguity and to form a closure, a *static scope* is decided to be used:

```
// global "x"
var x = 10;

// global function
function foo() {
  console.log(x);
}

(function (funArg) {

  // local "x"
  var x = 20;

  // there is no ambiguity,
  // because we use global "x",
  // which was statically saved in
  // [[Scope]] of the "foo" function,
  // but not the "x" of the caller's scope,
  // which activates the "funArg"

  funArg(); // 10, but not 20

})(foo); // pass "down" foo as a "funarg"
```

We may conclude that a *static scope* is an *obligatory requirement to have closures* in a language. However, some languages may provided combination of dynamic and static scopes, allowing a programmer to choose — what to closure and what do not. Since in ECMAScript only a static scope is used (i.e. we have solutions for both subtypes of the "funarg problem"), the conclusion is: *ECMAScript has complete support of closures*, which technically are implemented using `[[Scope]]` property of functions. Now we may give a correct definition of a closure:

A *closure* is a combination of a code block (in ECMAScript this is a function) and statically/lexically saved all parent scopes. Thus, via these saved scopes a function may easily refer free variables.

Notice, that since *every* (normal) function saves `[[Scope]]` at creation, theoretically, *all functions* in ECMAScript *are closures*.

Another important thing to note, that several functions may have *the same parent scope* (it's quite a normal situation when e.g. we have two inner/global functions). In this case variables stored in the `[[Scope]]` property are *shared between all functions* having the same parent scope chain. Changes of variables made by one closure are *reflected* on reading these variables in another closure:

```
function baz() {
  var x = 1;
  return {
    foo: function foo() { return ++x; },
```

```
    bar: function bar() { return --x; }
  };
}

var closures = baz();

console.log(
  closures.foo(), // 2
  closures.bar()  // 1
);
```

This code may be illustrated with the following figure:

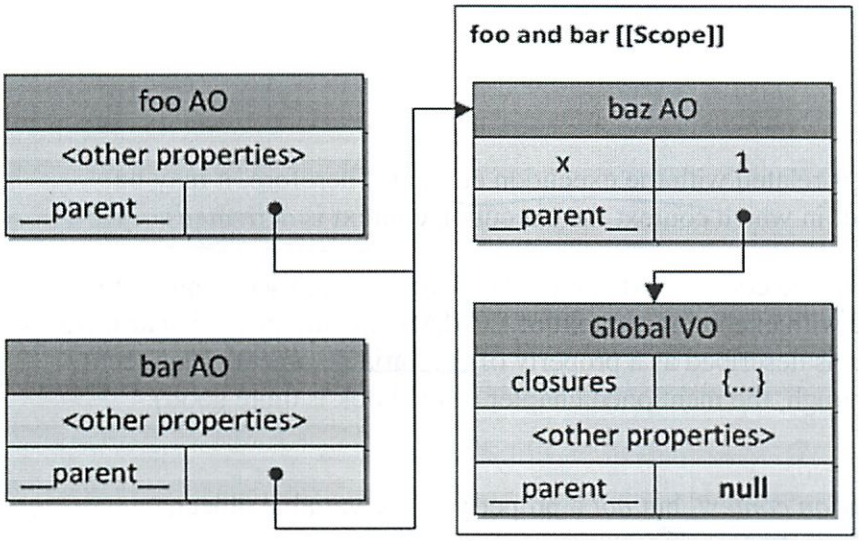


Figure 11. A shared [[Scope]].

Exactly with this feature confusion with creating several functions in a loop is related. Using a loop counter inside created functions, some programmers often get unexpected results when all functions have the *same* value of a counter inside a function. Now it should be clear why it is so — because all these functions have the same `[[Scope]]` where the loop counter has the last assigned value.

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = function () {
    alert(k);
  };
}

data[0](); // 3, but not 0
data[1](); // 3, but not 1
data[2](); // 3, but not 2
```

There are several techniques which may solve this issue. One of the techniques is to provide an additional object in the scope chain — e.g. using additional function:

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = (function (x) {
    return function () {
      alert(x);
    };
  })(k); // pass "k" value
}
```

```
// now it is correct
data[0](); // 0
data[1](); // 1
data[2](); // 2
```

Those who interested deeper in theory of closures and their practical application, may find additional information in the Chapter 6. Closures. And to get more information about a scope chain, take a look on the same name Chapter 4. Scope chain.

And we're moving to the next section, considering the last property of an execution context. This is concept of a `this` value.

This value

A `this` value is a special object which is related with the execution context. Therefore, it may be named as a *context object* (i.e. an object in which context the execution context is *activated*).

Any object may be used as `this` value of the context. I'd like to clarify again the misconception raises sometimes in some descriptions related with execution context of ECMAScript and in particular `this` value. Often, a `this` value, *incorrectly*, is described as a property of the variable object. The recent such a mistake was e.g. in this book (though, the mentioned chapter of the book is quite good). Remember once again:

a `this` value is a *property of the execution context*, but *not* a property of the variable object.

This feature is very important, because in *contrary to variables*, `this` *value never participates in identifier resolution process*. I.e. when accessing `this` in a code, its value is taken *directly* from the execution context and *without any scope chain lookup*. The value of `this` is determinate *only once* when *entering the context*.

By the way, in contrast with ECMAScript, e.g. Python has its `self` argument of methods as a simple variable which is resolved the same and may be even changed during the execution to another value. In ECMAScript it is *not possible* to assign a new value to `this`, because, repeat, it's not a variable and is not placed in the variable object.

In the global context, a `this value` is the *global object itself* (that means, `this` value here equals to *variable object*):

```
var x = 10;

console.log(
  x, // 10
  this.x, // 10
  window.x // 10
);
```

In case of a function context, `this` value in *every single function call* may be *different*. Here `this` value is provided by the *caller* via the *form of a call expression* (i.e. the way of how a function is activated). For example, the function `foo` below is a *callee*, being called from the global context, which is a *caller*. Let's see on the example, how for the same code of a function, `this` value in different calls (different ways of the function activation) is provided *differently* by the caller:

```
// the code of the "foo" function
```

```
// never changes, but the "this" value
// differs in every activation

function foo() {
  alert(this);
}

// caller activates "foo" (callee) and
// provides "this" for the callee

foo(); // global object
foo.prototype.constructor(); // foo.prototype

var bar = {
  baz: foo
};

bar.baz(); // bar

(bar.baz)(); // also bar
(bar.baz = bar.baz)(); // but here is global object
(bar.baz, bar.baz)(); // also global object
(false || bar.baz)(); // also global object

var otherFoo = bar.baz;
otherFoo(); // again global object
```

To consider deeply why (and that is more essential — *how*) `this` value may change in every function call, you may read Chapter 3. This where all mentioned above cases are discussed in detail.

Conclusion

At this step we finish this brief overview. Though, it turned out to not so "brief" 😉 However, the whole explanation of all these topics requires a complete book. We though didn't touch two major topics: *functions* (and the difference between some types of functions, e.g. *function declaration* and *function expression*) and the *evaluation strategy* used in ECMAScript. Both topics may be found in the appropriate chapters of ES3 series: Chapter 5. Functions and Chapter 8. Evaluation strategy.

If you have comments, questions or additions, I'll be glad to discuss them in comments.

Good luck in studying ECMAScript!

Written by: Dmitry A. Soshnikov
Published on: 2010-09-02

Tags: Activation object, ECMA-262-3, ECMAScript, execution context, JavaScript, OOP, Scope chain, this, Variable object, [[Scope]]

This entry was posted on September 02nd, 2010 and is filed under ECMAScript. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response or Trackback from your own site.

« Note 2. ECMAScript. Equality operators.

adequately good(/)

decent programming advice

written by ben cherry(http://twitter.com/bcherry)

2010-02-08

JavaScript Scoping and Hoisting(/2010/2/JavaScript-Scoping-and-Hoisting)

Do you know what value will be alerted if the following is executed as a JavaScript program?

```
var foo = 1;
function bar() {
        if (!foo) {
                var foo = 10;
        }
        alert(foo);
}
bar();
```

If it surprises you that the answer is "10", then this one will probably really throw you for a loop:

```
var a = 1;
function b() {
        a = 10;
        return;
        function a() {}
}
b();
alert(a);
```

Here, of course, the browser will alert "1". So what's going on here? While it might seem strange, dangerous, and confusing, this is actually a powerful and expressive feature of the language. I don't know if there is a standard name for this specific behavior, but I've come to like the term "hoisting". This article will try to shed some light on this mechanism, but first lets take a necessary detour to understand JavaScript's scoping.

Scoping in JavaScript

One of the sources of most confusion for JavaScript beginners is scoping. Actually, it's not just beginners. I've met a lot of experienced JavaScript programmers who don't fully understand scoping. The reason scoping is so confusing in JavaScript is because it looks like a C-family language. Consider the following C program:

```
#include <stdio.h>
int main() {
        int x = 1;
        printf("%d, ", x); // 1
        if (1) {
```

```
        int x = 2;
        printf("%d, ", x); // 2
    }
    printf("%d\n", x); // 1
}
```

The output from this program will be `1, 2, 1`. This is because C, and the rest of the C family, has **block-level scope**. When control enters a block, such as the `if` statement, new variables can be declared within that scope, without affecting the outer scope. This is not the case in JavaScript. Try the following in Firebug:

```
var x = 1;
console.log(x); // 1
if (true) {
    var x = 2;
    console.log(x); // 2
}
console.log(x); // 2
```

✓ this seems to make more sense

In this case, Firebug will show `1, 2, 2`. This is because JavaScript has **function-level scope**. This is radically different from the C family. Blocks, such as `if` statements, **do not** create a new scope. Only functions create a new scope.

To a lot of programmers who are used to languages like C, C++, C#, or Java, this is unexpected and unwelcome. Luckily, because of the flexibility of JavaScript functions, there is a workaround. If you must create temporary scopes within a function, do the following:

```
function foo() {
    var x = 1;
    if (x) {
        (function () {
            var x = 2;
            // some other code
        }());
    }
    // x is still 1.
}
```

what is that called

This method is actually quite flexible, and can be used anywhere you need a temporary scope, not just within block statements. However, I strongly recommend that you take the time to really understand and appreciate JavaScript scoping. It's quite powerful, and one of my favorite features of the language. If you understand scoping, hoisting will make a lot more sense to you.

Declarations, Names, and Hoisting

In JavaScript, a name enters a scope in one of four basic ways:

1. **Language-defined:** All scopes are, by default, given the names `this` and `arguments`.
2. **Formal parameters:** Functions can have named formal parameters, which are scoped to the body of that function.
3. **Function declarations:** These are of the form `function foo() {}`.
4. **Variable declarations:** These take the form `var foo;`.

✓ declare

Function declarations and variable declarations are always moved ("hoisted") invisibly to the top of their containing scope by the JavaScript interpreter. Function parameters and language-defined names are, obviously, already there. This means that code like this:

```
function foo() {
    bar();
    var x = 1;
}
```

is actually interpreted like this:

ok

```
function foo() {
    var x;
    bar();
```

```
        x = 1;
}
```

It turns out that it doesn't matter whether the line that contains the declaration would ever be executed. The following two functions are equivalent:

```
function foo() {
        if (false) {
                var x = 1;
        }
        return;
        var y = 1;
}
function foo() {
        var x, y;
        if (false) {
                x = 1;
        }
        return;
        y = 1;
}
```

Notice that the assignment portion of the declarations were not hoisted. Only the name is hoisted. This is not the case with function declarations, where the entire function body will be hoisted as well. But remember that there are two normal ways to declare functions. Consider the following JavaScript:

```
function test() {
        foo(); // TypeError "foo is not a function"
        bar(); // "this will run!"
        var foo = function () { // function expression assigned to local variable 'foo'
                alert("this won't run!");
        }
        function bar() { // function declaration, given the name 'bar'
                alert("this will run!");
        }
}
test();
```

In this case, only the function declaration has its body hoisted to the top. The name 'foo' is hoisted, but the body is left behind, to be assigned during execution.

That covers the basics of hoisting, which is not as complex or confusing as it seems. Of course, this being JavaScript, there is a little more complexity in certain special cases.

Name Resolution Order

The most important special case to keep in mind is name resolution order. Remember that there are four ways for names to enter a given scope. The order I listed them above is the order they are resolved in. In general, if a name has already been defined, it is never overridden by another property of the same name. This means that a function declaration takes priority over a variable declaration. This does not mean that an assignment to that name will not work, just that the declaration portion will be ignored. There are a few exceptions:

- The built-in name `arguments` behaves oddly. It seems to be declared following the formal parameters, but before function declarations. This means that a formal parameter with the name `arguments` will take precedence over the built-in, even if it is undefined. This is a bad feature. Don't use the name `arguments` as a formal parameter.
- Trying to use the name `this` as an identifier anywhere will cause a SyntaxError. This is a good feature.
- If multiple formal parameters have the same name, the one occurring latest in the list will take precedence, even if it is undefined.

Named Function Expressions

You can give names to functions defined in function expressions, with syntax like a function declaration. This does not make it a function

declaration, and the name is not brought into scope, nor is the body hoisted. Here's some code to illustrate what I mean:

```
foo(); // TypeError "foo is not a function"
bar(); // valid
baz(); // TypeError "baz is not a function"
spam(); // ReferenceError "spam is not defined"

var foo = function () {}; // anonymous function expression ('foo' gets hoisted)
function bar() {}; // function declaration ('bar' and the function body get hoisted)
var baz = function spam() {}; // named function expression (only 'baz' gets hoisted)

foo(); // valid
bar(); // valid
baz(); // valid
spam(); // ReferenceError "spam is not defined"
```

How to Code With This Knowledge

Now that you understand scoping and hoisting, what does that mean for coding in JavaScript? The most important thing is to always declare your variables with a `var` statement. I **strongly** recommend that you have *exactly one* `var` statement per scope, and that it be at the top. If you force yourself to do this, you will never have hoisting-related confusion. However, doing this can make it hard to keep track of which variables have actually been declared in the current scope. I recommend using **JSLint(http://www.jslint.com)** with the `onevar` option to enforce this. If you've done all of this, your code should look something like this:

```
/*jslint onevar: true [...] */
function foo(a, b, c) {
        var x = 1,
                bar,
                baz = "something";
}
```

What the Standard Says

I find that it's often useful to just consult the **ECMAScript Standard (pdf)(http://www.mozilla.org/js/language/E262-3.pdf)** directly to understand how these things work. Here's what it has to say about variable declarations and scope (section 12.2.2):

> *If the variable statement occurs inside a FunctionDeclaration, the variables are defined with function-local scope in that function, as described in section 10.1.3. Otherwise, they are defined with global scope (that is, they are created as members of the global object, as described in section 10.1.3) using property attributes { DontDelete }. Variables are created when the execution scope is entered. A Block does not define a new execution scope. Only Program and FunctionDeclaration produce a new scope. Variables are initialised to undefined when created. A variable with an Initialiser is assigned the value of its AssignmentExpression when the VariableStatement is executed, not when the variable is created.*

I hope this article has shed some light on one of the most common sources of confusion to JavaScript programmers. I have tried to be as thorough as possible, to avoid creating more confusion. If I have made any mistakes or have large omissions, please let me know.

filed under **javascript(/tag/javascript)**

got an error

Try saving it in an HTML file and running it over Firefox. Or run it in IE8, Chrome or Safari consoles. Apparently the Firebug console does not practice function hoisting when it runs in its "global" scope (which is actually not global but a special "Firebug" scope – try running "this == window" in the Firebug console).

Question 3 is based on similar logic to Question 1. This time it is the **foo** function that gets hoisted.

Now Question 4 seems easy. No function hoisting here...

Almost. If there were no hoisting at all, the TypeError would be "bar not defined" and not "bar not a function". There's no function hoisting, however there *is* variable hoisting. Thus **bar** gets declared up front but its value is not defined. Everything else runs to order.

```
1  //**Simulated processing sequence for Question 4**
2  function foo(){
3      //a declaration for each function expression
4      var bar = undefined;
5      var bar = undefined;
6      return bar(); //TypeError: "bar not defined"
7      //neither Function Expression is reached
8  }
9  alert(foo());
```

What else should I watch out for?

Function Declarations are officially prohibited within non-function blocks (such as if) . However all browsers allow them and interpret them in different ways.

For example the following code snippet in Firefox 3.6 throws an error because it interprets the Function Declaration as a Function Statement (see above) so x is not defined. However in IE8, Chrome 5 and Safari 5 the function x is returned (as expected with standard Function Declarations).

```
1  function foo() {
2      if(false) {
3          function x() {};
4      }
5      return x;
6  }
7  alert(foo());
```

I can see how using Function Declarations can cause confusion but are there any benefits?

Well you could argue that Function Declarations are forgiving – if you try to use a function before it is declared, hoisting fixes the order and the function gets called without mishap. But that kind of forgiveness does not encourage tight coding and in the long run is probably more likely to promote surprises than prevent them. After all, programmers arrange their statements in a particular sequence for a reason.

And there are other reasons to favour Function Expressions?

How did you guess?

a) Function Declarations feel like they were intended to mimic Java style method declarations but Java methods are very different animals. In JavaScript functions are living objects with values. Java methods are just metadata storage. Both the following snippets define functions but only the Function Expression suggests that we are creating an object.

```
1 //Function Declaration
2 function add(a,b) {return a + b};
3 //Function Expression
4 var add = function(a,b) {return a + b};
```

b) Function Expressions are more versatile. A Function Declaration can only exist as a "statement" in isolation. All it can do is create an object variable parented by its current scope. In contrast a Function Expression (by definition) is part of a larger construct. If you want to create an anonymous function or assign a function to a prototype or as a property of some other object you need a Function Expression. Whenever you create a new function using a high order application such as curry (http://javascriptweblog.wordpress.com/2010/04/05/curry-cooking-up-tastier-functions/) or compose (http://javascriptweblog.wordpress.com/2010/04/14/compose-functions-as-building-blocks/) you are using a Function Expression. Function Expressions and Functional Programming are inseparable.

```
1 //Function Expression
2 var sayHello = alert.curry("hello!");
```

Do Function Expressions have any drawbacks?

Typically functions created by Function Expressions are unnamed. For instance the following function is anonymous, *today* is just a reference to an unnamed function:

```
1 var today = function() {return new Date()}
```

Does this really matter? Mostly it doesn't, but as Nick Fitzgerald (http://fitzgeraldnick.com/weblog/) has pointed out debugging with anonymous functions can be frustrating. He suggests using Named Function Expressions (NFEs) as a workaround:

```
1 var today = function today() {return new Date()}
```

However as Asen Bozhilov points out (and Kangax documents (http://yura.thinkweb2.com/named-function-expressions/#jscript-bugs)) NFEs do not work correctly in IE < 9

Conclusions?

Since hoisting

Badly placed Function Declarations are misleading and there are few (if any) situations where you can't use a Function Expression assigned to a variable instead. However if you must use Function Declarations, it will minimize confusion if you place them at the top of the scope to which they belong. I would never place a Function Declarations in an **if** statement.

Since they get hoisted

Having said all this you may well find yourself in situations where it makes sense to use a Function

Declaration. That's fine. Slavish adherance to rules is dangerous and often results in tortuous code. Much more important is that you understand the concepts so that you can make your own informed decisions. I hope this article helps in that regard.

Comments are very welcome. Please let me know if you feel anything I've said is incorrect or if you have something to add.

See also ECMA-262 5th Edition (http://es5.github.com/) sections 10.5, 12.2, 13.0, 13.2

July 6, 2010 by *Angus Croll* | Tags: function declarations, function expressions, hoisting | 50 Comments

What's the big deal?

50 thoughts on "Function Declarations vs. Function Expressions"

Pingback: Function Declarations vs. Function Expressions « JavaScript … | Neorack Tutorials

2. *Nick Fitzgerald* says: *July 6, 2010 at 17:12*
   Nice discussion, Angus.

   I think it is important to know what the bugs with NFEs and JScript actually are, because I was originally not aware of them, and they are very subtle but dangerous (for example when doing variable shadowing). Everyone should go read the "JScript Bugs" section of Kangax's linked article. That way you can make an educated choice on whether the risks are worth the payoff in debugging and profiling.

   In the last month or so, I have started to only use NFEs when I am explicitly debugging/profiling, and remove them when I am ready to commit.

   NFEs were just too good to be true…

   Reply

   - *Angus Croll* says: *July 7, 2010 at 09:31*
     Hi Nick, yeah I was unaware too until alerted by Asen/Kangax. I think your strategy of NFE for development only is the only way to go right now

     Reply

   Pingback: RPW 07/07/10: étude de cas performances Mappy, Firefox 4b1, déclaration de fonctions en JS, tester simplement et automatiquement son interface | BrainCracking - Veille technologique sur les applications Web

4. *Bart* says: *December 25, 2010 at 03:46*
   You mention that the following values 8, 3, 3 and [Type Error: bar is not a function] should be alerted, but the value 8 is not alerted, instead it alerts 3.

```
1   // a globally-scoped variable
2   var a=1;
3
4   // global scope
5   function one(){
6       alert(a);
7   }
8
9   // local scope
10  function two(a){
11      alert(a);
12  }
13
14  // local scope again
15  function three(){
16    var a = 3;
17    alert(a);
18  }
19
20  // Intermediate: no such thing as block scope in javascript
21  function four(){
22      if(true){
23          var a=4;
24      }
25
26      alert(a); // alerts '4', not the global value of '1'
27  }
28
29
30  // Intermediate: object properties
31  function Five(){
32      this.a = 5;
33  }
34
35
36  // Advanced: closure
37  var six = function(){
38      var foo = 6;
39
40      return function(){
41          // javascript "closure" means I have access to foo in here,
42          // because it is defined in the function in which I was defined.
43          alert(foo);
44      }
45  }()
46
47
48  // Advanced: prototype-based scope resolution
49  function Seven(){
50    this.a = 7;
51  }
52
```

1

2

3

4

not seperate

5

ah

```
53  // [object].prototype.property loses to [object].property in the scope chain
54  Seven.prototype.a = -1; // won't get reached, because 'a' is set in the constructor
    above.
55  Seven.prototype.b = 8; // Will get reached, even though 'b' is NOT set in the
    constructor.
56
57
58
59  // These will print 1-8
60  one();
61  two(2);
62  three();
63  four();
64  alert(new Five().a);
65  six();
66  alert(new Seven().a);
67  alert(new Seven().b);
```

- 

ECMA-262

Read 11/9

by Dmitry Soshnikov

ECMA-262-3 in detail. Chapter 2. Variable object.

Read this article in: Russian, Chinese (version1, version2, version 3).

1. Introduction
2. Data declaration
3. Variable object in different execution contexts
    1. Variable object in global context
    2. Variable object in function context
4. Phases of processing the context code
    1. Entering the execution context
    2. Code execution
5. About variables
6. Feature of implementations: property __parent__
7. Conclusion
8. Additional literature

Introduction

Always in programs we declare functions and variables which then successfully use building our systems. But how and where the interpreter finds our *data* (functions, variable)? What occurs, when we reference to needed objects?

Many ECMAScript programmers know that variables are closely related with the execution context:

```
var a = 10; // variable of the global context

(function () {
  var b = 20; // local variable of the function context
})();

alert(a); // 10
alert(b); // "b" is not defined
```

the basics

Also, many programmers know that the isolated scope in the current version of specification is created only by execution contexts with "function" code type. I.e., in contrast with C/C++, for example the block of `for` loop in ECMAScript does not create a local context:

```
for (var k in {a: 1, b: 2}) {
  alert(k);
```

```
}

alert(k); // variable "k" still in scope even the loop is finished
```

Let's see in more details what occurs when we declare our data.

Data declaration

If variables are related with the execution context, it should know where its data are stored and how to get them. This mechanism is called a *variable object*.

A *variable object (in abbreviated form — VO)* is a special object related with an execution context and which stores:

- variables (`var`, VariableDeclaration);
- function declarations (FunctionDeclaration, in abbreviated form FD);
- and function formal parameters

declared in the context.

Notice, in ES5 the concept of *variable object* is replaced with *lexical environments* model, which detailed description can be found in appropriate chapter.

Schematically and for examples, it is possible to present variable object as a normal ECMAScript object:

```
VO = {};
```

And as we said, VO is a property of an execution context:

```
activeExecutionContext = {
  VO: {
    // context data (var, FD, function arguments)
  }
};
```

Indirect referencing to variables (via property names of VO) allows only variable object of the *global context* (where the global object is itself the variable object). For other contexts directly to reference the VO is not possible, it is purely mechanism of implementation.

When we declare a variable or a function, there is nothing else as creation of the new property of the VO with the name and value of our variable.

Example:

```
var a = 10;

function test(x) {
  var b = 20;
};

test(30);
```

And corresponding variable objects are:

```
// Variable object of the global context
VO(globalContext) = {
  a: 10,
  test: <reference to function>
};

// Variable object of the "test" function context
VO(test functionContext) = {
  x: 30,
  b: 20
};
```

But at implementation level (and specification) the variable object is an abstract essence. Physically, in concrete execution contexts, VO is named differently and has different initial structure.

Variable object in different execution contexts

Some operations (e.g. *variable instantiation*) and behavior of the variable object are common for all execution context types. From this viewpoint it is convenient to present the variable object as an abstract base thing. Function context can also define additional details related with the variable object.

```
AbstractVO (generic behavior of the variable instantiation process)

  ├──> GlobalContextVO
  │        (VO === this === global)
  │
  └──> FunctionContextVO
           (VO === AO, <arguments> object and <formal parameters> are added)
```

Let's consider it in detail.

Variable object in global context

Here, first it is necessary to give definition of the *Global object*.

Global object is the object which is created before entering any execution context; this object exists in the single copy, its properties are accessible from any place of the program, the life cycle of the global object ends with program end.

At creation the global object is initialized with such properties as `Math`, `String`, `Date`, `parseInt` etc., and also by additional objects among which can be the reference to the global object itself — for example, in BOM, `window` property of the global object refers to global object (however, not in all implementations):

```
global = {
  Math: <...>,
  String: <...>
  ...
  ...
  window: global
};
```

When referencing to properties of global object the prefix is usually omitted, because global object is not accessible directly by name. However, to get access to it is possible via this value in the global context, and also through recursive references to itself, for example `window` in BOM, therefore write simply:

```
String(10); // means global.String(10);

// with prefixes
window.a = 10; // === global.window.a = 10 === global.a = 10;
this.b = 20; // global.b = 20;
```

So, coming back to variable object of the global context — here variable object is the *global object itself*:

```
VO(globalContext) === global;
```

It is necessary to understand accurately this fact since for this reason declaring a variable in the global context, we have ability to reference it indirectly via property of the global object (for example when the variable name is unknown in advance):

```
var a = new String('test');

alert(a); // directly, is found in VO(globalContext): "test"

alert(window['a']); // indirectly via global === VO(globalContext): "test"
alert(a === this.a); // true

var aKey = 'a';
alert(window[aKey]); // indirectly, with dynamic property name: "test"
```

Variable object in function context

Regarding the execution context of functions — there VO is inaccessible directly, and its role plays so-called an *activation object (in abbreviated form — AO)*.

```
VO(functionContext) === AO;
```

An *activation object* is created on entering the context of a function and initialized by property `arguments` which value is the *Arguments object*:

```
AO = {
  arguments: <ArgO>
};
```

Arguments object is a property of the activation object. It contains the following properties:

- *callee* — the reference to the current function;
- *length* — quantity of *real passed* arguments;
- *properties-indexes* (integer, converted to string) which values are the values of function's arguments (from left to right in the list of arguments). Quantity of these properties-indexes == *arguments.length*. Values of properties-indexes of the *arguments* object and present (really passed) formal parameters are *shared*.

Example:

```
function foo(x, y, z) {

  // quantity of defined function arguments (x, y, z)
  alert(foo.length); // 3

  // quantity of really passed arguments (only x, y)
  alert(arguments.length); // 2

  // reference of a function to itself
```

```
  alert(arguments.callee === foo); // true

  // parameters sharing

  alert(x === arguments[0]); // true
  alert(x); // 10

  arguments[0] = 20;
  alert(x); // 20

  x = 30;
  alert(arguments[0]); // 30

  // however, for not passed argument z,
  // related index-property of the arguments
  // object is not shared

  z = 40;
  alert(arguments[2]); // undefined

  arguments[2] = 50;
  alert(z); // 40

}

foo(10, 20);
```

Concerning the last case, in older versions of Google Chrome there was a bug — there parameter `z` and `arguments[2]` were also shared.

In ES5 the concept of *activation object* is also replaced with common and single model of lexical environments.

Phases of processing the context code

Now we have reached the main point of this article. Processing of the execution context code is divided on two basic stages:

1. Entering the execution context;
2. Code execution.

Modifications of the variable object are closely related with these two phases.

Notice, that processing of these two stages are the *general behavior* and independent from the type of the context (i.e. it is fair for *both*: *global* and *function* contexts).

Entering the execution context

On entering the execution context (but *before* the code execution), VO is filled with the following properties (they have already been described at the beginning):

- for each *formal parameter* of a function (if we are in function execution context)

  — a property of the variable object with a name and value of formal parameter is created; for not passed parameters — property of VO with a name of formal parameter and value *undefined* is created;

- for each *function declaration (FunctionDeclaration, FD)*

  — a property of the variable object with a name and value of a function-object is created; if the variable object already contains a property with the same name, *replace* its value and attributes;

- for each *variable declaration (var, VariableDeclaration)*

  — a property of the variable object with a variable name and value *undefined* is created; if the variable name is the same as a name of already declared formal parameter or a function, the variable declaration *does not disturb* the existing property.

Let's see on the example:

```
function test(a, b) {
  var c = 10;
  function d() {}
  var e = function _e() {};
  (function x() {});
}

test(10); // call
```

On *entering* the `test` function context with the passed parameter `10`, AO is the following:

```
AO(test) = {
  a: 10,
  b: undefined,
  c: undefined,
  d: <reference to FunctionDeclaration "d">
  e: undefined
};
```

Notice, that AO does not contain function `x`. This is because `x` is not a function declaration but the *function-expression (FunctionExpression, in abbreviated form FE)* which *does not affect on VO.*

However, function `_e` is also a function-expression, but as we will see below, because of assigning it to the variable `e`, it becomes accessible via the `e` name. The difference of a `FunctionDeclaration` from the `FunctionExpression` is in detail discussed in Chapter 5. Functions.

And after that there comes the second phase of processing of a context code — the *code execution* stage.

Code execution

By this moment, AO/VO is already filled by properties (though, not all of them have the real values passed by us, most of them yet still have initial value `undefined`).

Considering all the same example, AO/VO during the code interpretation is modified as follows:

```
AO['c'] = 10;
AO['e'] = <reference to FunctionExpression "_e">;
```

Once again I notice that function expression `_e` is still in memory *only because it is saved to declared variable* `e`. But the function expression `x` is not in the AO/VO. If we try to call `x` function *before* or *even after* definition, we get an error: `"x" is not defined`. Unsaved to a variable function expression can be called only with its definition *(in place)* or recursively.

One more (classical) example:

```
alert(x); // function

var x = 10;
alert(x); // 10

x = 20;

function x() {}

alert(x); // 20
```

Why in the first alert `x` is a function and moreover is accessible *before* the declaration? Why it's not `10` or `20`? Because, according to the rule — VO is filled with function declarations *on entering the context*. Also, at the same phase, on entering the context, there is a variable declaration `x`, but as we mentioned above, the step of variable declarations semantically goes *after* function and formal parameters declarations and on this phase *do not disturb* the value of the *already declared function or formal parameter with the same name*. Therefore, on entering the context VO is filled as follows:

```
VO = {};

VO['x'] = <reference to FunctionDeclaration "x">

// found var x = 10;
// if function "x" would not be already defined
// then "x" be undefined, but in our case
// variable declaration does not disturb
// the value of the function with the same name

VO['x'] = <the value is not disturbed, still function>
```

And then at code execution phase, VO is modified as follows:

```
VO['x'] = 10;
VO['x'] = 20;
```

what we can see in the second and third alerts.

In the example below we see again that variables are put into the VO on entering the context phase (so, the `else` block is *never executed*, but nevertheless, the variable `b` *exists* in VO):

```
if (true) {
  var a = 1;
} else {
  var b = 2;
}

alert(a); // 1
alert(b); // undefined, but not "b is not defined"
```

About variables

Often various articles and even books on JavaScript claim that: "it is possible to declare global variables using *var* keyword (in the global context) and without using `var` keyword (in any place)". It is not so. Remember:

variables are declared only with using var keyword.

And assignments like:

```
a = 10;
```

just create the new *property* (but *not the variable*) of the global object. "Not the variable" is not in the sense that it cannot be changed, but "not the variable" in concept of variables in ECMAScript (which then also become properties of the global object because of VO(globalContext) === global, we remember, yeah?).

And the difference is the following (let's show on the example):

```
alert(a); // undefined
alert(b); // "b" is not defined

b = 10;
var a = 20;
```

All again depends on VO and phases of its modifications (*entering the context* stage and the *code execution* stage):

Entering the context:

```
VO = {
  a: undefined
};
```

We see that at this phase there is no any `b` since it is not a variable, `b` will appear only at code execution phase (but in our case won't since there is an error).

Let's change the code:

```
alert(a); // undefined, we know why

b = 10;
alert(b); // 10, created at code execution

var a = 20;
alert(a); // 20, modified at code execution
```

There is one more important point concerning variables. Variables, in contrast with simple properties, have attribute `{DontDelete}`, meaning impossibility to remove a variable via the `delete` operator:

```
a = 10;
alert(window.a); // 10

alert(delete a); // true

alert(window.a); // undefined

var b = 20;
alert(window.b); // 20

alert(delete b); // false

alert(window.b); // still 20
```

Note, in ES5 `{DontDelete}` is renamed into the `[[Configurable]]` and can be manually managed via `Object.defineProperty` method.

However there is one execution context on which this rule does not affect. It is the `eval` context: there

{DontDelete} attribute is not set for variables:

```
eval('var a = 10;');
alert(window.a); // 10

alert(delete a); // true

alert(window.a); // undefined
```

For those who test these examples in console of some debug tool, e.g. *Firebug*: notice, that *Firebug* also *uses* `eval` to execute your code from the console. So there *var*s also do not have {DontDelete} and can be deleted.

Feature of implementations: property __parent__

As it was already noted, by the standard, to get direct access to the activation object is impossible. However, in some implementations, namely in SpiderMonkey and Rhino, functions have special property __parent__, which is the reference to the activation object (or the global variable object) in which these functions have been created.

Example (SpiderMonkey, Rhino):

```
var global = this;
var a = 10;

function foo() {}

alert(foo.__parent__); // global

var VO = foo.__parent__;

alert(VO.a); // 10
alert(VO === global); // true
```

In the example above we see that function *foo* is created in the global context and, accordingly, its __parent__ property is set to variable object of the global context, i.e. to the global object.

However, to get the activation object in SpiderMonkey with the same way is not possible: depending on the version, __parent__ for inner function returns either `null` or global object.

In Rhino, access to the activation object is allowed and available via the same way:

Example (Rhino):

```
var global = this;
var x = 10;

(function foo() {

  var y = 20;

  // the activation object of the "foo" context
  var AO = (function () {}).__parent__;

  print(AO.y); // 20

  // __parent__ of the current activation
  // object is already the global object,
  // i.e. the special chain of variable objects is formed,
  // so-called, a scope chain
```

```
  print(AO.__parent__ === global); // true

  print(AO.__parent__.x); // 10

})();
```

Conclusion

In this article we have moved further forward in studying of objects related with execution contexts. I hope the material is useful and has clarified some aspects and ambiguities which, probably, you had before. Further by the plan, the next chapters will be devoted to the Scope chain, Identifier resolution and, as consequence, Closures.

If you have questions, feel free to ask them in comments.

Additional literature

- 10.1.3 – Variable Instantiation;
- 10.1.5 – Global Object;
- 10.1.6 – Activation Object;
- 10.1.8 – Arguments Object.

Translated by: Dmitry A. Soshnikov.
Published on: 2010-03-15

Originally written by: Dmitry A. Soshnikov [ru, read »]
Originally published on: 2009-06-27

Tags: Activation object, ECMA-262-3, ECMAScript, Variable object

This entry was posted on March 15th, 2010 and is filed under ECMAScript. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response or Trackback from your own site.

« ECMA-262-3 in detail. Chapter 1. Execution Contexts.
ECMA-262-3 in detail. Chapter 4. Scope chain. »

43 Comments:

1.  qFox #permalink
15. March 2010 at 23:50

- 

ECMA-262

by Dmitry Soshnikov

ECMA-262-3 in detail. Chapter 3. This.

Read this article in: Russian, Chinese (version1, version2, version 3).

1. Introduction
2. Definitions
3. This value in the global code
4. This value in the function code
    1. Reference type
    2. Function call and non-Reference type
    3. Reference type and null this value
    4. This value in function called as the constructor
    5. Manual setting of this value for a function call
5. Conclusion
6. Additional literature

Introduction

In this article we will discuss one more detail directly related with execution contexts. The topic of discussion is the `this` keyword.

As the practice shows, this topic is difficult enough and often causes issues in determination of `this` value in different execution contexts.

Many programmers are used to thinking that the `this` keyword in programming languages is closely related to the object-oriented programming, exactly referring the newly created object by the constructor. In ECMAScript this concept is also implemented, however, as we will see, here it is not limited only to definition of created object.

Let's see in detail what exactly `this` value is in ECMAScript.

Definitions

`this` value is a property of the execution context:

```
activeExecutionContext = {
  VO: {...},
  this: thisValue
};
```

where VO is variable object which we discussed in the previous chapter.

`this` is directly related to the type of executable code of the context. The value is determined *on entering the context* and is *immutable* while the code is running in the context.

Let's consider these cases more in detail.

This value in the global code

Here everything is simple enough. In the global code, `this` value is *always* the *global object* itself. Thus, it is possible to reference it indirectly:

```
// explicit property definition of
// the global object
this.a = 10; // global.a = 10
alert(a); // 10

// implicit definition via assigning
// to unqualified identifier
b = 20;
alert(this.b); // 20

// also implicit via variable declaration
// because variable object of the global context
// is the global object itself
var c = 30;
alert(this.c); // 30
```

This value in the function code

Things are more interesting when `this` is used in function code. This case is the most difficult and causes many issues.

The first (and, probably, the main) feature of `this` value in this type of code is that here it is *not statically bound* to a function.

As it has been mentioned above, `this` value is determined on entering the context, and in case with a function code the value can be *absolutely different every time*.

However, at runtime of the code `this` value is *immutable*, i.e. it is not possible to assign a new value to it since *this is not a variable* (in contrast, say, with *Python* programming language and its explicitly defined `self` object which can repeatedly be changed at runtime):

```
var foo = {x: 10};

var bar = {
  x: 20,
  test: function () {

    alert(this === bar); // true
    alert(this.x); // 20

    this = foo; // error, can't change this value

    alert(this.x); // if there wasn't an error, then would be 10, not 20

  }
```

```
};

// on entering the context this value is
// determined as "bar" object; why so - will
// be discussed below in detail

bar.test(); // true, 20

foo.test = bar.test;

// however here this value will now refer
// to "foo" - even though we're calling the same function

foo.test(); // false, 10
```

So what affects the variations of `this` value in function code? There are several factors.

First, in a usual function call, `this` is provided *by the caller* which activates the code of the context, i.e. *the parent context which calls the function*. And the value of `this` is determined by the *form of a call expression* (in other words by the form how *syntactically* the function is called).

It is necessary to understand and remember this *important* point in order to be able to determine `this` value in any context without any problems. Exactly the *form of a call expression*, i.e. the way of calling the function, influences `this` value of a called context *and nothing else*.

(as we can see in some articles and even books on JavaScript which claim that *"`this` value depends on how function is defined: if it is global function then `this` value is set to global object, if function is a method of an object `this` value is always set to this object"* — what is *mistaken* description). Moving forward, we see that even normal global functions can be activated with *different forms of a call expression* which influence a different `this` value:

```
function foo() {
  alert(this);
}

foo(); // global

alert(foo === foo.prototype.constructor); // true

// but with another form of the call expression
// of the same function, this value is different

foo.prototype.constructor(); // foo.prototype
```

It is similarly possible to call the function defined as a method of some object, but `this` value will not be set to this object:

```
var foo = {
  bar: function () {
    alert(this);
    alert(this === foo);
  }
};

foo.bar(); // foo, true

var exampleFunc = foo.bar;

alert(exampleFunc === foo.bar); // true

// again with another form of the call expression
// of the same function, we have different this value
```

```
exampleFunc(); // global, false
```

So how does the form of the call expression influences `this` value? In order to fully understand the determination of the `this` value, it's necessary to consider in detail one of the internal types — the `Reference` type.

Reference type

Using pseudo-code the value of `Reference` type can be represented as an object with two properties: *base* (i.e. object to which a property belongs) and a *propertyName* in this base:

```
var valueOfReferenceType = {
  base: <base object>,
  propertyName: <property name>
};
```

Value of `Reference` type can be *only in two cases*:

1. when we deal with an *identifier*;
2. or with a *property accessor*.

Identifiers are handled by the process of *identifiers resolution* which is in detail considered in the Chapter 4. Scope chain. And here we just notice that at return from this algorithm *always* there is a value of `Reference` type (it is important for `this` value).

Identifiers are variable names, function names, names of function arguments and names of unqualified properties of the global object. For example, for values on following identifiers:

```
var foo = 10;
function bar() {}
```

in intermediate results of operations, corresponding values of `Reference` type are the following:

```
var fooReference = {
  base: global,
  propertyName: 'foo'
};

var barReference = {
  base: global,
  propertyName: 'bar'
};
```

For getting the *real value* of an object from a value of `Reference` type there is `GetValue` method which in a pseudo-code can be described as follows:

```
function GetValue(value) {

  if (Type(value) != Reference) {
    return value;
  }

  var base = GetBase(value);

  if (base === null) {
    throw new ReferenceError;
  }

  return base.[[Get]](GetPropertyName(value));
```

```
}
```

where the internal `[[Get]]` method returns *the real value* of object's property, including as well analysis of the inherited properties from a prototype chain:

```
GetValue(fooReference); // 10
GetValue(barReference); // function object "bar"
```

Property accessors are also know; there are two variations: the *dot notation* (when the property name is correct identifier and is in advance known), or the *bracket notation*:

```
foo.bar();
foo['bar']();
```

On return of intermediate calculation we also have the value of `Reference` type:

```
var fooBarReference = {
  base: foo,
  propertyName: 'bar'
};

GetValue(fooBarReference); // function object "bar"
```

So, how a value of `Reference` type is related with `this` value of a function context? — *in the most important sense*. The given moment is the main of this article. The general rule of determination of `this` value in a function context sounds as follows:

The value of `this` in a function context is provided *by the caller and determined by the current form of a call expression* (how the function call is written syntactically).

If on the left hand side from the call parentheses `( ... )`, there is a value of `Reference` type then `this` value is set to the *base object* of this value of `Reference` type.

In all other cases (i.e. with *any other* value type which is distinct from the `Reference` type), `this` value is always set to `null`. But since there is no any sense in `null` for `this` value, it is *implicitly* converted to *global object*.

Let's show on examples:

```
function foo() {
  return this;
}

foo(); // global
```

We see that on the left hand side of call parentheses there is a `Reference` type value (because *foo* is an identifier):

```
var fooReference = {
  base: global,
  propertyName: 'foo'
};
```

Accordingly, `this` value is set to base object of this value of `Reference` type, i.e. to global object.

Similarly with the property accessor:

```
var foo = {
  bar: function () {
    return this;
  }
};

foo.bar(); // foo
```

Again we have the value of type `Reference` which base is `foo` object and which is used as `this` value at `bar` function activation:

```
var fooBarReference = {
  base: foo,
  propertyName: 'bar'
};
```

However, activating *the same function* with *another form of a call expression*, we have already other `this` value:

```
var test = foo.bar;
test(); // global
```

because `test`, being the identifier, produces other value of `Reference` type, which base (the global object) is used as `this` value:

```
var testReference = {
  base: global,
  propertyName: 'test'
};
```

Note, in the strict mode of ES5 `this` value is not coerced to global object, but instead is set to `undefined`.

Now we can precisely tell, why the same function activated with *different forms of a call expression*, has also different `this` values — the answer is in different intermediate values of type `Reference`:

```
function foo() {
  alert(this);
}

foo(); // global, because

var fooReference = {
  base: global,
  propertyName: 'foo'
};

alert(foo === foo.prototype.constructor); // true

// another form of the call expression

foo.prototype.constructor(); // foo.prototype, because

var fooPrototypeConstructorReference = {
  base: foo.prototype,
  propertyName: 'constructor'
};
```

Another (classical) example of dynamic determination of `this` value by the form of a call expression:

```
function foo() {
  alert(this.bar);
}
```

```
var x = {bar: 10};
var y = {bar: 20};

x.test = foo;
y.test = foo;

x.test(); // 10
y.test(); // 20
```

Function call and non-Reference type

So, as we have noted, in case when on the left hand side of call parentheses there is a value *not* of `Reference` type but *any another* type, `this` value is automatically set to `null` and, as consequence, to the *global* object.

Let's consider examples of such expressions:

```
(function () {
  alert(this); // null => global
})();
```

In this case, we have *function* object but not object of `Reference` type (it is not the identifier and not the property accessor), accordingly `this` value finally is set to global object.

More complex examples:

```
var foo = {
  bar: function () {
    alert(this);
  }
};

foo.bar(); // Reference, OK => foo
(foo.bar)(); // Reference, OK => foo

(foo.bar = foo.bar)(); // global?
(false || foo.bar)(); // global?
(foo.bar, foo.bar)(); // global?
```

So, why having a *property accessor* which intermediate result should be a value of `Reference` type, in certain calls we get for `this` value not the base object (i.e. `foo`) but *global*?

The matter is that last three calls, *after applying of certain operations*, have already on the left hand side of call parentheses the value *not of Reference type*.

With the first case all is clear – there unequivocally `Reference` type and, as consequence, `this` value is the base object, i.e. `foo`.

In the second case there is a *grouping operator* which *does not apply*, considered above, method of getting the real value of an object from value of `Reference` type, i.e. `GetValue` (see note of 11.1.6). Accordingly, at return from evaluation of the grouping operator — we still have a value of `Reference` type and that is why `this` value is again set to the base object, i.e. `foo`.

In the third case, *assignment operator*, unlike the grouping operator, *calls* `GetValue` *method* (see step 3 of 11.13.1). As a result at return there is already *function* object (but not a value of `Reference` type) which means that `this` value set to `null` and, as consequence, to *global*.

Similarly with the fourth and fifth cases — *the comma operator* and *logical OR expression* call the `GetValue` method and accordingly we lose value of type `Reference` and get value of type *function*; and again `this` value is set to *global*.

Reference type and null this value

There is a case when call expression determines on the left hand side of call parentheses the value of `Reference` type, however `this` value is set to `null` and, as consequence, to *global*. It is related to the case when the base object of `Reference` type value is the activation object.

We can see this situation on an example with the inner function called from the parent. As we know from the second chapter, local variables, inner functions and formal parameters are stored in the *activation object* of the given function:

```
function foo() {
  function bar() {
    alert(this); // global
  }
  bar(); // the same as AO.bar()
}
```

The activation object always returns as `this` value — `null` (i.e. pseudo-code `AO.bar()` is equivalent to `null.bar()`). Here again we come back to the described above case, and again, `this` value is set to *global object*.

The exception can be with a function call inside the block of the `with` statement in case if *with* object contains a function name property. The `with` statement adds its object in front of scope chain i.e. *before* the activation object. Accordingly, having values of type `Reference` (by the identifier or a property accessor) we have base object not as an activation object but object of a `with` statement. By the way, it relates not only to inner, but also to global functions because the `with` object *shadows* higher object (global or an activation object) of the scope chain:

```
var x = 10;

with ({

  foo: function () {
    alert(this.x);
  },
  x: 20

}) {

  foo(); // 20

}

// because

var  fooReference = {
  base: __withObject,
  propertyName: 'foo'
};
```

The similar situation should be with calling of the function which is the actual parameter of the `catch` clause: in this case the `catch` object is also added in *front* of scope chain i.e. *before* the activation or global object. However, the given behavior was recognized as a bug of ECMA-262-3 and is fixed in the new version of standard — ECMA-262-5. I.e. `this` value in the given activation should be set to

global object, but not to `catch` object:

```
try {
  throw function () {
    alert(this);
  };
} catch (e) {
  e(); // __catchObject - in ES3, global - fixed in ES5
}

// on idea

var eReference = {
  base: __catchObject,
  propertyName: 'e'
};

// but, as this is a bug
// then this value is forced to global
// null => global

var eReference = {
  base: global,
  propertyName: 'e'
};
```

The same situation with a recursive call of the named function expression (more detailed about functions see in Chapter 5. Functions). At the first call of function, base object is the parent activation object (or the global object), at the recursive call — base object should be special object storing the optional name of a function expression. However, in this case `this` value is also always set to *global*:

```
(function foo(bar) {

  alert(this);

  !bar && foo(1); // "should" be special object, but always (correct) global

})(); // global
```

This value in function called as the constructor

There is one more case related with `this` value in a function context — it is a call of function as the constructor:

```
function A() {
  alert(this); // newly created object, below - "a" object
  this.x = 10;
}

var a = new A();
alert(a.x); // 10
```

In this case, the new operator calls the internal [[Construct]] method of the `A` function which, in turn, after object creation, calls the internal [[Call]] method, all the same function `A`, having provided as `this` value newly created object.

Manual setting of this value for a function call

There are two methods defined in the `Function.prototype` (therefore they are accessible to all functions), allowing to specify `this` value of a function call manually. These are `apply` and `call`

methods.

Both of them accept as the first argument `this` value which is used in a called context. A difference between these methods is insignificant: for the `apply` the second argument necessarily should be an array (or, the *array-like object*, for example, `arguments`), in turn, the `call` method can accept any arguments; obligatory arguments for both methods is only the first — `this` value.

Examples:

```
var b = 10;

function a(c) {
  alert(this.b);
  alert(c);
}

a(20); // this === global, this.b == 10, c == 20

a.call({b: 20}, 30); // this === {b: 20}, this.b == 20, c == 30
a.apply({b: 30}, [40]) // this === {b: 30}, this.b == 30, c == 40
```

Conclusion

In this article we have discussed features of the `this` keyword in ECMAScript (and they really are *features*, in contrast, say, with C++ or Java). I hope article helped to understand more accurately how `this` keyword works in ECMAScript. As always, I am glad to answer your questions in comments.

Additional literature

10.1.7 – This;
11.1.1 – The this keyword;
11.2.2 – The new operator;
11.2.3 – Function calls.

Translated by: Dmitry A. Soshnikov with help of Stoyan Stefanov.
Published on: 2010-03-07

Originally written by: Dmitry A. Soshnikov [ru, read »]
With additions and corrections by: Zeroglif

Originally published on: 2009-06-28; **updated on:** 2010-03-07

Tags: ECMA-262-3, ECMAScript, this

This entry was posted on March 07th, 2010 and is filed under ECMAScript. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response or Trackback from your own site.

- 

ECMA-262

by Dmitry Soshnikov

ECMA-262-3 in detail. Chapter 5. Functions.

Read this article in: Russian, Chinese (version 1, version 2).

1. Introduction
2. Types of functions
    1. Function Declaration
    2. Function Expression
        1. Question "about surrounding parentheses"
        2. Implementations extension: Function Statement
        3. Feature of Named Function Expression (NFE)
        4. NFE and SpiderMonkey
        5. NFE and JScript
    3. Functions created via Function constructor
3. Algorithm of function creation
4. Conclusion
5. Additional literature

Introduction

In this article we will talk about one of the general ECMAScript objects — about functions. In particular, we will go through various types of functions, will define how each type influences variables object of a context and what is contained in the scope chain of each function. We will answer the frequently asked questions such as: *"is there any difference (and if there are, what are they?) between functions created as follows*:

```
var foo = function () {
  ...
};
```

from functions defined in a "habitual" way?":

```
function foo() {
  ...
}
```

Or, *"why in the next call, the function has to be surrounded with parentheses?"*:

```
(function () {
  ...
})();
```

Since these articles relay on earlier chapters, for full understanding of this part it is desirable to read Chatper 2. Variable object and Chapter 4. Scope chain, since we will actively use terminology from these chapters.

But let us give one after another. We begin with consideration of function types.

Types of functions

In ECMAScript there are three function types and each of them has its own features.

Function Declaration

A *Function Declaration (abbreviated form is FD)* is a function which:

- has an *obligatory* name;
- in the source code position it is positioned: either at the *Program* level or directly in the body of another function *(FunctionBody)*;
- is created on *entering the context* stage;
- *influences* variable object;
- and is declared in the following way:

```
function exampleFunc() {
  ...
}
```

The main feature of this type of functions is that *only they influence variable object* (they are stored in the VO of the context). This feature defines the second important point (which is a consequence of a variable object nature) — at the *code execution stage* they are already *available* (since FD are stored in the VO on entering the context stage — before the execution begins).

Example (function is called before its declaration in the source code position):

```
foo();

function foo() {
  alert('foo');
}
```

What's also important is the position at which the funcion is defined in the source code (see the second bullet in the *Function declaration* definition above):

```
// function can be declared:
// 1) directly in the global context
function globalFD() {
  // 2) or inside the body
  // of another function
  function innerFD() {}
}
```

These are *the only two positions* in code where a function may be *declared* (i.e. it is impossible to declare it in an *expression position* or inside a code *block*).

There's one alternative to function declarations which is called *function expressions*, which we are

The logical question now is why do we need this type of functions at all? The answer is obvious — to use them in *expressions* and "not pollute" the variables object. This can be demonstrated in passing a function as an argument to another function:

```
function foo(callback) {
  callback();
}

foo(function bar() {
  alert('foo.bar');
});

foo(function baz() {
  alert('foo.baz');
});
```

In case a FE is assigned to a variable, the function remains stored in memory and can later be accessed via this variable name (because variables as we know influence VO):

```
var foo = function () {
  alert('foo');
};

foo();
```

Another example is creation of encapsulated scope to hide auxiliary helper data from external context (in the following example we use FE which is called right after creation):

```
var foo = {};

(function initialize() {

  var x = 10;

  foo.bar = function () {
    alert(x);
  };

})();

foo.bar(); // 10;

alert(x); // "x" is not defined
```

We see that function `foo.bar` (via its `[[Scope]]` property) has access to the internal variable `x` of function `initialize`. And at the same time `x` is not accessible directly from the outside. This strategy is used in many libraries to create "private" data and hide auxiliary entities. Often in this pattern the name of initializing FE is omitted:

```
(function () {

  // initializing scope

})();
```

Here's another examples of FE which are created conditionally at runtime and do not pollute VO:

```
var foo = 10;

var bar = (foo % 2 == 0
  ? function () { alert(0); }
  : function () { alert(1); }
);
```

about to cover.

Function Expression

A *Function Expression (abbreviated form is FE)* is a function which:

- in the source code can only be defined at the *expression position*;
- can have an *optional* name;
- it's definition *has no effect* on variable object;
- and is created at the *code execution* stage.

The main feature of this type of functions is that in the source code they are always in the *expression* position. Here's a simple example such *assignment expression*:

```
var foo = function () {
  ...
};
```

This example shows how an *anonymous* FE is assigned to `foo` variable. After that the function is available via `foo` name — `foo()`.

The definition states that this type of functions can have an *optional* name:

```
var foo = function _foo() {
  ...
};
```

What's important here to note is that from the *outside* FE is accessible via variable `foo` — `foo()`, while from *inside* the function (for example, in the *recursive call*), it is also possible to use `_foo` name.

When a FE is assigned a name it can be difficult to distinguish it from a FD. However, if you know the definition, it is easy to tell them apart: *FE is always in the expression position*. In the following example we can see various ECMAScript expressions in which all the functions are FE:

```
// in parentheses (grouping operator) can be only an expression
(function foo() {});

// in the array initialiser - also only expressions
[function bar() {}];

// comma also operates with expressions
1, function baz() {};
```

The definition also states that FE is created at the code execution stage and is not stored in the variable object. Let's see an example of this behavior:

```
// FE is not available neither before the definition
// (because it is created at code execution phase),

alert(foo); // "foo" is not defined

(function foo() {});

// nor after, because it is not in the VO

alert(foo);  // "foo" is not defined
```

```
bar(); // 0
```

Question "about surrounding parentheses"

Let's go back and answer the question from the beginning of the article — *"why is it necessary to surround a function in parentheses if we want to call it right from it's definition"*. Here's an answer to this question: restrictions of the expression statement.

According to the standard, the *expression statement (ExpressionStatement)* cannot begin with an opening curly brace — { since it would be indistinguishable from the *block*, and also the expression statement cannot begin with a `function` keyword since then it would be indistinguishable from the *function declaration*. I.e., if we try to define an *immediately invoked function* the following way (starting with a `function` keyword):

```
function () {
  ...
}();

// or even with a name

function foo() {
  ...
}();
```

we deal with *function declarations*, and in both cases a parser will produce a parse error. However, the *reasons* of these parse errors vary.

If we put such a definition in the global code (i.e. on the `Program` level), the parser should treat the function as *declaration*, since it starts with a `function` keyword. And in first case we get a `SyntaxError` because of absence of the function's *name* (a function declaration as we said should always have a name).

In the second case we *do* have a name (`foo`) and the function declaration should be created normally. But it doesn't since we have *another* syntax error there — a *grouping operator without an expression* inside it. Notice, in this case it's exactly a *grouping operator* which follows the function declaration, *but not the parentheses of a function call*! So if we had the following source:

```
// "foo" is a function declaration
// and is created on entering the context

alert(foo); // function

function foo(x) {
  alert(x);
}(1); // and this is just a grouping operator, not a call!

foo(10); // and this is already a call, 10
```

everything is fine since here we have two syntactic productions — a *function declaration* and a *grouping operator* with an expression (`1`) inside it. The example above is the same as:

```
// function declaration
function foo(x) {
  alert(x);
}

// a grouping operator
```

```
// with the expression
(1);

// another grouping operator with
// another (function) expression
(function () {});

// also - the expression inside
("foo");

// etc
```

In case we had such a definition inside a *statement*, then as we said, there because of ambiguity we would get a syntax error:

```
if (true) function foo() {alert(1)}
```

The construction above *by the specification* is syntactically *incorrect* (an expression statement cannot begin with a `function` keyword), but as we will see below, none of the implementations provide the syntax error, but handle this case, though, every in it's own manner.

Having all this, how should we tell the parser that what we really want it to *call* a function immediately after its creation? The answer is obvious. It's should be a *function expression*, and *not* a *function declaration*. And the simplest way to create an expression is to use mentioned above grouping operator. Inside it always there is an *expression*. Thus, the parser distinguishes a code as a *function expression (FE)* and there is no ambiguity. Such a function will be created during the *execution* stage, then executed, and then removed (if there are no references to it).

```
(function foo(x) {
  alert(x);
})(1); // OK, it's a call, not a grouping operator, 1
```

In the example above the parentheses at the end (`Arguments` production) are already *call* of the function, and *not* a *grouping operator* as it was in case of a FD.

Notice, in the following example of the immediate invocation of a function, the surrounding parentheses *are not required*, since the function *is already in the expression position* and the parser knows that it deals with a FE which should be created at code execution stage:

```
var foo = {

  bar: function (x) {
    return x % 2 != 0 ? 'yes' : 'no';
  }(1)

};

alert(foo.bar); // 'yes'
```

As we see, `foo.bar` is a *string* but not a function as can seem at first inattentive glance. The function here is used only for initialization of the property — depending on the conditional parameter — it is created and called right after that.

Therefore, the complete answer to the question "about parentheses" is the following:

Grouping parentheses *are needed* when a function *is not at the expression position* and if we want to *call it immediately* right after its creation — in this case we just *manually transform the function to FE*.

In case when a parser knows that it deals with a FE, i.e. the function *is already at the expression position* — the parentheses are *not required*.

Apart from surrounding parentheses it is possible to use *any other* way of transformation of a function to FE type. For example:

```
1, function () {
  alert('anonymous function is called');
}();

// or this one
!function () {
  alert('ECMAScript');
}();

// and any other manual
// transformation

...
```

However, grouping parentheses are just the most widespread and the elegant way to do it.

By the way, the grouping operator can surround the function description as without call parentheses, and also including call parentheses. I.e. both expressions below are correct FE:

```
(function () {})();
(function () {}());
```

Implementations extension: Function Statement

The following example shows a code in which *none of implementations processes accordingly to the specification*:

```
if (true) {

  function foo() {
    alert(0);
  }

} else {

  function foo() {
    alert(1);
  }

}

foo(); // 1 or 0 ? test in different implementations
```

Here it is necessary to say that according to the standard this syntactic construction in general is *incorrect*, because as we remember, a function declaration (FD) *cannot appear inside a code block* (here `if` and `else` contain code blocks). As it has been said, FD can appear only in two places: at the *Program* level or directly inside a body of another function.

The above example is incorrect because the code block can contain *only statements*. And the only place in which function can appear within a block is one of such statements — the *expression statement*. But by definition it *cannot begin* with an *opening curly brace* (since it is indistinguishable from the *code block*) or a `function` keyword (since it is indistinguishable from FD).

However in section of errors processing the standard allows for implementations *extensions of program syntax*. And one of such *extensions* can be seen in case of functions which appear in blocks. All implementations existing today do not throw an exception in this case and process it. But every in its own way.

Presence of `if-else` branches assumes a choice is being made which of the two function will be defined. Since this decision is to be made at runtime, that implies that a *function expression (FE)* should be used. However *the majority of implementations* will simply create both of the *function declarations (FD)* on entering the context stage, but since both of the functions use the same name, only the *last declared function* will get called. In this example the function `foo` shows `1` although the `else` *branch never executes*.

However, SpiderMonkey implementation treats this case in two ways: on the one hand it does not consider such functions as declarations (i.e. the function is created on the condition at the code execution stage), but on the other hand they are not real function expressions since they cannot be called without surrounding parentheses (again the parse error — "indistinguishably from FD") and they *are stored in the variable object*.

My opinion is that SpiderMonkey handles this case correctly, separating the own *middle type of function — (FE + FD)*. Such functions are correctly created due the time and according to conditions, but also unlike FE, and more like FD, are available to be called from the outside. This syntactic extension SpiderMonkey names as *Function Statement (in abbreviated form FS)*; this terminology is mentioned in MDC. JavaScript inventor Brendan Eich also noticed this type of functions provided by SpiderMonkey implementation.

Feature of Named Function Expression (NFE)

In case FE has a name *(named function expression, in abbreviated form NFE)* one important feature arises. As we know from definition (and as we saw in the examples above) function expressions do not influence variable object of a context (this means that it's impossible to call them *by name* before or after their definition). However, FE can call itself by name in the recursive call:

```
(function foo(bar) {

  if (bar) {
    return;
  }

  foo(true); // "foo" name is available

})();

// but from the outside, correctly, is not

foo(); // "foo" is not defined
```

Where is the name "foo" stored? In the activation object of `foo`? No, since nobody has defined any "foo" name inside `foo` function. In the parent variable object of a context which creates `foo`? Also not, remember the definition — FE does not influence the VO — what is exactly we see when calling `foo` from the outside. Where then?

Here's how it works: when the interpreter at the code execution stage meets named FE, before creating FE, it creates *auxiliary special object* and *adds it in front of the current scope chain*. Then it creates FE itself at which stage the function gets the `[[Scope]]` property (as we know from the

Chapter 4. Scope chain) — the scope chain of the context which created the function (i.e. in `[[Scope]]` there is that special object). After that, *the name of FE is added to the special object* as unique property; value of this property is the reference to the FE. And the last action is removing that special object from the parent scope chain. Let's see this algorithm on the pseudo-code:

```
specialObject = {};

Scope = specialObject + Scope;

foo = new FunctionExpression;
foo.[[Scope]] = Scope;
specialObject.foo = foo; // {DontDelete}, {ReadOnly}

delete Scope[0]; // remove specialObject from the front of scope chain
```

Thus, from the outside this function name is *not* available (since it is not present in parent scope), but special object which has been saved in `[[Scope]]` of a function and there this name *is available*.

It is necessary to note however, that some implementations, for example Rhino, save this optional name not in the special object but in the *activation object of the FE*. Implementation from Microsoft — JScript, completely breaking FE rules, keeps this name in the *parent variables object* and the function becomes available outside.

NFE and SpiderMonkey

Let's have a look at how different implementations handle this problem. Some versions of SpiderMonkey have one feature related to special object which can be treated as a bug (although all was implemented according to the standard, so it is more of an editorial defect of the specification). It is related to the mechanism of the identifier resolution: the scope chain analysis is two-dimensional and when resolving an identifier it considers the *prototype chain* of every object in the scope chain as well.

We can see this mechanism in action if we define a property in `Object.prototype` and use a "nonexistent" variable from the code. In the following example when resolving the name `x` the global object is reached without finding `x`. However since in SpiderMonkey the global object inherits from `Object.prototype` the name `x` is resolved *there*:

```
Object.prototype.x = 10;

(function () {
  alert(x); // 10
})();
```

Activation objects do not have prototypes. With the same start conditions, it is possible to see the same behavior in the example with inner function. If we were to define a local variable `x` and declare inner function (FD or *anonymous* FE) and then to reference `x` from the inner function, this variable would be resolved normally in the *parent function context* (i.e. there, where it should be and is), instead of in `Object.prototype`:

```
Object.prototype.x = 10;

function foo() {

  var x = 20;

  // function declaration

  function bar() {
```

```
    alert(x);
  }

  bar(); // 20, from AO(foo)

  // the same with anonymous FE

  (function () {
    alert(x); // 20, also from AO(foo)
  })();

}

foo();
```

Some implementations *set a prototype for activation objects*, which is an exception compared to most of other implementations. So, in the *Blackberry* implementation value `x` from the above example is resolved to `10`. I.e. do not reach activation object of `foo` since value is found in `Object.prototype`:

```
AO(bar FD or anonymous FE) -> no ->
AO(bar FD or anonymous FE).[[Prototype]] -> yes - 10
```

And we can see absolutely the same situation in SpiderMonkey in case of special object of a named FE. This special object (by the standard) is *a normal object — "as if by expression* `new Object()`*"*, and accordingly it should be inherited from `Object.prototype`, what is exactly what can be seen in SpiderMonkey implementation (but only up to version 1.7). Other implementations (including newer versions of SpiderMonkey) do not set a prototype for that special object:

```
function foo() {

  var x = 10;

  (function bar() {

    alert(x); // 20, but not 10, as don't reach AO(foo)

    // "x" is resolved by the chain:
    // AO(bar) - no -> __specialObject(bar) -> no
    // __specialObject(bar).[[Prototype]] - yes: 20

  })();
}

Object.prototype.x = 20;

foo();
```

NFE and JScript

ECMAScript implementation from Microsoft — JScript which is currently built into Internet Explorer (up to JScript 5.8 — IE8) has a number of bugs related with named function expressions (NFE). Every of these bugs completely contradicts ECMA-262-3 standard; some of them may cause serious errors.

First, JScript in this case breaks the main rule of FE that they *should not be stored in the variable object by name of functions*. An optional FE name which should be stored in the *special object* and be accessible *only inside the function itself (and nowhere else)* here is stored directly in the *parent variable object*. Moreover, named FE is treated in JScript as the *function declaration (FD)*, i.e. is created *on entering the context* stage and is available before the definition in the source code:

```
// FE is available in the variable object
// via optional name before the
```

```
// definition like a FD
testNFE();

(function testNFE() {
  alert('testNFE');
});

// and also after the definition
// like FD; optional name is
// in the variable object
testNFE();
```

As we see, complete violation of rules.

Secondly, in case of assigning the named FE to a variable at declaration, JScript creates *two different function objects*. It is difficult to name such behavior as logical (especially considering that outside of NFE its name should not be accessible at all):

```
var foo = function bar() {
  alert('foo');
};

alert(typeof bar); // "function", NFE again in the VO - already mistake

// but, further is more interesting
alert(foo === bar); // false!

foo.x = 10;
alert(bar.x); // undefined

// but both function make
// the same action

foo(); // "foo"
bar(); // "foo"
```

Again we see the full disorder.

However it is necessary to notice that if to describe NFE separately from assigning to variable (for example via the grouping operator), and only after that to assign it to a variable, then check on equality returns `true` just like it would be one object:

```
(function bar() {});

var foo = bar;

alert(foo === bar); // true

foo.x = 10;
alert(bar.x); // 10
```

This moment can be explained. Actually, again *two objects* are created but after that remains, really, only *one*. If again to consider that NFE here is treated as the function declaration (FD) then on entering the context stage *FD bar* is created. After that, already at code execution stage the second object — *function expression (FE) bar* is created and *is not saved anywhere*. Accordingly, as there is no any reference on *FE* `bar` it is removed. Thus there is only one object — *FD* `bar`, the reference on which is assigned to `foo` variable.

Thirdly, regarding the indirect reference to a function via `arguments.callee`, it references that object with which name a function is activated (to be exact — functions since there are two objects):

```
var foo = function bar() {

  alert([
    arguments.callee === foo,
    arguments.callee === bar
  ]);

};

foo(); // [true, false]
bar(); // [false, true]
```

Fourthly, as JScript treats NFE as usual FD, it is not submitted to conditional operators rules, i.e. just like a FD, NFE is created on entering the context and the last definition in a code is used:

```
var foo = function bar() {
  alert(1);
};

if (false) {

  foo = function bar() {
    alert(2);
  };

}
bar(); // 2
foo(); // 1
```

This behavior can also be "logically" explained. On entering the context stage the last met FD with name *bar* is created, i.e. function with `alert(2)`. After that, at code execution stage already new function — *FE* `bar` is created, the reference on which is assigned to `foo` variable. Thus (as further in the code the *if-block* with a condition `false` is unreachable), `foo` activation produces `alert(1)`. The logic is clear, but taking into account IE bugs, I have quoted "logically" word since such implementation is obviously broken and depends on JScript bugs.

And the fifth NFE bug in JScript is related with creation of properties of global object via assigning value to an unqualified identifier (i.e. without `var` keyword). Since NFE is treated here as FD and, accordingly, stored in the variable object, assignment to unqualified identifier (i.e. not to variable but to usual property of global object) in case when the *function name is the same as unqualified identifier*, this property *does not become global*.

```
(function () {

  // without var not a variable in the local
  // context, but a property of global object

  foo = function foo() {};

})();

// however from the outside of
// anonymous function, name foo
// is not available

alert(typeof foo); // undefined
```

Again, the "logic" is clear: the function declaration *foo* gets to the activation object of a local context of anonymous function *on entering the context* stage. And at the moment of code execution stage, the name *foo* already exists in AO, i.e. *is treated as local*. Accordingly, at assignment operation there is simply an *update of already existing in AO property foo*, but not creation of *new property of global*

object as should be according to the logic of ECMA-262-3.

Functions created via Function constructor

This type of function objects is discussed separately from FD and FE since it also has its own features. The main feature is that the `[[Scope]]` property of such functions contains only global object:

```
var x = 10;

function foo() {

  var x = 20;
  var y = 30;

  var bar = new Function('alert(x); alert(y);');

  bar(); // 10, "y" is not defined

}
```

We see that the [[Scope]] of *bar* function does not contain AO of *foo* context — the variable "y" is not accessible and the variable "x" is taken from the global context. By the way, pay attention, the *Function* constructor can be used both with *new* keyword and without it, in this case these variants are equivalent.

The other feature of such functions is related with Equated Grammar Productions and Joined Objects. This mechanism is provided by the specification as suggestion for the optimization (however, implementations have the right not to use such optimization). For example, if we have an array of 100 elements which is filled in a loop with functions, then implementation can use this mechanism of *joined objects*. As a result *only one function object for all elements* of an array can be used:

```
var a = [];

for (var k = 0; k < 100; k++) {
  a[k] = function () {}; // possibly, joined objects are used
}
```

But functions created via *Function* constructor are *never joined*:

```
var a = [];

for (var k = 0; k < 100; k++) {
  a[k] = Function(''); // always 100 different funcitons
}
```

Another example related with joined objects:

```
function foo() {

  function bar(z) {
    return z * z;
  }

  return bar;
}

var x = foo();
var y = foo();
```

Here also implementation has the right to join objects *x* and *y* (and to use one object) because

functions physically (including their internal [[Scope]] property) are not distinguishable. Therefore, the functions created via *Function* constructor always require more memory resources.

Algorithm of function creation

The pseudo-code of function creation algorithm (except steps with joined objects) is described below. This description helps to understand in more detail which function objects exist in ECMAScript. The algorithm is identical for all function types.

```
F = new NativeObject();

// property [[Class]] is "Function"
F.[[Class]] = "Function"

// a prototype of a function object
F.[[Prototype]] = Function.prototype

// reference to function itself
// [[Call]] is activated by call expression F()
// and creates a new execution context
F.[[Call]] = <reference to function>

// built in general constructor of objects
// [[Construct]] is activated via "new" keyword
// and it is the one who allocates memory for new
// objects; then it calls F.[[Call]]
// to initialize created objects passing as
// "this" value newly created object
F.[[Construct]] = internalConstructor

// scope chain of the current context
// i.e. context which creates function F
F.[[Scope]] = activeContext.Scope
// if this functions is created
// via new Function(...), then
F.[[Scope]] = globalContext.Scope

// number of formal parameters
F.length = countParameters

// a prototype of created by F objects
__objectPrototype = new Object();
__objectPrototype.constructor = F // {DontEnum}, is not enumerable in loops
F.prototype = __objectPrototype

return F
```

Pay attention, *F.[[Prototype]]* is a prototype of the *function (constructor)* and *F.prototype* is a prototype *of objects created by this function* (because often there is a mess in terminology, and *F.prototype* in some articles is named as a "prototype of the constructor" that is incorrect).

Conclusion

This article has turned out rather big; however, we will mention functions again when will discuss their work as constructors in one of chapters about objects and prototypes which follow. As always, I am glad to answer your questions in comments.

Additional literature

- 

ECMA-262

by Dmitry Soshnikov

ECMA-262-3 in detail. Chapter 7.1. OOP: The general theory.

Read this article in: Russian.

1. Introduction
2. General provisions, paradigms and ideology
    1. Features of class based and prototype based models
        1. Static class based model
            1. Classes and objects
            2. Hierarchical inheritance
            3. Key concepts of class based model
        2. Prototype based model
            1. Delegation based model
            2. Concatenative model
            3. "Duck" typing
            4. Key concepts of prototype based model
        3. Dynamic class based model
    2. Additional features of various OOP implementations
        1. Polymorphism
        2. Encapsulation
        3. Multiple inheritance
        4. Mixins
        5. Traits
        6. Interfaces
        7. Object composition
        8. AOP features
3. Conclusion
4. Additional literature

Introduction

In this article we consider major aspects of object-oriented programming in ECMAScript. That the article has not turned to "yet another" (as this topic already discussed in many articles), more attention will be given, besides, to theoretical aspects to see these processes from within. In particular, we will consider algorithms of objects creation, see how relationships between objects (including the basic relationship — inheritance) are made, and also give accurate definitions which can be used in

discussions (that I hope will dispel some terminological and ideological doubts and messes arising often in articles on OOP in JavaScript).

General provisions, paradigms and ideology

Before analysis of technical part of OOP in ECMAScript, it is necessary to specify a number of general characteristics, and also to clarify the key concepts of the general theory.

ECMAScript supports multiple programming paradigms, which are: structured, object-oriented, functional, imperative and, in the certain cases, aspect-oriented; but, as article is devoted to OOP, let us give the definition of ECMAScript concerning this essence:

ECMAScript is the *object-oriented* programming language with the *prototype based* implementation.

Prototype based model of OOP has a number of differences from the *static class based* paradigm. Let's take a look at them in detail.

Features of class based and prototype based models

Notice, in the previous sentence an important point has been noted — exactly *static* class based. With a "static" term we understand static objects and classes and, as a rule, strong typing (though, the last is not required).

This case is noticed, because often enough in various articles and on forums, calling JavaScript *"another"*, *"different"* the main reason can be used as an oppose pair: *"class vs. prototype"*, although only this difference in some implementations (e.g. in dynamic class-based Python or Ruby) is not so essential (and accepting some conditions, JavaScript becomes not so "another", though the differences in the certain ideological features, nevertheless, are). But more essential is the opposition of *"statics + classes vs. dynamics + prototypes"*. Exactly a statics and classes (examples: C++, Java) and related mechanisms of properties/methods resolution allow to see an accurate difference from the prototype based implementation.

But let us give one after another. Let's consider the general theory and key concepts of these paradigms.

Static class based model

In the class based model there is a concept of a *class* and an *instance* which belongs to this classification. Instances of a class are also often named as *objects* or *exemplars*.

Classes and objects

The class represents a *formal abstract set* of the generalized characteristics of an instance (the *knowledge* about objects).

The term *set* in this respect is closer to mathematics, however, it is possible to call it as *type* or *classification*.

Example (here and below examples will be given in pseudo-code):

```
C = Class {a, b, c} // class C, with characteristics a, b, c
```

Characteristics of instances are: *properties (object description)* and *methods (object activity)*.

Characteristics themselves also can be treated as objects: i.e. whether a property is writable, configurable, active (getter/setter), etc.

Thus, objects store a *state* (i.e. concrete values of all properties described in a class), and classes define *strict unchangeable structure* (i.e. presence of those or other properties) and *strict unchangeable behavior* (presence of those or other methods) of their instances.

```
C = Class {a, b, c, method1, method2}

c1 = {a: 10, b: 20, c: 30} // object c1 of the class C
c2 = {a: 50, b: 60, c: 70} // object c2 with its own state, of the same class C
```

Hierarchical inheritance

For improvement of *a code reuse*, classes can *extend* other classes, bringing necessary additions. This mechanism is called as *(hierarchical) inheritance*.

```
D = Class extends C = {d, e} // {a, b, c, d, e}
d1 = {a: 10, b: 20, c: 30, d: 40, e: 50}
```

The resolution of methods when calling methods from instances is handled by *strict, invariant and consecutive* examination of the class on presence of this or that method; if the method is not found in the native class, search in the class-ancestor, in the ancestor of the class-ancestor etc. i.e. in the *strict hierarchical chain* proceeds. If on reaching the base link of inheritance chain the method still remains unresolved, the conclusion is: *the object does not have (in its set and its hierarchical chain) similar behavior, and to get desirable result is not possible*.

```
d1.method1() // D.method1 (no) -> C.method1 (yes)
d1.method5() // D.method5 (no) -> C.method5 (no) -> no result
```

In contrast with methods which at inheritance are not copied into a class-descendant but form hierarchy, properties are always copied. We can see this behavior on the example of class `D` which ancestor is the class `C`: properties `a`, `b` and `c` are copied, and the structure of D is: `{a, b, c, d, e}`. However, methods `{method1, method2}` are not copied, but inherited. Therefore, the memory usage in this aspect is directly proportional to the depth of hierarchy. The basic lack here is that even if at deeper levels of a hierarchy some properties are not needed to object, it will have all of them anyway.

Key concepts of class based model

So, we have the following key concepts:

- to create an object first it is necessary to define its class;
- thus, the object will be created in its own classification "image and similarity" (structure and behavior);
- resolution of methods is handled by a strict, direct, unchangeable chain of *inheritance*;
- classes-descendants (and accordingly, objects created from them) contain all properties of an inheritance chain (even if some of these properties are not necessary to the concrete inherited class);
- being created, the class cannot (because of the static model) to change a set of characteristics

(neither properties, nor methods) of their instances;
- instances (again because of strict static model) cannot have neither additional own (unique) behavior, nor the additional properties which are distinct from structure and behavior of the class.

Let's take a look at what the alternative OOP model, based on prototypes, suggests.

Prototype based model

Here the basic concept is *dynamic mutable objects*.

Mutations (full convertibility: not only values, but also all of characteristics) are directly related with dynamics of the language.

Such objects can independently store all their characteristics (properties, methods) and do not need the class.

```
object = {a: 10, b: 20, c: 30, method: fn};
object.a; // 10
object.c; // 30
object.method();
```

Moreover, because of dynamics, they can easily change (add, delete, modify) their characteristics:

```
object.method5 = function () {...}; // add new method
object.d = 40; // add new property "d"
delete object.c; // remove property "c"
object.a = 100; // modify property "a"

// as a result: object: {a: 100, b: 20, d: 40, method: fn, method5: fn};
```

That is, at assignment if some characteristic does not exist in object, it is created and initialized with passed value; if it exists, it is just updated.

The code reuse in this case is achieved not via extending the classes (note, we don't say a word about any classes as sets of unchangeable characteristics; the classes are not present here at all), but by referencing to, so-called, *prototype*.

Prototype is an object, which is used either as an *original copy* for other objects, or as a *helper object to which characteristics other objects can delegate* in case if these objects do not have the necessary characteristic themselves.

Delegation based model

Any object can be used as a prototype of another object and again because of mutations the object can easily change its prototype dynamically at runtime.

Notice, currently we're considering the general theory, not touching concrete implementations; when we will discuss concrete implementations (and in particular, ECMAScript), we will see a number of own features.

Example (pseudo-code):

```
x = {a: 10, b: 20};
```

```
y = {a: 40, c: 50};
y.[[Prototype]] = x; // x is the prototype of y

y.a; // 40, own characteristic
y.c; // 50, also own
y.b; // 20 - is gotten from the prototype: y.b (no) -> y.[[Prototype]].b (yes): 20

delete y.a; // removed own "a"
y.a; // 10 - is gotten from the prototype

z = {a: 100, e: 50}
y.[[Prototype]] = z; // changed the prototype of the y to z
y.a; // 100 - is gotten from the prototype
y.e // 50, also - is gotten from the prototype

z.q = 200 // added new property to the prototype
y.q // changes are available and on y
```

This example shows the important feature and the mechanism related with a prototype when it is used as a helper object to which properties, in case of absence of own similar properties, other objects can delegate.

This mechanism is called a *delegation* and based on it prototypical model is a *delegating prototyping (or a delegation based prototyping)*.

Referencing to the characteristics in this case is called a *sending a message* to an object. I.e., when the object cannot respond to the message itself, it delegates to the prototype (asking it to try to answer the message).

The code reuse in this case is called *delegation based inheritance* or *prototype based inheritance*.

Since any object can be used as a prototype, it means that prototypes can also have their own prototypes. This connected combination of prototypes forms a, so-called, *prototype chain*. The chain also like in static classes is *hierarchical*, however because of mutations it can easily rearrange, changing hierarchy and the structure.

I've never thought of it like this before!

```
x = {a: 10}

y = {b: 20}
y.[[Prototype]] = x

z = {c: 30}
z.[[Prototype]] = y

z.a // 10

// z.a found in prototype chain:
// z.a (no) ->
// z.[[Prototype]].a (no) ->
// z.[[Prototype]].[[Prototype]].a (yes): 10
```

If an object and its prototype chain could not respond to the sent message, the object can activate a corresponding *system signal*, handling which it is possible to continue dispatching and delegation to another chain.

This system signal is available in many implementations, including the dynamic class based systems: it's *#doesNotUnderstand* in SmallTalk; *method_missing* in Ruby; *__getattr__* in Python; *__call* in PHP; *__noSuchMethod__* in one of ECMAScript implementations, etc.

Example (SpiderMonkey ECMAScript implementation):

```
var object = {

  // catch the system signal about
  // impossibility to respond the message
  __noSuchMethod__: function (name, args) {
    alert([name, args]);
    if (name == 'test') {
      return '.test() method is handled';
    }
    return delegate[name].apply(this, args);
  }

};

var delegate = {
  square: function (a) {
    return a * a;
  }
};

alert(object.square(10)); // 100
alert(object.test()); // .test() method is handled
```

That is, in contrast with the static class based implementation, in case of impossibility to respond the message, the conclusion is: *the object at the moment does not have the requested characteristic, however to get the result is still possible if to try to analyze alternative prototype chain, or probably, the object will have such characteristic after a number of mutations.*

Regarding ECMAScript, here exactly this implementation — *delegation based prototyping* is used. However, as we will see, up to the specification and implementations there are also some own features.

Concatenative model

But for to be fair, it is necessary to tell some words about other case from definition (though it is not used in ECMAScript) when prototype represents original object from which other object are copied.

Here for a code reuse is used not a delegation but *exact copy (a clone)* of a prototype at the moment of object's *creation*.

This kind of prototyping is called as *concatenative* prototyping.

Having copied in itself all of prototype's characteristics, an object can further fully change its properties and methods also as a prototype can change its own (and this changes will not affect on already existing objects as it would be with changing prototype's characteristics in delegation based model). As the advantage of such approach can be decreasing of the time for dispatching and delegation and the basic lack is higher memory usage.

"Duck" typing

Coming back to dynamics, weak typing and mutations of objects, in contrast with the static class based model, passage of the test for ability to do some actions here can be related *not with what type (class) the object has*, but whether *it is able to respond the message* (whether it is capable to do that will be required after passing the test).

Can it respond to the message?

Example:

```
// in static class based model
```

```
if (object instanceof SomeClass) {
  // some actions are allowed
}

// in dynamic implementation
// it is not essential what the type of an object
// at the moment, because of mutations, type and
// characteristics can be transformed
// repeatedly, but essential, whether the object
// can respond the "test" message

if (isFunction(object.test)) // ECMAScript

if object.respond_to?(:test) // Ruby

if hasattr(object, 'test'): // Python
```

On jargon, it's called as duck typing. That is, objects can be identified by set of their characteristics which are present at the moment of check, instead of object's position in hierarchy or their belonging to any concrete type.

Key concepts of prototype based model

So, let's check the main features of this approach:

- the basic concept is an *object*;
- objects are fully dynamic and mutable (and in the theory can completely mutate from one type into another);
- objects do not have the strict classes which describe their structure and behavior; objects do not need classes;
- however, not having classes, objects can have prototypes to which they can delegate if cannot answer the message themselves;
- the object prototype can be changed at any moment at runtime;
- in *delegation based* model changing prototype's characteristics will affect on all objects related with this prototype;
- in *concatenative prototype* model prototype is the *original copy* from which other objects are cloned and further become completely independent; changes of prototype's characteristics do not affect on objects cloned from it;
- if it is not possible to respond to a message, it is possible to signal the caller about it which can take additional measures (for example, to change dispatching);
- identification of objects can be made not by their hierarchy and belonging to concrete type, but by a current set of characteristics.

However, there is one more model which we should consider also.

Dynamic class based model

We consider this model to show on the example what has been mentioned in the beginning — the difference between just *"a class vs. a prototype"* is not so essential (*especially if the prototype chain is invariant*; for more accurate distinction, it is necessary to consider also a *statics* in classes). As an example, it is possible to use Python or Ruby (or other similar languages). Both languages are use dynamic class based paradigm. However, in certain aspects, some features of prototype based implementation can be seen.

In the following example we can see that just like in the delegation based prototyping, we can augment a class (prototype), and it affects all objects related with this class, we can also dynamically, at runtime, to change object's class (providing a new object for delegation) etc.

```
# Python

class A(object):

    def __init__(self, a):
        self.a = a

    def square(self):
        return self.a * self.a

a = A(10) # creating an instance
print(a.a) # 10

A.b = 20 # new property of the class
print(a.b) # 20 - available via "delegation" for the "a" instance

a.b = 30 # own property created
print(a.b) # 30

del a.b # removed own property
print(a.b) # 20 - again is taken from the class (prototype)

# just like in prototype based model
# it is possible to change "prototype"
# of an object at runtime

class B(object): # "empty" class B
    pass

b = B() # an instance of the class B

b.__class__ = A # changing class (prototype) dynamically

b.a = 10 # create new property
print(b.square()) # 100 - method of the class A is available

# we can delete explicit references on classes
del A
del B

# but the object still have implicit
# reference and the methods are still available
print(b.square()) # 100

# but, to change the class on some of build-in
# is impossible (in current version) and this is the
# feature of implementation
b.__class__ = dict # error
```

In Ruby the picture is similar: there also fully dynamic classes are used (by the way, in current version of Python, in contrast with Ruby and ECMAScript, it is impossible to augment built-in classes/prototypes), we can completely change characteristics of objects and classes (to add methods/properties in a class, and these changes will affect on already existing objects); however, for example, it is impossible to change the class of an object dynamically.

But, this article is dedicated not to Python and Ruby; therefore we're finishing this comparison, and starting to discuss ECMAScript itself.

But before, we still need to take a look on additional *"syntactic and ideological sugar"*, available in some OOP implementations, because such questions often appear in some articles about JavaScript.

And this section has been considerate only to note incorrectness of statements like *"JavaScript is another, it has prototypes, instead of classes"*. It is necessary to understand that *not all* class based implementations are *completely different* in their implementation. And even if we may talk that *"JavaScript is different"* it is necessary to consider (besides only the concept of a "class") also all other related features.

Additional features of various OOP implementations

In this section we will take a brief look on additional features and kinds of code reuse in various OOP implementations, making a parallel with ECMAScript's OOP implementation. The reason is that in appearing articles on JavaScript, OOP concept is limited to some habitual implementation, regardless that there can be various implementations; the only (and main) requirement is that they should technologically and ideologically be proved. Without having found similarity with some "syntactic sugar" from one (habitual) OOP implementation, JavaScript can hasty be named as a "not pure OOP language" that is the incorrect statement.

Polymorphism

Objects in ECMAScript are polymorphic in several meanings.

For example, one function can be applied to different objects, just like it would be the native characteristic of an object (because this value determinate on entering the execution context):

```
function test() {
  alert([this.a, this.b]);
}

test.call({a: 10, b: 20}); // 10, 20
test.call({a: 100, b: 200}); // 100, 200

var a = 1;
var b = 2;

test(); // 1, 2
```

However, there are exceptions: for example, method *Date.prototype.getTime()*, by the standard, as *this* value always should have a date object, otherwise, the exception is being thrown:

```
alert(Date.prototype.getTime.call(new Date())); // time
alert(Date.prototype.getTime.call(new String(''))); // TypeError
```

Or so-called *parametric polymorphism* when function is defined equally for all data types, but however, accepts polymorphic functional argument (example is the *.sort* method of arrays and its argument — polymorphic sort function). By the way, the example above also can be treated as a kind of parametric polymorphism.

Or in the prototype the method can be defined empty, and all created objects should redefine (implement) this method (i.e. "one interface (signature), and a lot of implementations").

Also polymorphism here can be related with *duck typing* which we mentioned above: i.e. the type of object and its place in hierarchy are not so important, but if it has all necessary characteristics, it can be easily accepted (i.e., again the general interface is important, and implementations can vary).

Encapsulation

With this idea there is often a mess and errors in perception. In this case we discuss one of convenient "sugars" of some OOP implementations — well known modifiers: *private, protected* and *public* which also are called as *access levels (or access modifiers)* to characteristics of objects.

I want to note and remind the main purpose of the encapsulation essence: *encapsulation* is an *increasing of abstraction*, but not a paranoid hiding from "malicious hackers" which, "want to write something directly into fields of your classes".

It is a big (and widespread) *mistake* to use *hiding for the sake of hiding*.

Access levels (private, protected and public), have been provided in several OOP implementations first of all for *convenience of the programmer* (and, really, are convenient enough "sugar") to more abstractly describe and build system.

This can be seen in some implementations (e.g. in already mentioned Python and Ruby). On one hand (in Python), these are *__private* and *_protected* properties (which are specified by naming convention via leading underscores) and are not accessible from the outside. On the other hand, Python simply renames such fields by special rules *(_ClassName__field_name)*, and by this name they are already *accessible from the outside*.

```
class A(object):

    def __init__(self):
      self.public = 10
      self.__private = 20

    def get_private(self):
        return self.__private

# outside:

a = A() # instance of A

print(a.public) # OK, 30
print(a.get_private()) # OK, 20
print(a.__private) # fail, available only within A description

# but Python just renames such properties to
# _ClassName__property_name
# and by this name theses properties are
# available outside

print(a._A__private) # OK, 20
```

Or in Ruby: on one hand, there is an ability to define private and protected characteristics; on the other hand, there are special methods (e.g. *instance_variable_get*, *instance_variable_set*, *send* etc.), which allow to get access to encapsulated data.

```
class A

  def initialize
    @a = 10
  end

  def public_method
    private_method(20)
  end

private

  def private_method(b)
```

```
    return @a + b
  end

end

a = A.new # new instance

a.public_method # OK, 30

a.a # fail, @a - is private instance variable without "a" getter

# fail "private_method" is private and
# available only within A class definition

a.private_method # Error

# But, using special meta methods - we have
# access to that encapsulated data:

a.send(:private_method, 20) # OK, 30
a.instance_variable_get(:@a) # OK, 10
```

The main reason is that a programmer *himself* wants to get access to the encapsulated (please notice, I'm specifically not using term "hidden") data. And if these data will be somehow incorrectly changed or there will be any error — the full responsibility for this is completely on the programmer, but not simply a "typing error" or "someone has casually changed some field". But if such cases become frequent, we may nevertheless note a *bad programming practice and style*, since usually it's the best to "talk" with objects only via public API.

The basic purpose of the encapsulation, repeat, is an *abstraction* from the user of the *auxiliary helper data* and not a "way to secure object from hackers". Much more serious measures, rather than the "private" modifier are used for software security and safety.

Encapsulating *auxiliary* helper (local) objects, we provide possibility for further behavior changes of the public-interface with a *minimum of expenses*, localizing and predicting places of these changes. *And exactly this is the main encapsulation purpose*.

Also the important purpose of a setter method is to abstract difficult calculations. For example, the *element.innerHTML* setter — we simply *abstractly* statement — *"now html of this element is the following"* while in setter function for the *innerHTML* property there will be difficult calculations and checks. In this case the issue mostly relates to the *abstraction*, but *encapsulation* as its *increasing* also takes place.

The concept of encapsulation can be related not only to OOP. For example, it can be a simple function which *encapsulates* various calculations, making its using *abstract* (it is not so important for user to know, how e.g. function Math.round(...) is implemented; he simply calls it). It is an encapsulation and, pay attention, I don't say a word about any "private, protected and public".

ECMAScript, in the current version of specification, does not define private, protected, and private modifiers.

However, on practice it is possible to see something that is named "imitation of encapsulation in JS". Usually for this purpose surrounding context (as a rule, the constructor function itself) is used. Unfortunately, often implementing such "imitation", programmers can produce "getters/setters" for *absolutely non-abstract* entities (I repeat, incorrectly treating encapsulation itself):

```
function A() {
```

```
  var _a; // "private" a

  this.getA = function _getA() {
    return _a;
  };

  this.setA = function _setA(a) {
    _a = a;
  };

}

var a = new A();

a.setA(10);
alert(a._a); // undefined, "private"
alert(a.getA()); // 10
```

Thus, everybody understands that for each created object, the pair of methods "getA/setA" is created and what causes the memory issues directly proportional to quantity of created objects (in contrast with if methods would be defined in a prototype). Although, in first case theoretically could be optimization with joined objects.

Also in various articles about JavaScript for such methods, there's a name *"privileged methods"*. To specify, note: ECMA-262-3 does not define any "privileged methods" concept.

However, it can be normal to create methods in the constructor function, as it in ideology of the language — objects are completely mutable and can have the unique characteristics (in the constructor, by a condition, some object can get an additional method, and the other — not etc.).

Moreover, regarding JavaScript, such *"hidden" "private" vars* are not so hidden (if encapsulation is still misinterpreted as protection against "the malicious hacker" which wants to write value in a certain field directly, instead of using a setter method). In some implementations, it is possible to get access to the necessary scope chain (and accordingly to all variable objects in it), by passing a calling context to the *eval* function (it can be tested in SpiderMonkey up to version 1.7):

```
eval('_a = 100', a.getA); // or a.setA, as "_a" is in [[Scope]] of both methods
a.getA(); // 100
```

Or, in the implementations that allow direct access to the activation object (for example, Rhino); changing value of internal variables is possible via accessing corresponding properties of that object:

```
// Rhino
var foo = (function () {
  var x = 10; // "private"
  return function () {
    print(x);
  };
})();
foo(); // 10
foo.__parent__.x = 20;
foo(); // 20
```

Sometimes, as organizational measures (which also can be treated as a kind of encapsulation), "private" and "protected" data in JavaScript are marked by a leading underscore (but in contrast with the Python, here it is only the naming convention):

```
var _myPrivateData = 'testString';
```

Regarding the effect with surrounding execution context, it's used very often, but for encapsulation of *really* auxiliary (helper) data, which is not directly related to the object, and that's convenient to abstract them from the external API:

```
(function () {

  // initializing context

})();
```

Multiple inheritance

Multiple inheritance is a convenient "sugar" for code reuse improvement (if we can inherit one class why not to inherit ten at once?). However, it has a number of lacks and that's why is not popular in implementations.

ECMAScript does not support multiple inheritance (i.e. only one object can be used as a direct prototype) although its ancestor Self programming language had such ability. But in some implementations, such as SpiderMonkey, using *__noSuchMethod__*, it is possible to manage dispatching and delegation to alternative prototype chains.

Mixins

Mixins are also a convenient way of a code reuse. Mixins have been suggested as an alternative to multiple inheritance. These are independent elements which can be *mixed* with any objects, extending their functionality (thus the object can mix several mixins). ECMA-262-3 specification does not define *"mixin"* concept, however according to mixins definition, and because ECMAScript has dynamic mutable objects, nothing prevents from mixing any object with another, simply augmenting its characteristics.

Classical example:

```
// helper for augmentation
Object.extend = function (destination, source) {
  for (property in source) if (source.hasOwnProperty(property)) {
    destination[property] = source[property];
  }
  return destination;
};

var X = {a: 10, b: 20};
var Y = {c: 30, d: 40};

Object.extend(X, Y); // mix Y into X
alert([X.a, X.b, X.c, X.d]); 10, 20, 30, 40
```

Note, I take these definitions ("mixin", we "mix") in quotes as was mentioned that ECMA-262-3 does not define such concept and moreover it's not a mix but usual extending of object by new characteristics (while, for example, in Ruby where the concept of mixins is official, mixin creates a *reference* to included module (i.e. as a matter of fact — creates additional object ("prototype") for delegation), instead of simply copies all properties of the module into an object).

Traits

Traits are similar to mixins, however have a number of features (basic of which is that traits, by

definition, should not have a state which can make naming conflict as it could be with mixins). Regarding ECMAScript, traits are imitated by the same principle, as mixins; the standard doesn't define "traits" concept.

Interfaces

Interfaces available in some OOP implementations are similar to mixins and traits. However, in contrast with traits and mixins, interfaces force classes to completely implement behavior of signatures of their methods.

Interfaces can be treated as completely abstract classes. However, in contrast with abstract classes, (which can implement part of methods itself and other part — to define as signatures) at single inheritance a class can implement several interfaces; for this reason, interfaces (as well as mixins) can be treated as alternative to multiple inheritance.

Standard ECMA-262-3 defines neither concepts of "interface", nor concepts of "abstract class". However, as imitation, it is possible to use augmentation of objects with objects having "empty" methods (or, throwing an exception in methods, signaling that this method should be implemented).

Object composition

Object composition is also one of techniques for a dynamic code reuse. Object composition differs from inheritance with higher flexibility and implements a delegation to dynamically mutable delegates. And this, in turn, is a basis of delegation based prototyping. Besides dynamically mutable prototype, the object can aggregate (and as a result create a *composition*, an *aggregation*) object for delegation, and further at a sending of a certain message to object, to delegate to this delegate. There can be more than one delegate and because of dynamic nature it is possible to change them at runtime.

As an example already mentioned *__noSuchMethod__* can be, but also let's show how to use delegates explicitly:

Example:

```
var _delegate = {
  foo: function () {
    alert('_delegate.foo');
  }
};

var agregate = {

  delegate: _delegate,

  foo: function () {
    return this.delegate.foo.call(this);
  }

};

agregate.foo(); // delegate.foo

agregate.delegate = {
  foo: function () {
    alert('foo from new delegate');
  }
};
```

```
agregate.foo(); // foo from new delegate
```

This relation of objects is called as *"has-a"*, i.e. *"contains inside"* in contrast with inheritance what *"is-a"* i.e. *"is a descendant"*.

As the lack of explicit composition (along with its flexibility in contrast with inheritance) the increasing of an intermediate code can be.

AOP features

As a feature of aspect-oriented programming, *function decorators* can be. Specification ECMA-262-3 doesn't define concept of "function decorators" explicitly (in contrast with, say, Python where this term is official). However, having functional arguments function can be *decorated* and activated in some *aspect* (by applying so-called *advice*):

Example of the simplest decorator:

```
function checkDecorator(originalFunction) {
  return function () {
    if (fooBar != 'test') {
      alert('wrong parameter');
      return false;
    }
    return originalFunction();
  };
}

function test() {
  alert('test function');
}

var testWithCheck = checkDecorator(test);
var fooBar = false;

test(); // 'test function'
testWithCheck(); // 'wrong parameter'

fooBar = 'test';
test(); // 'test function'
testWithCheck(); // 'test function'
```

Conclusion

At this point we're finishing consideration of the general theory (I hope this material has appeared useful to you), and continue with the ECMA-262-3 in detail. Chapter 7.2. OOP: ECMAScript implementation.

Additional literature

- Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems (by Henry Lieberman);
- Prototype-based programming;
- Class;
- Object-oriented programming;
- Abstraction;

This is the second part of article about object-oriented programming in ECMAScript. In the first part we discussed the general theory and drew parallels with ECMAScript. Before reading of the current part, if it is necessary, I recommend reading the first part as in this article we will actively use the passed terminology. You can find the first part here: ECMA-262-3 in detail. Chapter 7.1. OOP: The general theory.

ECMAScript OOP implementation

Having passed the way of highlights of the general theory, we at last have reached the ECMAScript itself. Now, when we know its OOP approach, let's make once again an accurate definition:

ECMAScript is an *object-oriented* programming language supporting *delegating inheritance based on prototypes*.

We begin the analysis from consideration of data types. And first it is necessary to notice that ECMAScript distinguishes entities on *primitive values* and *objects*. Therefore the phrase *"everything in JavaScript is an object"* sometimes arising in various articles, is not correct (is not full). Primitive values concern to a data of certain *types* which we should discuss in detail.

Data types

Though ECMAScript is a dynamic, weakly typed language with "duck" typing, and automatic type conversion, it nevertheless has certain data types. That is, at one moment, an object belongs to one concrete type.

Standard defines nine types, and only six are directly accessible in an ECMAScript program:

- Undefined
- Null
- Boolean
- String
- Number
- Object

Other three types are accessible only at implementation level (none of ECMAScript objects can have such type) and used by the specification for explaining behavior of some operations, for storing intermediate values and other. These are following types:

- Reference
- List
- Completion

Thus (in short overview), `Reference` type is used for an explanation of such operators as `delete`, `typeof`, `this` and other, and consists of a *base object* and a *property name*. `List` type describes behavior of the arguments list (in the `new` expression and function calls). `Completion` type in turn is used for an explanation of behavior `break`, `continue`, `return` and `throw` statements.

Primitive value types

Coming back to the six types used by ECMAScript programs, first five of them: `Undefined`, `Null`, `Boolean`, `String` and `Number` are types of *primitive values*.

Examples of primitive values:

```
var a = undefined;
var b = null;
var c = true;
var d = 'test';
var e = 10;
```

These values are represented in implementations directly on a low level. They are *not objects*, they do not have neither prototypes, nor constructors.

The `typeof` operator can be unintuitive if not properly understood. And one such example of that is with the value `null`. When `null` is supplied to the `typeof` operator, the result is `"object"` regardless of the fact that the type of `null` is specified as `Null`.

```
alert(typeof null); // "object"
```

And the reason is that the `typeof` operator returns the value taken from standard table which simply says: *"for `null` value string `"object"` should be returned"*.

Specification doesn't clarify this, however Brendan Eich (JavaScript inventor) noticed that `null` in contrast with `undefined`, is used in mostly where *objects appear*, i.e. is an essence closely related to objects (meaning the "empty" reference to an object, probably reserved a place for the future purposes). But, in some drafts, there was provided the document where this "phenomenon" was described as a usual bug. Also, this bug appeared in one of bug-trackers where Brendan Eich also participated; as a result it has been decided to leave `typeof null` as is, i.e. `"object"` though ECMA-262-3 standard defines type of `null` as `Null`.

Object type

In turn, the `Object` type (*do not confuse* with the `Object` *constructor*, we're talking now only about abstract types!) is the only type that represents ECMAScript objects.

Object is an unordered collection of key-value pairs.

The keys of objects are called *properties*. Properties are containers for primitive values and other objects. In case when properties contain functions as their values, they are called *methods*.

Example:

```
var x = { // object "x" with three properties: a, b, c
  a: 10, // primitive value
  b: {z: 100}, // object "b" with property z
  c: function () { // function (method)
    alert('method x.c');
  }
};

alert(x.a); // 10
alert(x.b); // [object Object]
alert(x.b.z); // 100
x.c(); // 'method x.c'
```

Dynamic nature

As we noted in chapter 7.1, objects in ES are fully *dynamic*. It means that we may add, modify or remove properties of objects at any time of program execution.

For example:

```
var foo = {x: 10};

// add new property
foo.y = 20;
console.log(foo); // {x: 10, y: 20}

// change property value to function
foo.x = function () {
  console.log('foo.x');
};

foo.x(); // 'foo.x'

// delete property
delete foo.x;
console.log(foo); // {y: 20}
```

Some properties cannot be modified — *read-only* properties or deleted — *non-configurable* properties. We'll consider these cases shortly in the section of property attributes.

Note, ES5 standardized *static* objects which cannot be extended with new properties and none of the properties can be modified or deleted. These are so-called *frozen* objects, which can be gotten by applying `Object.freeze(o)` method.

```
var foo = {x: 10};

// freeze the object
Object.freeze(foo);
console.log(Object.isFrozen(foo)); // true

// can't modify
foo.x = 100;

// can't extend
foo.y = 200;

// can't delete
delete foo.x;

console.log(foo); // {x: 10}
```

Also it's possible just to prevent extensions using `Object.preventExtensions(o)` method, and to control specific attributes with `Object.defineProperty(o)` method:

```
var foo = {x : 10};

Object.defineProperty(foo, "y", {
  value: 20,
  writable: false, // read-only
  configurable: false // non-configurable
});

// can't modify
foo.y = 200;

// can't delete
```

```
delete foo.y; // false

// prevent extensions
Object.preventExtensions(foo);
console.log(Object.isExtensible(foo)); // false

// can't add new properties
foo.z = 30;

console.log(foo); {x: 10, y: 20}
```

For details see this chapter.

Built-in, native and host objects

It is necessary to notice also that the specification distinguishes *native* objects, *built-in* objects and *host* objects.

Built-in and *native* objects are defined by the ECMAScript specification and the implementation, and a difference between them insignificant. *Native* objects are the all objects provided by ECMAScript implementation (some of them can be *built-in*, some can be created during the program execution, for example user-defined objects).

The *built-in* objects are a subtype of *native* objects which are *built into* the ECMAScript *prior to the beginning* of a program (for example, `parseInt`, `Math` etc.).

All *host objects* are objects provided by the host environment, typically a browser, and may include, for example, `window`, `alert`, etc.

Notice, that host objects may be implemented using ES itself and completely correspond to the specification's semantics. From this viewpoint, they can be named *(unofficially)* as *"native-host"* objects, though it's mostly a theoretical aspect. The specification however does not define any "native-host" concept.

Boolean, String and Number objects

Also for some primitives the specification defines special *wrapper objects*. These are following objects:

- Boolean-object
- String-object
- Number-object

Such objects are created with corresponding built in constructors and contain primitive value as one of internal properties. Object representation can be converted into primitive values and vice-versa.

Examples of the object values corresponding to primitive types:

```
var c = new Boolean(true);
var d = new String('test');
var e = new Number(10);

// converting to primitive
// conversion: ToPrimitive
// applying as a function, without "new" keyword
```

Save as Java

```
c = Boolean(c);
d = String(d);
e = Number(e);

// back to Object
// conversion: ToObject
c = Object(c);
d = Object(d);
e = Object(e);
```

Besides, there are also objects created by special built in constructors: `Function` (function objects constructor) `Array` (arrays constructor), `RegExp` (regular expressions constructor), `Math` (the mathematical module), `Date` (the constructor of dates), etc. Such objects are also values of type `Object` and their distinction from each other is managed by internal properties which we will discuss below.

Literal notations

For three object values: *object*, *array* and *regular expression* there are short notations which called accordingly an *object initialiser*, an *array initialiser* and a *regular expression literal*:

```
// equivalent to new Array(1, 2, 3);
// or array = new Array();
// array[0] = 1;
// array[1] = 2;
// array[2] = 3;
var array = [1, 2, 3];

// equivalent to
// var object = new Object();
// object.a = 1;
// object.b = 2;
// object.c = 3;
var object = {a: 1, b: 2, c: 3};

// equivalent to new RegExp("^\\d+$", "g")
var re = /^\d+$/g;
```

Notice, that in case of reassigning the name bindings — `Object`, `Array` or `RegExp` to some new objects, the semantics of subsequent using of the literal notations may vary in implementations. For example in the current Rhino implementation or in the old SpiderMonkey 1.7 appropriate literal notation will create an object corresponding to the *new* value of constructor name. In other implementations (including current Spider/TraceMonkey) semantics of the literal notations is not being changed even if constructor name is rebound to the new object:

```
var getClass = Object.prototype.toString;

Object = Number;

var foo = new Object;
alert([foo, getClass.call(foo)]); // 0, "[object Number]"

var bar = {};

// in Rhino, SpiderMonkey 1.7 - 0, "[object Number]"
// in other: still "[object Object]", "[object Object]"
alert([bar, getClass.call(bar)]);

// the same with Array name
Array = Number;

foo = new Array;
```

```
alert([foo, getClass.call(foo)]); // 0, "[object Number]"

bar = [];

// in Rhino, SpiderMonkey 1.7 - 0, "[object Number]"
// in other: still "", "[object Object]"
alert([bar, getClass.call(bar)]);

// but for RegExp, semantics of the literal
// isn't being changed in all tested implementations

RegExp = Number;

foo = new RegExp;
alert([foo, getClass.call(foo)]); // 0, "[object Number]"

bar = /(?!)/g;
alert([bar, getClass.call(bar)]); // /(?!)/g, "[object RegExp]"
```

Regular Expression Literal and RegExp Objects

Notice although, that in ES3 the two last cases with regular expressions being equivalent semantically, nevertheless differ. The *regexp literal* exists *only in one instance* and is created on parsing stage, while `RegExp` constructor creates always a *new object*. This can cause some issues with e.g. `lastIndex` property of regexp objects when regexp test is fail:

```
for (var k = 0; k < 4; k++) {
  var re = /ecma/g;
  alert(re.lastIndex); // 0, 4, 0, 4
  alert(re.test("ecmascript")); // true, false, true, false
}

// in contrast with

for (var k = 0; k < 4; k++) {
  var re = new RegExp("ecma", "g");
  alert(re.lastIndex); // 0, 0, 0, 0
  alert(re.test("ecmascript")); // true, true, true, true
}
```

Note, in ES5 this issue has been fixed and regexp literal also always creates a new object.

Associative arrays?

Often in various articles or discussions, JavaScript objects (and usually exactly those which created in declarative form — via the object initialiser — {}) are called *hash-tables* or simply — *hashes* (terms from Ruby or Perl), *associative arrays* (term from PHP), *dictionaries* (term from Python) etc.

Using of this terminology is a habit to concrete technology. Really, they are similar enough, and in respect of *"key-value"* pairs storage completely correspond to the theoretical *"associative array"* or *"hash tables"* data structures. Moreover, a hash table abstract data type may be and *usually is* used at implementation level.

However, although terminology is used to describe a conceptual way of thinking, it is not actually technically correct, regarding ECMAScript. As it has been noted, ECMAScript has only one object type and its "subtypes" in respect of a "key-value" pairs storage *do not differ from each other*. Therefore, there is no separated special term (hash or other) for that. Because any object regardless its internal properties can store these pairs:

```
var a = {x: 10};
a['y'] = 20;
a.z = 30;

var b = new Number(1);
b.x = 10;
b.y = 20;
b['z'] = 30;

var c = new Function('');
c.x = 10;
c.y = 20;
c['z'] = 30;

// etc. - with any object "subtype"
```

Moreover, objects in ECMAScript because of delegation can be nonempty, therefore the term "hash" also can be improper:

```
Object.prototype.x = 10;

var a = {}; // create "empty" "hash"

alert(a["x"]); // 10, but it's not empty
alert(a.toString); // function

a["y"] = 20; // add new pair to "hash"
alert(a["y"]); // 20

Object.prototype.y = 20; // and property into the prototype

delete a["y"]; // remove
alert(a["y"]); // but key and value are still here - 20
```

Notice, that ES5 standardized the ability to create objects *without prototypes* — that is, their prototype is set to `null`. It's achieved with using the `Object.create(null)` method. From this viewpoint such objects are simple hash-tables:

```
var aHashTable = Object.create(null);
console.log(aHashTable.toString); // undefined
```

Also, some properties can have specific getters/setters, so it can also confuse:

```
var a = new String("foo");
a['length'] = 10;
alert(a['length']); // 3
```

However, even if to consider that "hash" could have a "prototype" (as for example, in Ruby or Python — a class to which delegate hash-objects), in ECMAScript this terminology can also be improper because *there is no semantic differentiation* between *kinds of property accessors* (i.e. *dot* and *bracket* notations).

Also in ECMAScript concept of a "property" semantically is not separated into a "key", "array index", "method" or "property". Here all of them are *properties* which obey to the common law of reading/writing algorithm with examination of the prototype chain.

In the following example on Ruby we see this distinction in semantics and consequently there such terminology can differ:

```
a = {}
a.class # Hash
```

```
a.length # 0

# new "key-value" pair
a['length'] = 10;

# but semantics for the dot notation
# remains other and means access
# to the "property/method", but not to the "key"

a.length # 1

# and the bracket notation
# provides access to "keys" of a hash

a['length'] # 10

# we can augment dynamically Hash class
# with new properties/methods and they via
# delegation will be available for already created objects

class Hash
  def z
    100
  end
end

# a new "property" is available

a.z # 100

# but not a "key"

a['z'] # nil
```

ECMA-262-3 standard *does not define* concept of *"hash"* (and similar). However, if theoretical data structure is meant, it is possible to name objects so.

Type conversion

To convert an object into a primitive value the method `valueOf` can be used. As we noted, the call of the constructor (for certain types) as a function, i.e. *without* `new` operator performs conversion of object type to a primitive value. For this conversion exactly implicit call of the `valueOf` method is used:

```
var a = new Number(1);
var primitiveA = Number(a); // implicit "valueOf" call
var alsoPrimitiveA = a.valueOf(); // explicit

alert([
  typeof a, // "object"
  typeof primitiveA, // "number"
  typeof alsoPrimitiveA // "number"
]);
```

This method allows objects to participate in various operations, for example, in addition:

```
var a = new Number(1);
var b = new Number(2);

alert(a + b); // 3

// or even so

var c = {
  x: 10,
```

```
  y: 20,
  valueOf: function () {
    return this.x + this.y;
  }
};

var d = {
  x: 30,
  y: 40,
  // the same .valueOf
  // functionality as "c" object has,
  // borrow it:
  valueOf: c.valueOf
};

alert(c + d); // 100
```

The value of the `valueOf` method by default (if it is not overridden) can vary depending on object type. For some objects it returns the `this` value — for example, `Object.prototype.valueOf()`, for others — any calculated value, as e.g. `Date.prototype.valueOf()`, which returns the time of a date:

```
var a = {};
alert(a.valueOf() === a); // true, "valueOf" returned this value

var d = new Date();
alert(d.valueOf()); // time
alert(d.valueOf() === d.getTime()); // true
```

Also there is one more primitive representation of an object — a string representation. For this `toString` method is responsible, which in some operations is also applied automatically:

```
var a = {
  valueOf: function () {
    return 100;
  },
  toString: function () {
    return '__test';
  }
};

// in this operation
// toString method is
// called automatically
alert(a); // "__test"

// but here - the .valueOf() method
alert(a + 10); // 110

// but if there is no
// valueOf method, it
// will be replaced with the
//toString method
delete a.valueOf;
alert(a + 10); // "_test10"
```

The `toString` method defined on `Object.prototype` has special meaning. It returns the value of the internal `[[Class]]` property, which we'll discuss below.

Along with `ToPrimitive` conversion, there is also `ToObject` conversion which vice-versa converts the value to the *object type*.

One of explicit ways to call `ToObject` is to use built in `Object` constructor as a function (though for some types using of `Object` with the `new` operator is also possible):

```
var n = Object(1); // [object Number]
var s = Object('test'); // [object String]

// also for some types it is
// possible to call Object with new operator
var b = new Object(true); // [object Boolean]

// but applied without arguments,
// new Object creates a simple object
var o = new Object(); // [object Object]

// in case if argument for Object function
// is already object value,
// it simply returns
var a = [];
alert(a === new Object(a)); // true
alert(a === Object(a)); // true
```

Regarding calls of the built in constructors with the `new` and without `new` operator, there is no the general rule and it depends on the constructor. For example `Array` or `Function` constructors produce *the same* results when are called as a constructor (with `new`) and as a simple function (without `new`):

```
var a = Array(1, 2, 3); // [object Array]
var b = new Array(1, 2, 3); // [object Array]
var c = [1, 2, 3]; // [object Array]

var d = Function(''); // [object Function]
var e = new Function(''); // [object Function]
```

There are also explicit and implicit type casting when some operators are applied:

```
var a = 1;
var b = 2;

// implicit
var c = a + b; // 3, number
var d = a + b + '5' // "35", string

// explicit
var e = '10'; // "10", string
var f = +e; // 10, number
var g = parseInt(e, 10); // 10, number

// etc.
```

Property attributes

All properties can have a number of attributes:

- `{ReadOnly}` — attempt to write value to the property is ignored; however, ReadOnly-properties can be changed by host-environment actions, therefore ReadOnly — does not mean "constant value";
- `{DontEnum}` — the property is not enumerable by a `for..in` loop;
- `{DontDelete}` — action of the `delete` operator applied to the property is ignored;
- `{Internal}` — the property is internal, it has no name and is used only on implementation level; such properties are not accessible to the ECMAScript program.

Note, in ES5 `{ReadOnly}`, `{DontEnum}` and `{DontDelete}` are renamed accordingly into the `[[Writable]]`, `[[Enumerable]]` and `[[Configurable]]` and can be manually managed via the `Object.defineProperty` and similar methods.

```
var foo = {};

Object.defineProperty(foo, "x", {
  value: 10,
  writable: true, // aka {ReadOnly} = false
  enumerable: false, // aka {DontEnum} = true
  configurable: true // {DontDelete} = false
});

console.log(foo.x); // 10

// attributes set is called a descriptor
var desc = Object.getOwnPropertyDescriptor(foo, "x");

console.log(desc.enumerable); // false
console.log(desc.writable); // true
// etc.
```

Internal properties and methods

Objects also can have a number of internal properties which are a part of implementation and inaccessible for ECMAScript programs directly (however as we will see below, some implementations allow the access to some such properties). These properties by the convention are enclosed with double square brackets — `[[ ]]`.

We will touch some of them (obligatory for all objects); description of other properties can be found in the specification.

Each object should implement the following internal properties and methods:

- `[[Prototype]]` — the prototype of this object (it will be considered below in detail);
- `[[Class]]` — a string representation of object's *kind* (for example, `Object`, `Array`, `Function`, etc.); it is used to distinguish the objects;
- `[[Get]]` — a method of getting the property's value;
- `[[Put]]` — a method of setting the property's value;
- `[[CanPut]]` — checks whether writing to the property is possible;
- `[[HasProperty]]` — checks whether the object has already this property;
- `[[Delete]]` — removes the property from the object;
- `[[DefaultValue]]` — returns a primitive value corresponding with the object (for getting this value the `valueOf` method is called; for some objects, `TypeError` exception can be thrown).

  To get the `[[Class]]` property from ECMAScript programs is possible indirectly via the `Object.prototype.toString()` method. This method should return the following string: `"[object " + [[Class]] + "]"`. For example:

  ```
  var getClass = Object.prototype.toString;

  getClass.call({}); // [object Object]
  getClass.call([]); // [object Array]
  getClass.call(new Number(1)); // [object Number]
  // etc.
  ```

  This feature is often used to check the kind of an object, however, it is necessary to note that by the specification internal `[[Class]]` property of *hosts-objects* can be *any*, including values of the `[[Class]]` property of the built in objects, that in theory does not make such checks 100% proved. For example, `[[Class]]` property of the `document.childNodes.item(...)` method in IE returns `"String"` (in other implementations, `"Function"` is returned):

```
// in IE - "String", in other - "Function"
alert(getClass.call(document.childNodes.item));
```

Constructor

So, as we mentioned above, objects in ECMAScript are created via, so-called, *constructors*.

Constructor is a function that *creates* and *initializes* the newly created object.

For *creation (memory allocation)* the `[[Construct]]` internal method of a constructor function is responsible. The behavior of this internal method is specified and all constructor functions uses this method to allocate memory for new object.

And *initialization* is managed by calling the function in context of newly created object. For this already internal `[[Call]]` method of the constructor function is responsible.

Note, that from user-code only the initialization phase is accessible. Though, even from initialization we can return *different object* ignoring `this` object which was created at the first stage:

```
function A() {
  // update newly created object
  this.x = 10;
  // but return different object
  return [1, 2, 3];
}

var a = new A();
console.log(a.x, a); undefined, [1, 2, 3]
```

Referencing to algorithm of creation of Function objects discussed in the Chapter 5. Functions, we see that function is a native object which among other properties has this internal `[[Construct]]` and `[[Call]]` properties and also explicit `prototype` property — the reference to a prototype of the future objects (notice, `NativeObject` here and below is my pseudo-code naming convention for "native object" concept from ECMA-262-3, but not the built-in constructor).

```
F = new NativeObject();

F.[[Class]] = "Function"

.... // other properties

F.[[Call]] = <reference to function> // function itself

F.[[Construct]] = internalConstructor // general internal constructor

.... // other properties

// prototype of objects created by the F constructor
__objectPrototype = {};
__objectPrototype.constructor = F // {DontEnum}
F.prototype = __objectPrototype
```

Thus `[[Call]]` besides the `[[Class]]` property (which equals to `"Function"`) is the main in respect of objects distinguishing. Therefore the objects having internal `[[Call]]` property are called as *functions*. The `typeof` operator for such objects returns `"function"` value. However, it mostly relates to *native objects*, in case of *host callable objects*, the `typeof` operator (no less

than `[[Class]]` property) of some implementations can return other value: for example, `window.alert(...)` in IE:

```
// in IE - "Object", "object", in other - "Function", "function"
alert(Object.prototype.toString.call(window.alert));
alert(typeof window.alert); // "Object"
```

The internal `[[Construct]]` method is activated by the `new` operator applied to the constructor function. As we said this method is responsible for memory allocation and creation of the object. If there are no arguments, call parenthesis of constructor function can be omitted:

```
function A(x) { // constructor A
  this.x = x || 10;
}

// without arguments, call
// brackets can be omitted
var a = new A; // or new A();
alert(a.x); // 10

// explicit passing of
// x argument value
var b = new A(20);
alert(b.x); // 20
```

And as also we know, this value inside the constructor (at initialization phase) is set to the *newly created object*.

Let's consider the algorithm of objects creation.

Algorithm of objects creation

The behavior of the internal `[[Construct]]` method can be described as follows:

```
F.[[Construct]](initialParameters):

O = new NativeObject();

// property [[Class]] is set to "Object", i.e. simple object
O.[[Class]] = "Object"

// get the object on which
// at the moment references F.prototype
var __objectPrototype = F.prototype;

// if __objectPrototype is an object, then:
O.[[Prototype]] = __objectPrototype
// else:
O.[[Prototype]] = Object.prototype;
// where O.[[Prototype]] is the prototype of the object

// initialization of the newly created object
// applying the F.[[Call]]; pass:
// as this value - newly created object - O,
// arguments are the same as initialParameters for F
R = F.[[Call]](initialParameters); this === O;
// where R is the returned value of the [[Call]]
// in JS view it looks like:
// R = F.apply(O, initialParameters);

// if R is an object
return R
// else
```

```
return O
```

Note two major features:

First, the *prototype* of the created object is taken from the `prototype` property of a function on the *current* moment (it means that the prototype of two created objects from one constructor can vary since the `prototype` property of a function can also vary).

Secondly, as we have mentioned above, if at object initialization the `[[Call]]` has returned an *object*, exactly it is used as the *result* of the whole `new` expression:

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10 - by delegation, from the prototype

// set .prototype property of the
// function to new object; why explicitly
// to define the .constructor property,
// will be described below
A.prototype = {
  constructor: A,
  y: 100
};

var b = new A();
// object "b" has new prototype
alert(b.x); // undefined
alert(b.y); // 100 - by delegation, from the prototype

// however, prototype of the "a" object
// is still old (why - we will see below)
alert(a.x); // 10 - by delegation, from the prototype

function B() {
  this.x = 10;
  return new Array();
}

// if "B" constructor had not return
// (or was return this), then this-object
// would be used, but in this case - an array
var b = new B();
alert(b.x); // undefined
alert(Object.prototype.toString.call(b)); // [object Array]
```

Let's consider a prototype deeply in detail.

Prototype

Every object has a prototype (exceptions can be with some system objects). Communication with a prototype is organized via the *internal*, implicit and inaccessible directly `[[Prototype]]` property. A prototype can be either an *object*, or the `null` value.

Property `constructor`

In the above example there are two important points. The first relates to `constructor` property of the function's `prototype` property.

As we can see in algorithm of function objects creation, `constructor` property is set to function's `prototype` property at function *creation*. The value of this property is the *circular reference to the function itself*:

```
function A() {}
var a = new A();
alert(a.constructor); // function A() {}, by delegation
alert(a.constructor === A); // true
```

Often in this case there is a misunderstanding — `constructor` property is *incorrectly* treated as own property of the created object. However as we have seen, this property belongs to a *prototype* and is accessible to object via *inheritance*.

Via the *inherited* `constructor` property instances can *indirectly* get the reference to the prototype object:

```
function A() {}
A.prototype.x = new Number(10);

var a = new A();
alert(a.constructor.prototype); // [object Object]

alert(a.x); // 10, via delegation
// the same as a.[[Prototype]].x
alert(a.constructor.prototype.x); // 10

alert(a.constructor.prototype.x === a.x); // true
```

Notice though, that both `constructor` and `prototype` properties of the function can be *redefined* after the object is created. In this case the object looses the reference via the mechanism above.

If we *add* new or *modifiy* existing property in the original prototype via the function's `prototype` property, instances will see the newly added properties.

However, if we *change* function's `prototype` property *completely* (via *assigning* a new object), the reference to the *original* constructor (as well as to the original prototype) *is lost*. This is because we create the new object which *does not have* `constructor` property:

```
function A() {}
A.prototype = {
  x: 10
};

var a = new A();
alert(a.x); // 10
alert(a.constructor === A); // false!
```

Therefore, this reference should be restored manually:

```
function A() {}
A.prototype = {
  constructor: A,
  x: 10
};

var a = new A();
alert(a.x); // 10
alert(a.constructor === A); // true
```

Notice though that the *restored manually* `constructor` property, in contrast with the *lost original*, *has no* attribute `{DontEnum}` and, as consequence, is enumerable in the `for..in` loop over the `A.prototype`.

In ES5 was introduced the ability of controlling enumerable state of properties via the `[[Enumerable]]` attribute.

```
var foo = {x: 10};

Object.defineProperty(foo, "y", {
  value: 20,
  enumerable: false // aka {DontEnum} = true
});

console.log(foo.x, foo.y); // 10, 20

for (var k in foo) {
  console.log(k); // only "x"
}

var xDesc = Object.getOwnPropertyDescriptor(foo, "x");
var yDesc = Object.getOwnPropertyDescriptor(foo, "y");

console.log(
  xDesc.enumerable, // true
  yDesc.enumerable  // false
);
```

Explicit `prototype` and implicit `[[Prototype]]` properties

Often prototype of an object is incorrectly confused with explicit reference to the prototype via the function's `prototype` property. Yes, really, it references to the *same object*, as object's `[[Prototype]]` property:

```
a.[[Prototype]] ----> Prototype <---- A.prototype
```

Moreover, `[[Prototype]]` of an instance gets its value from exactly the `prototype` property of the constructor — at object's creation.

However, replacing `prototype` property of the constructor *does not affect* the prototype of *already created objects*. It's *only* the `prototype` property of the constructor that is changed! It means that *new objects* will have a *new prototype*. But *already created* objects (before the `prototype` property was changed), have reference to the *old prototype* and this reference *cannot be changed already*:

```
// was before changing of A.prototype
a.[[Prototype]] ----> Prototype <---- A.prototype

// became after
A.prototype ----> New prototype // new objects will have this prototype
a.[[Prototype]] ----> Prototype // reference to old prototype
```

Example:

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10
```

```
A.prototype = {
  constructor: A,
  x: 20
  y: 30
};

// object "a" delegates to
// the old prototype via
// implicit [[Prototype]] reference
alert(a.x); // 10
alert(a.y) // undefined

var b = new A();

// but new objects at creation
// get reference to new prototype
alert(b.x); // 20
alert(b.y) // 30
```

Therefore, sometimes arising statements in articles on JavaScript claiming that *"dynamic changing of the prototype will affect all objects and they will have that new prototype"* is *incorrect*. New prototype will have *only new* objects which will be created after this changing.

The main rule here is: the object's prototype is set at the moment of object's *creation* and after that *cannot be changed* to new object. Using the explicit `prototype` reference from the constructor if it still refers to the *same* object, it is possible *only* to *add new* or *modify* existing properties of the object's prototype.

Non-standard `__proto__` property

However, some implementations, for example, SpiderMonkey, provide *explicit* reference to object's prototype via the non-standard `__proto__` property:

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10

var __newPrototype = {
  constructor: A,
  x: 20,
  y: 30
};

// reference to new object
A.prototype = __newPrototype;

var b = new A();
alert(b.x); // 20
alert(b.y); // 30

// "a" object still delegates
// to the old prototype
alert(a.x); // 10
alert(a.y); // undefined

// change prototype explicitly
a.__proto__ = __newPrototype;

// now "a" object references
// to new object also
alert(a.x); // 20
alert(a.y); // 30
```

Note, ES5 introduced `Object.getPrototypeOf(O)` method, which directly returns the `[[Prototype]]` property of an object — the original prototype of the instance. However, in contrast with `__proto__`, being only a *getter*, it does not allow to set the prototype.

```
var foo = {};
Object.getPrototypeOf(foo) == Object.prototype; // true
```

Object is independent from its constructor

Since the prototype of an instance is independent from the constructor and the `prototype` property of the constructor, the constructor after its main purpose — creation of the object — can be *removed*. The prototype object will continue to exist, being referenced via the `[[Prototype]]` property:

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10

// set "A" to null - explicit
// reference on constructor
A = null;

// but, still possible to create
// objects via indirect reference
// from other object if
// .constructor property has not been changed
var b = new a.constructor();
alert(b.x); // 10

// remove both implicit references
delete a.constructor.prototype.constructor;
delete b.constructor.prototype.constructor;

// it is not possible to create objects
// of "A" constructor anymore, but still
// there are two such objects which
// still have reference to their prototype
alert(a.x); // 10
alert(b.x); // 10
```

Feature of `instanceof` operator

With the explicit reference to a prototype — via the `prototype` property of the constructor, the work of the `instanceof` operator is related.

This operator works *exactly* with the *prototype chain* of an object but not with the constructor itself. Take this into account, since there is often misunderstanding at this place. That is, when there is a check:

```
if (foo instanceof Foo) {
  ...
}
```

it *does not mean* the check whether the object `foo` is *created* by the `Foo` constructor!

All the `instanceof` operator does is only takes the value of the `Foo.prototype` property and checks its *presence* in the *prototype chain* of `foo`, starting from the `foo.[[Prototype]]`. The

`instanceof` operator is activated by the internal [[HasInstance]] method of the constructor.

Let's see it on the example:

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10

alert(a instanceof A); // true

// if set A.prototype
// to null...
A.prototype = null;

// ...then "a" object still
// has access to its
// prototype - via a.[[Prototype]]
alert(a.x); // 10

// however, instanceof operator
// can't work anymore, because
// starts its examination from the
//prototype property of the constructor
alert(a instanceof A); // error, A.prototype is not an object
```

On the other hand, it is possible to create object by one constructor, but `instanceof` will return `true` on check with *another* constructor. All that is necessary is to set object's `[[Prototype]]` property and `prototype` property of the constructor to the same object:

```
function B() {}
var b = new B();

alert(b instanceof B); // true

function C() {}

var __proto = {
  constructor: C
};

C.prototype = __proto;
b.__proto__ = __proto;

alert(b instanceof C); // true
alert(b instanceof B); // false
```

Prototype as a storage for methods and shared properties

The most useful application of the prototype in ECMAScript is the storage of *methods*, *default state* and *shared properties* of objects.

Indeed, objects can have their own *states*, but methods are usually the same. Therefore, methods, for optimization of a memory usage, are usually defined in the prototype. It means that all instances created by this constructor, always *share* the *same* method.

```
function A(x) {
  this.x = x || 100;
}

A.prototype = (function () {
```

```
  // initializing context,
  // use additional object

  var _someSharedVar = 500;

  function _someHelper() {
    alert('internal helper: ' + _someSharedVar);
  }

  function method1() {
    alert('method1: ' + this.x);
  }

  function method2() {
    alert('method2: ' + this.x);
    _someHelper();
  }

  // the prototype itself
  return {
    constructor: A,
    method1: method1,
    method2: method2
  };

})();

var a = new A(10);
var b = new A(20);

a.method1(); // method1: 10
a.method2(); // method2: 10, internal helper: 500

b.method1(); // method1: 20
b.method2(); // method2: 20, internal helper: 500

// both objects are use
// the same methods from
// the same prototype
alert(a.method1 === b.method1); // true
alert(a.method2 === b.method2); // true
```

Reading and writing properties

As we mentioned, reading and writing of properties are managed by the internal methods `[[Get]]` and `[[Put]]`. The methods are activated by *property accessors* — dot notation or brackets notation:

```
// write
foo.bar = 10; // [[Put]] is called

console.log(foo.bar); // 10, [[Get]] is called
console.log(foo['bar']); // the same
```

Let's show the work of these methods as a pseudo-code.

`[[Get]]` method

The `[[Get]]` method considers the properties from the *prototype chain* of object as well. Therefore properties of a prototype are accessible to object as own.

```
O.[[Get]](P):

// if there is own
```

```
// property, return it
if (O.hasOwnProperty(P)) {
  return O.P;
}

// else, analyzing prototype
var __proto = O.[[Prototype]];

// if there is no prototype (it is,
// possible e.g. in the last link of the
// chain - Object.prototype.[[Prototype]],
// which is equal to null),
// then return undefined;
if (__proto === null) {
  return undefined;
}

// else, call [[Get]] method recursively -
// now for prototype; i.e. go through prototype
// chain: try to find property in the
// prototype, after that - in a prototype of
// the prototype and so on, until
// [[Prorotype]] will be equal to null
return __proto.[[Get]](P)
```

Note, since the `[[Get]]` method in one of cases can return `undefined`, checks on variable presence, like the following are possible:

```
if (window.someObject) {
  ...
}
```

Here, property `someObject` is not found in `window`, then in its prototype, in the prototype of the prototype etc., and in this case, by the algorithm, `undefined` value is returned.

Notice, that for *exactly presence* the `in` operator is responsible. It also considers the prototype chain:

```
if ('someObject' in window) {
  ...
}
```

It helps to avoid cases when, for example, `someObject` can be equal to `false` and the first check does not pass even if `someObject` exists.

`[[Put]]` method

The `[[Put]]` method in contrast creates or updates an *own* property of the object and *shadows* the property with the same name from the prototype.

```
O.[[Put]](P, V):

// if we can't write to
// this property then exit
if (!O.[[CanPut]](P)) {
  return;
}

// if object doesn't have such own,
// property, then create it; all attributes
// are empty (set to false)
if (!O.hasOwnProperty(P)) {
  createNewProperty(O, P, attributes: {
```

```
    ReadOnly: false,
    DontEnum: false,
    DontDelete: false,
    Internal: false
  });
}

// set the value;
// if property existed, its
// attributes are not changed
O.P = V

return;
```

For example:

```
Object.prototype.x = 100;

var foo = {};
console.log(foo.x); // 100, inherited

foo.x = 10; // [[Put]]
console.log(foo.x); // 10, own

delete foo.x;
console.log(foo.x); // again 100, inherited
```

Notice, it's *not* possible to *shadow inherited read-only* property. Result of an assignment is just ignored. This is controlled by the `[[CanPut]]` internal method; see 8.6.2.3 of ES3.

```
// For example, property "length" of
// string objects is read-only; let's make a
// string as a prototype of our object and try
// to shadow the "length" property

function SuperString() {
  /* nothing */
}

SuperString.prototype = new String("abc");

var foo = new SuperString();

console.log(foo.length); // 3, the length of "abc"

// try to shadow
foo.length = 5;
console.log(foo.length); // still 3
```

In strict mode of ES5 an attempt to shadow a non-writable property results a `TypeError`.

Property accessors

That's said, internal methods `[[Get]]` and `[[Put]]` are activated by *property accessors* which in ECMAScript are available via the *dot notation*, or via the *bracket notation*. The dot notation is used when the property name is a valid identifier name and in advance known, bracket notation allows forming names of properties dynamically.

```
var a = {testProperty: 10};

alert(a.testProperty); // 10, dot notation
alert(a['testProperty']); // 10, bracket notation

var propertyName = 'Property';
```

```
alert(a['test' + propertyName]); // 10, bracket notation with dynamic property
```

There is one important feature — property accessor always calls `ToObject` conversion for the object standing on left hand side from the property accessor. And because of this implicit conversion it is possible *roughly speaking* to say that *"everything in JavaScript is an object"* (however as we already know — of course not everything since there are also primitive things).

If we use property accessor with a *primitive value*, we just create *intermediate wrapper object* with corresponding value. After the work is finished, this wrapper is *removed.*

Example:

```
var a = 10; // primitive value

// but, it has access to methods,
// just like it would be an object
alert(a.toString()); // "10"

// moreover, we can even
// (try) to create a new
// property in the "a" primitive calling [[Put]]
a.test = 100; // seems, it even works

// but, [[Get]] doesn't return
// value for this property, it returns
// by algorithm - undefined
alert(a.test); // undefined
```

So, why in this example "primitive" value `a` has access to the `toString` method, but has no to the newly created `test` property?

The answer is simple:

First, as we said, after the property accessor is applied, it is already *not a primitive*, but the *intermediate object*. In this case *new Number(a)* is used, which via delegation finds the `toString` method in the prototype chain:

```
// Algorithm of evaluating a.toString():

1. wrapper = new Number(a);
2. wrapper.toString(); // "10"
3. delete wrapper;
```

Next, `[[Put]]` method also creates *its own wrapper object* when evaluating the `test` property:

```
// Algorithm of evaluating a.test = 100:

1. wrapper = new Number(a);
2. wrapper.test = 100;
3. delete wrapper;
```

We see that in step 3 the wrapper is *removed* and its *newly created* `test` property *is of course also* — with removing the object itself.

Then again `[[Get]]` is called where the property accessor creates *again new wrapper* which of course *does not known* anything about any `test` property:

```
// Algorithm of evaluating a.test:

1. wrapper = new Number(a);
```

```
2. wrapper.test; // undefined
```

That is the reference to properties/methods from a *primitive* value makes sense only for *reading* the properties. Also if any of primitive values often uses the access to properties, for economy of time resources, there is a sense directly to replace it with an object representation. And on the contrary — if values participate only in some small calculations which are not demanding the access to properties then more efficiently primitive values can be used.

Inheritance

As we know, ECMAScript uses *delegating inheritance based on prototypes*.

Chaining, prototypes generate already mentioned *prototype chain*.

Actually, all work for implementing delegation and the analysis of a prototype chain is reduced to the work of the mentioned above `[[Get]]` method.

If you completely understand the simple algorithm of the `[[Get]]` method, the question on inheritance in JavaScript will disappear by itself and the answer to it will become clear.

Often on forums when the talk comes about inheritance in JavaScript, I show as an example only one line of ECMAScript code which very exactly and accurate describes object structure of the language and shows delegation based inheritance. Indeed, we can do not create any constructors or objects but the whole language *is already full of inheritance*. The line of code is very simple:

```
alert(1..toString()); // "1"
```

Now, when we know the algorithm of the `[[Get]]` method and property accessors, we can see what happens here:

1. First, from a primitive value `1` the *wrapper object* as `new Number(1)` is created;

2. Then the *inherited* method `toString` is called from this *wrapper*.

Why the inherited? Because objects in ECMAScript can have *own* properties, and the created wrapper object, in this case, *has no* own `toString` method. Therefore, it *inherits* it from a prototype, i.e. `Number.prototype`.

Notice the subtle case of the syntax. Two dots in the example above *is not an error*. The first dot is used for *fractional part of a number*, and the second one is already a *property accessor*:

```
1.toString(); // SyntaxError!

(1).toString(); // OK

1 .toString(); // OK (space after 1)

1..toString(); // OK

1['toString'](); // OK
```

Prototype chain

Let's show how to create these prototype chains for the *user-defined* objects. It is quite simple:

```
function A() {
  alert('A.[[Call]] activated');
  this.x = 10;
}
A.prototype.y = 20;

var a = new A();
alert([a.x, a.y]); // 10 (own), 20 (inherited)

function B() {}

// the easiest variant of prototypes
// chaining is setting child
// prototype to new object created,
// by the parent constructor
B.prototype = new A();

// fix .constructor property, else it would be A
B.prototype.constructor = B;

var b = new B();
alert([b.x, b.y]); // 10, 20, both are inherited

// [[Get]] b.x:
// b.x (no) -->
// b.[[Prototype]].x (yes) - 10

// [[Get]] b.y
// b.y (no) -->
// b.[[Prototype]].y (no) -->
// b.[[Prototype]].[[Prototype]].y (yes) - 20

// where b.[[Prototype]] === B.prototype,
// and b.[[Prototype]].[[Prototype]] === A.prototype
```

This approach has two features.

First, `B.prototype` will contain `x` property. At first glance, seems that it is not correct, since `x` property is defined in `A` as *own* and is expected to be *own* as well in objects of the `B` constructor.

In a case of prototypal inheritance though it is normal situation, since the descendant object, if has no such own property delegates to a prototype. The idea behind this is that probably, objects created by the `B` constructor *do not* need `x` property. In contrast, in the class based model, all properties are *copied* to the class-descendant.

However, if nevertheless it is needed (emulating class-based approach) that `x` property be *own* for the objects created by `B` constructor, there are some techniques for this, one of which we will show below.

Secondly, that is already not a feature but the *disadvantage* — the code of the constructor is also executed when the descendant prototype is created. We can see that the message `"A.[[Call]] activated"` is shown *twice* — when the object created by the `A` constructor is used for `B.prototype` and also at creation of object `a` object itself!

A more critical example is a thrown exception in the parent constructor: perhaps, for the *real* objects created by this constructor such checks are needed, but obviously, the same case is completely unacceptable with using these parent objects as prototypes:

```
function A(param) {
  if (!param) {
    throw 'Param required';
```

```
  }
  this.param = param;
}
A.prototype.x = 10;

var a = new A(20);
alert([a.x, a.param]); // 10, 20

function B() {}
B.prototype = new A(); // Error
```

Besides, heavy calculations in the parent constructor can also be considered as disadvantage of this approach.

To solve these "features" and issues, today programmers use standard pattern for chaining the prototypes, which we show below. The main goal of this trick consists in creation of the *intermediate wrapper constructor* which chains the needed prototypes.

```
function A() {
  alert('A.[[Call]] activated');
  this.x = 10;
}
A.prototype.y = 20;

var a = new A();
alert([a.x, a.y]); // 10 (own), 20 (inherited)

function B() {
  // or simply A.apply(this, arguments)
  B.superproto.constructor.apply(this, arguments);
}

// inheritance: chaining prototypes
// via creating empty intermediate constructor
var F = function () {};
F.prototype = A.prototype; // reference
B.prototype = new F();
B.superproto = A.prototype; // explicit reference to ancestor prototype, "sugar"

// fix .constructor property, else it would be A
B.prototype.constructor = B;

var b = new B();
alert([b.x, b.y]); // 10 (own), 20 (inherited)
```

Notice how we create own property `x` on `b` instance: we call parent constructor via the `B.superproto.constructor` reference in context of newly created object.

We have fixed also the issue with non-needed call of the parent constructor for creating the descendant prototype. Now the message `"A.[[Call]] activated"` is shown when is needed.

And for not to repeat every time the same actions of prototypes chaining (creation of the intermediate constructor, setting this `superproto` sugar, restoring the original `constructor` etc.), this template can be encapsulated in the convenient util function, which purpose is to chain prototypes regardless the concrete names of constructors:

```
function inherit(child, parent) {
  var F = function () {};
  F.prototype = parent.prototype
  child.prototype = new F();
  child.prototype.constructor = child;
  child.superproto = parent.prototype;
  return child;
```

```
}
```

Accordingly, inheritance:

```
function A() {}
A.prototype.x = 10;

function B() {}
inherit(B, A); // chaining prototypes

var b = new B();
alert(b.x); // 10, found in the A.prototype
```

There are many variations of such wrappers (in respect of syntax); however, all of them are reduced to the actions described above.

For example, we can optimize the previous wrapper if we will put intermediate constructor outside (thus, only one function will be created), thereby, reusing it:

```
var inherit = (function(){
  function F() {}
  return function (child, parent) {
    F.prototype = parent.prototype;
    child.prototype = new F;
    child.prototype.constructor = child;
    child.superproto = parent.prototype;
    return child;
  };
})();
```

Since the real prototype of an object is the `[[Prototype]]` property, it means that the `F.prototype` can be easily changed and reused, because `child.prototype`, being created via `new F`, will get its `[[Prototype]]` from the the *current* value of `child.prototype`:

```
function A() {}
A.prototype.x = 10;

function B() {}
inherit(B, A);

B.prototype.y = 20;

B.prototype.foo = function () {
  alert("B#foo");
};

var b = new B();
alert(b.x); // 10, is found in A.prototype

function C() {}
inherit(C, B);

// and using our "superproto" sugar
// we can call parent method with the same name

C.prototype.foo = function () {
  C.superproto.foo.call(this);
  alert("C#foo");
};

var c = new C();
alert([c.x, c.y]); // 10, 20

c.foo(); // B#foo, C#foo
```

Note, that ES5 has standardized this util function for better prototypes chaining. It is the `Object.create` method.

Simplified version in ES3 can nearly be implemented in the following way:

```
Object.create ||
Object.create = function (parent, properties) {
  function F() {}
  F.prototype = parent;
  var child = new F;
  for (var k in properties) {
    child[k] = properties[k].value;
  }
  return child;
}
```

Usage:

```
var foo = {x: 10};
var bar = Object.create(foo, {y: {value: 20}});
console.log(bar.x, bar.y); // 10, 20
```

For details see this chapter.

Also, all existing variations of imitations of *"classical inheritance in JS"* are based on this principle. Now we see, that in fact it is even not an "imitation of class based inheritance", but simply a *convenient code reuse for chaining prototypes*.

Conclusion

This article has turned out big enough and detailed. I hope that its material is useful and has dispelled some doubts regarding ECMAScript. If you have any questions or additions, they as always can be discussed in comments.

Additional literature

- 4.2 — Language Overview;
- 4.3 — Definitions;
- 7.8.5 — Regular Expression Literals;
- 8 — Types;
- 9 — Type Conversion;
- 11.1.4 — Array Initialiser;
- 11.1.5 — Object Initialiser;
- 11.2.2 — The new Operator;
- 13.2.1 — [[Call]];
- 13.2.2 — [[Construct]];
- 15 — Native ECMAScript Objects.

Translated by: Dmitry Soshnikov with additions by Garrett Smith.
Published on: 2010-03-04

6.858
Fix

11/16

Jwang: In Ex3 must log on to real site afterwards
he tied iframe but cross origin
he warned of race condition
↳ make sure your email sends last

So how does real log in work?
POST request to /login
; So we should post the real responses there
Just remove return false;

So must wait for img to load;
Then submit form;

So just submitting form does not actually submit it.
Why?
Oh I missed the form target ← thought it was there

(2)

This would be much easier w/ server side scripting

(1) it actually submitted
but didn't load the img

So call back on img?
but how to actually submit?

Why does onload not work?
Need onerror() since not an actual img

But why does it just put us at the log in screen?
It does not redirect
'missing a post param
'wrong referer'

Yeah need submit_login
Just add as hidden param
Since no register
(2) Works

Michael E Plasmeier

From: Taesoo Kim <tsgatesv@gmail.com> on behalf of 6.858-staff <taesoo@MIT.EDU>
Sent: Thursday, November 01, 2012 5:53 PM
To: Michael E Plasmeier
Subject: [6.858] Lab 4 code reviews

Follow Up Flag: Flag for follow up
Flag Status: Flagged

Hi theplaz,

Your lab3 code has been reviewed by two of your classmates. unread

[review0]
---
lab4-code0.txt
- authsvc: passing type = cookie or token returns the login token without security checks? so anyone who can talk to the authsvc can log in as any user this is a
- registersvc: apparently unused, but duplicates functionality in transfersvc, and still gets run with slightly different permissions from transfersvc? probably not actually a vulnerability, but extra attack surface at least...
- transfersvc: relies on transfer.py to do sanity checks on transfers, but it's probably fine since you need a valid token and stuff anyway?
- service calls and RPC flow look generally reasonable
- can't see any obvious problems with permissions
- basically along the same lines as my approach?

- the magic filenames don't seem to do any permissions checks, so profile code can transfer any number of zoobars from anyone to anyone else?? ha did you fix
- otherwise, looks fine...

[review1]
---
lab4-code1.txt

[Section 1 - Privilege Separation]

First off, I noticed that you chose to use uids 1001-1003 in your code; this seemed a bit of an odd choice for me, as uids in the low 1000 range are usually used for regular users -- for example, the user we do the labs as is uid 1000. Also, the uids already used were in the much-more-out-of-the-way 61000s. But that's not necessarily bad, since you control the whole system, and as far as we know there aren't other users 1001-1003 :)

All of your socket directories have at permissions 770 -- I think this is a problem. Anybody who may call the service may also replace the socket with a different socket for a different service, perhaps an attacker-provided service that steals passwords and relaysthem to an attacker-controlled server, or uses them to do nasty things with the real auth and zoobar services.

Also problematic is that you don't appear to have an explicit service for working with the zoobar database, and you allow any of the other services you've created to modify it. The transfer logging service, for example, should certainly not be allowed to modify the zoobar database. The permissions on your auth database, however, are much better. Your person database remains vulnerable to the zookfs service, however (although that's probably okay; we weren't asked to

secure this in the lab), and your transfer service remains vulnerable to the registration service, although that's also a fairly small attack surface.

Your profile.py was moderately deceptive, since you don't use profile.py :) Might want to get rid of that / in general clean up your code. It's probably a pretty bad thing that you print passwords to the web server's log, for example. But the profile.py thing is just a stylistic suggestion. I think it's easier to determine if clean code is secure.

Anyway, now for vulnerabilities in the services themselves:

- If zookfs is compromised, one user may 'make' another user visit their profile at will, because the profile-visiting call to the authentication service doesn't take any token or password for the user. We weren't supposed to allow this.

- Your transfer service doesn't perform any checks on the amounts transfered; this allows for stealing zoobars by transferring negative amounts (if zookfs is compromised and an attacker may call the transfer service socket directly). Additionally, it does both the logging and the changing of zoobars. We were supposed to separate these (see exercise 6).

- Any code that may invoke your transfer service may, if compromised, set a user's zoobar balance to 10, with your 'add' RPC.

- RED ALERT RED ALERT: any attacker may query the auth service (even in their profile) for another user's token, defeating the purpose of the auth service. Don't know how I almost missed that, but that's very bad.

[Section 2 - Python Sandbox]

On the subject of temporary directories: it's probably better to restrict usernames to alphanumeric characters only at registration time, rather than when running the user's profile. This isn't a vulnerability, just bad design.

Your sandbox still lets any user transfer between users without verifying the user, which is bad, but we've been over that. Actually, I don't think your code works; did you not finish the lab? If so, I'm sorry. Hope you got a chance to later.

You allow users to open as writable any file visible to them that the sandbox may open as writable, raising OSError in no cases. Only some files (such as files in the user's temporary directory) should be opened as writable. Similarly, you allow users to write to any fd, not just ones opened writable and checked by you (or not checked by you, as is the case) with open.

Good luck.

Now we would like you to evaluate the two reviews. Assign each a score from 0 to 3 (0=none, 1=poor, 2=fair, 3=good), and submit them with the following template to the submission web page.

[score-lab4.txt]

---

review0: # 2

review1: # 3

Thanks,
6.858 staff

6.858
Code Reviews

11/14
late

These are reviews of my code for lab 3

Review 1

- ~~xxx~~ returns token to anyone
  (that was silly!)

- "how is register service descriptive
  No permission problems
- "how was magic filenames supposed to do
  permission check
  - what did ya do?
- Wish had authoritive solution to compare w/
  or looking at my code ← look it up!
  and able to ask q.

(2)

Review 2

Don't use Uids in low range
- usually regular users

Socket permissions
- I saw this in my review

Zoobars db - no service (Circe)
- really?

So what is he try to say

2 Services
- Zoobar move
- transfer log

Why can't we do 1? ← claims ex 6 said so

What's wrong w/ profile.py?
(provided code?)

Profile visit code should req a token
- I don't remember this req!?

Negative transfer allowed

(3)

Yeah big opps on that

How did we prevent another user from
making to visit profile
thought I read this was allowed

How were we supposed to restrict who could call
transfer service?
must provide token of user from

but then how do profile
'i can profile
w/ from token

Why did I have it call the auth service
Need to stop + think!

Grade said return a yes/no check
"I remember doing that - but in wrong place I think!

(4)

Section 2

└ yes - I cut corners

I thought my code worked

(don't have lab here...)

Must check which files writable

This is all good stuff to think about

```
1   review0:2 Didn't indiciate the magnitude of the security risks.  Missed things
    identified by review1
2   review1:3 Very complete and thorough
```

11/14

Lab 3

4/15

How should you have stored token?

Pizza

11/19
Asked TA

Don't want people to get token
But expect others to run profile
Like setuid

Can run David-i profile
Get token -> but only for profile
So run profile fun
So auth service also calls run profile
Only output return
Zoobar service only returns ~~token~~ profile

Lab 3 and 6 are weird
PY is a sandbox
Caja more hope of getting it right
Lots of rare properties

6.858 Fall 2012 Lab 6: Javascript isolation

Handed out: Wednesday, November 7, 2012
Due: Friday, November 16, 2012 (5:00pm)

Introduction

In this lab, you will implement a system to allow a limited set of Javascript to execute as part of zoobar user profiles. You will implement a combination of static rewriting and dynamic sandboxing to ensure that code running as part of the profile cannot modify the rest of the page, but yet it can make some changes to HTML elements that were part of the profile itself.

To give you an example of the kind of profile code that we will support, a user should be able to place the following code in their zoobar profile:

```
<div id="a">x</div>
<div id="b">x</div>
<div id="c">scrolling message.. </div>
<div id="count"></div>
<script>
    var count = 0;

    function flip(a, b) {
        document.getElementById(a).textContent = "nothing here";
        document.getElementById(b).textContent = "-- click me! --";
        var bump = function (x) { return x+1; }
        count = bump(count);
        document.getElementById('count').textContent = 'click count: ' + count;
    }

    flip('a', 'b');
    document.getElementById('a').onclick = function() { flip('a', 'b'); };
    document.getElementById('b').onclick = function() { flip('b', 'a'); };

    function scroll(id) {
        var s = document.getElementById(id).textContent;
        var ns = s.substring(1) + s[0];
        document.getElementById(id).textContent = ns;
        setTimeout(function() { scroll(id); }, 100);
    }

    scroll('c');
</script>
```

and get a profile that looks like the following:

```
nothing here
-- click me! --
scrolling message..
click count: 1
```

You will build an HTML/Javascript rewriter that will ensure that this code cannot tamper with the rest of the page, steal the cookies, etc.

The system you will be building will be a simpler version of Facebook's FBJS system. You may find it useful to refer to their documentation to understand how their system works, or refer to the paper on Run-Time Enforcement of Secure JavaScript Subsets. Note that Javascript isolation in general is a very difficult problem, and most systems that have been developed have historically turned out to be insecure in a variety of ways. Although we are not aware of any vulnerabilities in the system that you will be building in this lab assignment, it has not been thoroughly vetted or analyzed, and could very well have some subtle holes in it. (If you find any, let us know!)

Log in as the `httpd` user, check in your solution for lab 5, and fetch the new code for lab 6. Note that, for simplicity, you do not

need to integrate changes from previous labs into this lab; we will focus just on rewriting HTML code in profiles for now.

```
httpd@vm-6858:~$ cd lab
httpd@vm-6858:~/lab$ git add answer-1.txt answer-2.html answer-3.html answer-4.txt answer-chal.html
httpd@vm-6858:~/lab$ git commit -am 'my solution to lab5'
[lab5 dc6f228] my solution to lab5
 1 files changed, 1 insertions(+), 0 deletions(-)
httpd@vm-6858:~/lab$ git pull
Already up-to-date.
httpd@vm-6858:~/lab$ git checkout -b lab6 origin/lab6
Branch lab6 set up to track remote branch lab6 from origin.
Switched to a new branch 'lab6'
httpd@vm-6858:~/lab$
```

Now, build and install this code as before. Since the code for lab 6 differs from the code for previous labs, be sure to remove the `/jail` directory and start from scratch for this lab:

```
httpd@vm-6858:~/lab$ make clean
rm -f *.o *.pyc *.bin zookld zookfs zookd zooksvc *.log
httpd@vm-6858:~/lab$ make
...
httpd@vm-6858:~/lab$ sudo rm -rf /jail
[sudo] password for httpd: 6858
httpd@vm-6858:~/lab$ sudo make setup
./chroot-setup.sh
...
httpd@vm-6858:~/lab$
```

Javascript rewriting

To understand how we will isolate Javascript code, let's first examine the new code in this lab. We have implemented a new function, called `filter_html` in `zoobar/htmlfilter.py`, which sanitizes user profiles. This function is invoked from `users.py` on each user profile. The `filter_html` function does three things, as follows.

- First, it parses the HTML using the lxml library, and strips away any dangerous tags and attributes (such as `<style>` tags). It also rewrites the `id` attributes on all HTML elements by prepending the string `sandbox-` to them. This will help us later identify at runtime which DOM elements belong to the profile, and which DOM elements do not.
- Second, this function finds `<script>` tags and uses the Slimit Javascript parsing library to parse and rewrite that code. Parsing first converts the Javascript code into an AST, and then rewriting is done by using a visitor pattern, as implemented in `lab6visitor.py`. The code in `lab6visitor.py` contains a separate method for each AST node, which is invoked recursively: a parent AST node calls `self.visit()` on child nodes. Each such method is responsible for returning a string object representing the rewritten Javascript code. The initial code in `lab6visitor.py` that we provide does not modify the Javascript at all, and simply prints back the exact code corresponding to the AST.
- Third, to help with Javascript rewriting, the `htmlfilter.py` code adds a trusted library (included inline at the top of `htmlfilter.py` as `libcode`). This library exports trusted interfaces that the rewritten code can access, such as a safe way of accessing the profile's DOM objects.

We have constructed a number of test cases to help you debug your Javascript sandboxing system. They are stored in `profiles`, and include the sample profile above with the annoying scrolling message (`demo.html`), an automated test case checking that this example profile works (`good-all.html`), and thirteen different malicious profiles that you will need to confine (`bad-00-eval.html` through `bad-12-definegetter.html`).

You can invoke the HTML / Javascript rewriter by running `zoobar/filter-test.py`; it reads profile code as input and prints out sandboxed HTML and Javascript. For example:

```
httpd@vm-6858:~/lab$ ./zoobar/filter-test.py < ./profiles/bad-00-eval.html
...
var s = "window.location = 'http://localhost:8900/test-bad';";
eval(s);</script></div>
httpd@vm-6858:~/lab$
```

To isolate Javascript, you will take the following approach:

- First, you will rewrite all function and variable names to prepend a unique prefix, `sandbox_`, to ensure that the code cannot directly access any functions or objects natively exported by the browser (such as `eval()`, the `window` object, the `document` object, etc). This way, if the code tries to invoke `eval()`, the rewritten version will invoke `sandbox_eval()`; since that function is not defined (or defined by the sandboxed code to point to some other sandboxed code), invoking that function will not allow escaping from the sandbox. Note that attributes like `bar` in `foo.bar` do not get prefixed. Neither do unquoted identifiers in object literals, like `{ x: 1, y: 2 }`.

  This will break all the tests until you also extend the trusted library (`libcode`) to support the `setTimeout()` Javascript function and the `textContent` property of DOM elements. The original functions are no longer accessible prefixed. Be sure to guard against possible attacks through these interfaces; the native `setTimeout()` function allows specifying the callback object either as a function or as a string (which then gets `eval()`ed). We use `textContent` over `innerHTML` because `textContent` creates a text node directly without interpreting as HTML, so you needn't sanitize the input for additional `script` tags.

- Second, you will need to prevent the sandboxed code from accessing dangerous properties of objects. For example, each object in Javascript has a property that gives access to that object's *prototype*, which you can think of as being similar to the object's "class" in traditional object-oriented languages. Having access to the prototype allows changing the methods of that prototype ("class"). For example, an adversary can change the `substring` method of *all* strings by doing something like:

  ```
  var s = "any string";
  s.__proto__.substring = function() { return "gotcha!"; };
  ```

  This is dangerous because it affects how other objects in the system behave, including objects used by other (trusted) Javascript code in the same page. Other methods allow indirect access to `eval`-like functionality, such as the `Function` constructor:

  ```
  var f = function() { return 0; };
  var newfunc = f.constructor("alert(document.cookie);");
  newfunc();
  ```

  And `__defineGetter__` may be called outside an object, in which case you define variables in global scope, which can confuse the outside code:

  ```
  var f = [].__defineGetter__;
  f("foo", function () { ... });
  ```

  To prevent these attacks, you will need to find all instances where an object's property is accessed (as `objname.propname`), and check that `propname` is not one of the dangerous attributes: `__proto__`, `constructor`, `__defineGetter__`, and `__defineSetter__`. If it is, replace it with `__invalid__` or so. Raising an exception will also work, but the tests will be unhappy.

- Third, you will need to ensure that dangerous attributes cannot be accessed using array-like brackets (e.g., `object['__proto__']`), even if the array index inside of the brackets is a variable whose value is only known at runtime. To do this, we suggest using the same trick as FBJS uses: rewrite statements like `o[i]` to `o[bracket_check(i)]`, where `bracket_check()` is a Javascript function that you include in your trusted `libcode`, which checks if its argument is one of the dangerous attributes (listed above), and if not, returns its argument verbatim.

  Be careful, in the implementation of `bracket_check`, of objects with custom `toString` or `valueOf` methods. An adversary can pass in an object which stringifies as a safe string the first time, and as a blacklisted string the second.

- Finally, you will need to work around JavaScript's handling of the `this` keyword. If a function is called directly, as opposed to as a property of an object, `this` refers to the global object (in a browser, this is `window`). This would allow the sandboxed code to access unprefixed global variables. We recommend using a similar trick to FBJS: rewrite instances of `this` to a call to `this_check(this)`, where `this_check` returns `null` if its argument was `window`.

Exercise. Implement Javascript sandboxing as described above. You will need to modify `lab6visitor.py` and

`libcode` in `htmlfilter.py`.

Make sure that your sandbox works correctly with the `demo.html` profile, and stops the attacks in `bad-*.html` profiles. You can test this profile code by uploading it into (and viewing it through) the zoobar site on your VM. Alternatively, you can manually test it by running the profile code through `./zoobar/filter-test.py` (as shown above), and then loading the resulting HTML code in your browser. It will redirect to a URL containing either `test-ok`, `test-bad`, or `test-broken`.

You can check whether your system works correctly by running `make check`. This can take some time, because it spawns a fresh Firefox web browser in your VM to check each profile. If your profile does not work correctly, the script aborts after a 30-second timeout. If your VM is particularly slow, and you find that this timeout fires prematurely, you can increase this timeout in `test-url.sh` (e.g., change the `sleep 30` command to `sleep 120`).

You are done! Run `make submit` to upload your answers.

Lab 6
Javascript isolation

11/14
Plaxe

JS sandbox

So it looks like a fair bit has already been done!

lets check at the code

Some lib code which has grader

and some other functions

Use Xml cleaner

Nothing ever calls filter_js...

---

try out base case + see what happens

Canvaying working on plaxe

So it auto added the header

and converted line breaks to &#13

but has not seemed to lock the js

that it didn't run it...

Oh to the HTML it changed the div id
Cleaner

Why is cleaner.scripts = False?
(prob just removes them altogether

Function js rewrite
Ok that is called
Oh that is a fn
Which is filter_js - makes sense now

So that is what we want to add
Currently HTML just adds elements
(the hard part here is making sure it is comprehensive..
Well labbevisitor
which is based on slimit
↳ just a minifier

(3)

Slimit also lets us parse

Converts JS to AST

↑ abstract syntax tree

Then sep code for each type of node

in-volked recursivly

self. visit on childern

each returns string obj w/ rewritten js

So then what we actually need to do

1. Rewrite all fns and variable names to append

Sandbox —

except foo, bar

↑ don't touch

{x: 1, y: 2}

(4)

Ok now in what fns do we do this?
What nodes are in our visitor?
"Or whatever they are..."

node. __class__ ~~name~~. __name__

Var statement / dec ✓ changed

what is % for strings?
fill in value in where %s is

what is get attr?
what does Assign do?
Expr Statement?
Func Decl?
└ change ✓

Func Call ✓
Array ✓ ; did we have to change there
not here I think

(5)

this
↳ that other thing
to later

We still need just plan var used

'i is that in fn arguments
those get visited seperlly
'i how/where

'i how could ya debug this
↳ break points

For In loop

This runs as py on server
hard to breakpoint...

Ah but can print node!

(6)

Hmm no changes visible
What is wrong?

Readers on shaw

Oh - runs at of /jail
---- Fix permissions

Got some things to work!
Oh functions — (underscore)
variable — (dash)

document. exception
Fn defn
variables in fn
Set Timeout exception!

Oh it just printed "Identifier"
(to string)

6

Oh there is a line in there for Identifier
look at that
('sideeffects('

That applied to way too much
('we don't know its context
Must apply further up...
(I got the hang of this lab now!)

I Think I have basic filtering working!
Now the other stuff

So we correctly block the document. object

Extend libcode to support setTimeout()
and textContent properties
(Oh that is how ya do sleep!)

8

Sandbox_document is pre-written
Supports get el by id

Cool get/set
did not know you could do that!

So must add our own setTimeout like this
and be sure to filter!

How do we filter?
Skip for now
(get it working 1st)

How do we do textContent over innerHTML?
Ok there code does it
Not something we need to filter for

(9)

Prototype – prevent access to do

or __defineGetter__

__proto__

"the other way"

not in scope here perhaps

Check for object property

objname.property

I remember seeing that...

Nothing in this example

Skip for now

Brackets check

object["__proto__"]

So o[bracketcheck(i)] for o[i]

(10)

btp that out later

This

this - check (this)

So this on Interpeter?

Or where is this used?

All of those were not in our simple good ( )

checker

try to fix that - last

then present bad stuff

So getElementById( )

(not rewritten

Or is elements in text quotes

↳ should we replace

I guess so...

(11)

So String
Func'Expr

Now code totally disappeared!
↳ parse error

---

did too much — can't see what...

So added debug for string
node.value includes ' '

✓ Built this complicated text filter

---

but get ElById still bad

Opps had done this wrong
↳ well suboptimal

(12)

Where is getElById

Or its all text...

Listed all

'i dot accessor...

Or the one afterwards...

dont want sandbox->getEl

but getEl(sandbox->)

Identifiers

One is getElementById

One is a

but how do you tell

trace upwards

FunctionCall:

Yes instrument its arguments

(13)

but this screwed up the one w/ text node

Sandbox - 'sandbox-a'

I'm going down the wrong path...
this is not correct...

Woot think got that

Non why is it not working?

Are - not allowed?

Error in header → spec semicolon?

No can't spell fuction!

Wait that is silly
replace sandbox - w/ sandbox 0

Or all of these should be _ (underscore)

(14)

Why Sandbox — return ;

And on fn add a 2nd return

Prob that hack I did...

---

already there...

Func Expr

So I put in #s to track it down

Now error in template code...

Oh argument should not be marked

Oh should not have touched text!!!

@ Ohh grr hrs of wasted work

Now no errors on a, b

So why no work?

(45)

Sandbox_document is not working....
but I didn't change it!
and I don't know what is going on here!
w/ returning a fn...

rewriting it gets the text to show up!
but onclick prob broken...

ho it works!

Opps I sandbox an int...
forget how to check an int
that entire thing needs to be redone...

Ok now some new things I am not
doing ~~w~~ right

S[check_bracket(0)]
↑
~~Shall~~ should rewrite...

Oeasy

(16)

Now equality ... = ns

? Sandbox...

lol search for = %s

Unless we want to go back to every Instance

↳ no!

essentially if Arg is identifier

- not #, String
- fix later

ns is a node

↳ just an identifier

I think this is assign...

But how differentiate it?

A direct identifier

↳ I put a crappy fix in

? How do better?

(17)

Another non identifier
do the identifier thing

✓ seems to have fixed!

✓ scrolling works!

✓ text switch works

✗ clicking doesn't

Count not rewritten

That is the left side of before ...

And also textContent ...

One assignment left

One binary Op

So made general checkIdent + Visit ()

✓ assignment

(18)

Bin Op

prob same

✓ Fixed

---

i do variables inside fn

function (x) {

prob should

}

✓ Fixed

✓ Done base profile

(19)

11/15
9p

Now the bad cases

Lets try

and some things I didn't do

Or do good all.html

Oh command to print w/ cmd line opps

(X) 372 prop for node prop needs 2 arguments

Oh property filtering never can be before!

I should set up auto submit...

Oh that runs non jail

✓ Fixed

20

My server is not running!
Well localhost

---

is it the hotel wired?
restart Putty

Oh well use IP site
hopefully does not matter here...
Oh it checks - rewrite...

Counter. prototype undefined

So this is an identifier Prototype
Part of DotAccessor
have it check there...
removed
should we have?
appears to be good code

(21)

skip for now
it is Counter. invalid

Sandbox_sandbox_document
wtf?

Expr Statement
Function Call
Dot accessor

Prob sep diag
No - both funct call...
And why not on other one when argument?

Now it added 3!
so it runs on each accessor...
That prevents one... but still lot on there...

Why does my string matching not work?

(22)

So somehow that does not make a difference!

Why?

It replies right thing!

'or don't add'

No - most sometimes....

Why does Func Call sometimes not do it..

Moved down

✓ Better

✗ error in sandbox ~~add~~ f(sandbox_id)

not enough args

Prototype ✓ 300

do allan, Prototype!

---

code: sandbox_f > scroll

that should work...

Same as original...

(23)

But not sandbox — scroll
The problem!

Var Decl
When 2nd identifier
✓ Fixed

Counter.prototype undefined

Should be sandbox_Counter

Also new Counter()

New Expr

✓ Added my checkExpr fn

ExprStatement's
oh should be recursive---
Put check on left of DotAssignExpr

(24)

(X) Check_this not defined

(✓) Fixed -w/ passthrough

No errors - s not working
update code!

(✓) Works
clicking
messages

(X) No scrolling

(X) sandbox — window not defined
that is in grade...

(✓) works when I remove all grades
but

(X) test broken 6
Oh Array access

(25)

So array is written as

Sandbox_Foo

Should not be touched

Var decl?

✓ Fixed

✓ tests pass

but click stuts at 3

Oh by design

Non fail tests

~~✓~~ OO; eval ✓

Oh should redirect

when remove grader line

✓ passes

26

01: Eval write

✓ pass

pasted problem w/ grader on Piazza @ 34

02: Win

Window location

✓ pass

03: This

✓ passes

but get

sandbox_~~this~~check_this()

Can do special check...

Or define sandbox_check_this...

prob deeper problem somewhere...

(27)

I pass it through + it works ....

Oh more on bracket...

04 Constctor

(1) passes

05 Constructa Bracket

(1) passes

06 bracket To String

slimit error

idoes that cant?

Skip for now...

07 new

We overwite new...

that is bad

I think...

iAgain what is doing that?

bit hack in Fn call
✓ Fixes

✓ passes

08 Set tiner

I don't think I got
Most file strings
since we don't parse
Hmm seemed to work - IDk

✓ passes

09 Tiner to string
Save slimit error

10 Proto
Some other syntax error
Opps
Func Call again!
tried to just check for Identifier

19

✓ Passed

Really? was that simple!

---

11 Proto Bracket

✓ passed

p is not defined

in check_bracket

↳ how does check bracket just work?

Or I see proto → make invalid

Idk why that works!

Sandbox_p not rewritten!

12 Define getter ...

again Idk why this passes

but it does

✓ passes

✓ fixed to be invalid

(30)

03 This

Return null if arg = window

✓ die

✓ still passes

Lactually mentions null

06 Bracket to string

the Js limit error

So before thought I knew where...

Object

Assign

That memory loc not anywhere in log!

I see what I did!

Key is not rewritten...

✓ Fixed

(51)

(X) to StringConst not defined....

↳ multiple places

~~Var Dec~~

~~For Expr~~

Unary Op

(✓) Fixed w/ checkVisit Ident

also return to StringConst

(✓) Fixed w/ same way

---

f is not defined

if (f)

I have a feeling - better sol than that!
checkVisitIdent

(✓) Passes cleanly

Prob a bunch I have not seen...

But fired earlier & not all IntLits

Ideally should recheck...

(32)

11 Proto Bracket

✓ passes silently

--proto-- was not on blacklist

Oh when test...

? How do bracket check?

blacklist check again!,

Skip...

04 Set timeout to string

✓ passes silently

08 set Timeout

✓ still works

Idk why this works...

Put some code in for fun

:)

(33)

Make check

Will be extra output

All error
Oh from my print comment ech
Broke some stuff

Says Sandbox Unknown or el...
Which means could not reach URL

Hmm just submit...
(slow here at hotel!)
But that should not affect it...
Unless this is the document window thing
Still open on Piazza
See tmo

B4

11/16
EA

TA!

So emitting variables wrong

~~Sandbox~~

isn't it correct?

Ok should work?

11/18
3P

QBO1 Means timeout is too short

Try check-experimental

Or was my localhost not working

Not able to get it work

even when new window

Q: But code does not seem to use localhost

Check-experimental Timeouts too...

But also gives errors

What is it checking?

But we want errors!

PhantomJS is a headless webkit browser...

(2)

Still timing out...

Leven w/ 120

But local host should not matter here...

11/19
OH

Original code ran in set timeout

Since last min added check ← was rather stupid

Not actually running server

11/19
am

① error 5

but all bad cases pass ②

~~then~~ which is textContent check

So need to get it to work to check

Why not running! ;

Copied to html file

Oh it worked

(well broke)

(2)

No @ Fieby error

So put in an alert

Illing message undefined undefined undefined

So doesn't append 500 either

The checkBracket

Always returns null

Since -1

Index of does not work

Tried a few things

(1) Tests pass

but failed 5, 6, 11 bad

(3)

11 has s[__prototype__]

Hmm ok runs in ff ...

Sometimes ...

05

Also runs in FF

but fails on check-experimental

Check still seems to fail

Oh no initalize check_bracket

var i = 0

gcc, sloppy

✓ 5, 11

✗ still 6

(4)

So it appears that is a fn

So need to check for those

✓ 6 works

✗ 6 and broke 5

Oh number

L also good

✓ all tests pass

resubmit + email TA

Lab 6

100/100 :)

```
Username: theplaz

Output of make check:
./check-lab6-experimental.sh
------------------------------------------------
Profile: ./profiles/good-all.html
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-00-eval.html
  JS error: ReferenceError: Can't find variable: sandbox_eval
    file:///tmp/sb.16894.html: 64
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-01-eval-copy.html
  JS error: ReferenceError: Can't find variable: sandbox_eval
    file:///tmp/sb.16894.html: 64
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-02-win.html
  JS error: ReferenceError: Can't find variable: sandbox_window
    file:///tmp/sb.16894.html: 63
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-03-this.html
  JS error: TypeError: 'null' is not an object (evaluating 'sandbox_check_this(this)[check_bracket('ev' + 'al')]')
    file:///tmp/sb.16894.html: 64
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-04-constructor.html
  JS error: TypeError: 'undefined' is not a function (evaluating 'sandbox_f.__invalid__(sandbox_s)')
    file:///tmp/sb.16894.html: 67
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-05-constructor-bracket.html
  JS error: TypeError: 'undefined' is not a function (evaluating 'sandbox_f[check_bracket('' + sandbox_i + 'or')](sandbox_s)')
    file:///tmp/sb.16894.html: 68
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-06-bracket-tostring.html
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-07-new.html
  JS error: ReferenceError: Can't find variable: sandbox_Function
    file:///tmp/sb.16894.html: 64
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-08-settimeout.html
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-09-settimeout-tostring.html
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-10-proto.html
  JS error: TypeError: 'undefined' is not an object (evaluating 'sandbox_s.__invalid__.toString = function() {
  return "window.location = 'http://localhost:8900/test-bad';";
}')
    file:///tmp/sb.16894.html: 71
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-11-proto-bracket.html
  JS error: TypeError: 'undefined' is not an object (evaluating 'sandbox_s[check_bracket(sandbox_p)].toString = function() {
  return "window.location = 'http://localhost:8900/test-bad';";
}')
    file:///tmp/sb.16894.html: 72
Sandbox: OK
------------------------------------------------
Profile: ./profiles/bad-12-definegetter.html
  JS error: TypeError: 'undefined' is not a function (evaluating 'sandbox_f')
    file:///tmp/sb.16894.html: 76
Sandbox: OK
```

Read 11/18

Remote Timing Attacks are Practical

David Brumley
Stanford University
dbrumley@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

Abstract

Timing attacks are usually used to attack weak computing devices such as smartcards. We show that timing attacks apply to general software systems. Specifically, we devise a timing attack against OpenSSL. Our experiments show that we can extract private keys from an OpenSSL-based web server running on a machine in the local network. Our results demonstrate that timing attacks against network servers are practical and therefore security systems should defend against them.

1 Introduction

Timing attacks enable an attacker to extract secrets maintained in a security system by observing the time it takes the system to respond to various queries. For example, Kocher [10] designed a timing attack to expose secret keys used for RSA decryption. Until now, these attacks were only applied in the context of hardware security tokens such as smartcards [4, 10, 18]. It is generally believed that timing attacks cannot be used to attack general purpose servers, such as web servers, since decryption times are masked by many concurrent processes running on the system. It is also believed that common implementations of RSA (using Chinese Remainder and Montgomery reductions) are not vulnerable to timing attacks.

We challenge both assumptions by developing a remote timing attack against OpenSSL [15], an SSL library commonly used in web servers and other SSL applications. Our attack client measures the time an OpenSSL server takes to respond to decryption queries. The client is able to extract the private key stored on the server. The attack applies in several environments.

Network. We successfully mounted our timing attack between two machines on our campus network. The attacking machine and the server were in different buildings, with three routers and multiple switches between them. With this setup we were able to extract the SSL private key from common SSL applications such as a web server (Apache+mod_SSL) and a SSL-tunnel.

Interprocess. We successfully mounted the attack between two processes running on the same machine. A hosting center that hosts two domains on the same machine might give management access to the admins of each domain. Since both domain are hosted on the same machine, one admin could use the attack to extract the secret key belonging to the other domain.

Virtual Machines. A Virtual Machine Monitor (VMM) is often used to enforce isolation between two Virtual Machines (VM) running on the same processor. One could protect an RSA private key by storing it in one VM and enabling other VM's to make decryption queries. For example, a web server could run in one VM while the private key is stored in a separate VM. This is a natural way of protecting secret keys since a break-in into the web server VM does not expose the private key. Our results show that when using OpenSSL the network server VM can extract the RSA private key from the secure VM, thus invalidating the isolation provided by the VMM. This is especially relevant to VMM projects such as Microsoft's NGSCB architecture (formerly Palladium). We also note that NGSCB enables an application to ask the VMM (aka Nexus) to decrypt (aka unseal) application data. The application could expose the VMM's secret key by measuring the time the VMM takes to respond to such requests.

Many crypto libraries completely ignore the timing attack and have no defenses implemented to prevent it. For example, libgcrypt [14] (used in GNUTLS and GPG) and Cryptlib [5] do not defend against timing attacks. OpenSSL 0.9.7 implements a defense against the timing attack as an option. However, common applications such as mod_SSL, the Apache SSL module, do not en-

able this option and are therefore vulnerable to the attack. These examples show that timing attacks are a largely ignored vulnerability in many crypto implementations. We hope the results of this paper will help convince developers to implement proper defenses (see Section 6). Interestingly, Mozilla's NSS crypto library properly defends against the timing attack. We note that most crypto acceleration cards also implement defenses against the timing attack. Consequently, network servers using these accelerator cards are not vulnerable.

We chose to tailor our timing attack to OpenSSL since it is the most widely used open source SSL library. The OpenSSL implementation of RSA is highly optimized using Chinese Remainder, Sliding Windows, Montgomery multiplication, and Karatsuba's algorithm. These optimizations cause both known timing attacks on RSA [10, 18] to fail in practice. Consequently, we had to devise a new timing attack based on [18, 19, 20, 21, 22] that is able to extract the private key from an OpenSSL-based server. As we will see, the performance of our attack varies with the exact environment in which it is applied. Even the exact compiler optimizations used to compile OpenSSL can make a big difference.

In Sections 2 and 3 we describe OpenSSL's implementation of RSA and the timing attack on OpenSSL. In Section 4 we discuss how these attacks apply to SSL. In Section 5 we describe the actual experiments we carried out. We show that using about a million queries we can remotely extract a 1024-bit RSA private key from an OpenSSL 0.9.7 server. The attack takes about two hours.

Timing attacks are related to a class of attacks called side-channel attacks. These include power analysis [9] and attacks based on electromagnetic radiation [16]. Unlike the timing attack, these extended side channel attacks require special equipment and physical access to the machine. In this paper we only focus on the timing attack. We also note that our attack targets the implementation of RSA decryption in OpenSSL. Our timing attack does not depend upon the RSA padding used in SSL and TLS.

2 OpenSSL's Implementation of RSA

We begin by reviewing how OpenSSL implements RSA decryption. We only review the details needed for our attack. OpenSSL closely follows algorithms described in the Handbook of Applied Cryptography [11], where more information is available.

2.1 OpenSSL Decryption

At the heart of RSA decryption is a modular exponentiation $m = c^d \bmod N$ where $N = pq$ is the RSA modulus, d is the private decryption exponent, and c is the ciphertext being decrypted. OpenSSL uses the Chinese Remainder Theorem (CRT) to perform this exponentiation. With Chinese remaindering, the function $m = c^d \bmod N$ is computed in two steps. First, evaluate $m_1 = c^{d_1} \bmod p$ and $m_2 = c^{d_2} \bmod q$ (here d_1 and d_2 are precomputed from d). Then, combine m_1 and m_2 using CRT to yield m.

RSA decryption with CRT gives up to a factor of four speedup, making it essential for competitive RSA implementations. RSA with CRT is not vulnerable to Kocher's original timing attack [10]. Nevertheless, since RSA with CRT uses the factors of N, a timing attack can expose these factors. Once the factorization of N is revealed it is easy to obtain the decryption key by computing $d = e^{-1} \bmod (p-1)(q-1)$.

2.2 Exponentiation

During an RSA decryption with CRT, OpenSSL computes $c^{d_1} \bmod p$ and $c^{d_2} \bmod q$. Both computations are done using the same code. For simplicity we describe how OpenSSL computes $g^d \bmod q$ for some g, d, and q.

The simplest algorithm for computing $g^d \bmod q$ is *square and multiply*. The algorithm squares g approximately $\log_2 d$ times, and performs approximately $\frac{\log_2 d}{2}$ additional multiplications by g. After each step, the product is reduced modulo q.

OpenSSL uses an optimization of square and multiply called *sliding windows* exponentiation. When using sliding windows a block of bits (window) of d are processed at each iteration, where as simple square-and-multiply processes only one bit of d per iteration. Sliding windows requires pre-computing a multiplication table, which takes time proportional to $2^{w-1}+1$ for a window of size w. Hence, there is an optimal window size that balances the time spent during precomputation vs. actual exponentiation. For a 1024-bit modulus OpenSSL uses a window size of five so that about five bits of the exponent d are processed in every iteration.

For our attack, the key fact about sliding windows is that during the algorithm there are many multiplications by g, where g is the input ciphertext. By querying on many

inputs g the attacker can expose information about bits of the factor q. We note that a timing attack on sliding windows is much harder than a timing attack on square-and-multiply since there are far fewer multiplications by g in sliding windows. As we will see, we had to adapt our techniques to handle sliding windows exponentiation used in OpenSSL.

2.3 Montgomery Reduction

The sliding windows exponentiation algorithm performs a modular multiplication at every step. Given two integers x, y, computing $xy \bmod q$ is done by first multiplying the integers $x * y$ and then reducing the result modulo q. Later we will see each reduction also requires a few additional multiplications. We first briefly describe OpenSSL's modular reduction method and then describe its integer multiplication algorithm.

Naively, a reduction modulo q is done via multi-precision division and returning the remainder. This is quite expensive. In 1985 Peter Montgomery discovered a method for implementing a reduction modulo q using a series of operations efficient in hardware and software [13].

Montgomery reduction transforms a reduction modulo q into a reduction modulo some power of 2 denoted by R. A reduction modulo a power of 2 is faster than a reduction modulo q as many arithmetic operations can be implemented directly in hardware. However, in order to use Montgomery reduction all variables must first be put into Montgomery form. The Montgomery form of number x is simply $xR \bmod q$. To multiply two numbers a and b in Montgomery form we do the following. First, compute their product as integers: $aR * bR = cR^2$. Then, use the fast Montgomery reduction algorithm to compute $cR^2 * R^{-1} = cR \bmod q$. Note that the result $cR \bmod q$ is in Montgomery form, and thus can be directly used in subsequent Montgomery operations. At the end of the exponentiation algorithm the output is put back into standard (non-Montgomery) form by multiplying it by $R^{-1} \bmod q$. For our attack, it is equivalent to use R and $R^{-1} \bmod N$, which are public.

Hence, for the small penalty of converting the input g to Montgomery form, a large gain is achieved during modular reduction. With typical RSA parameters the gain from Montgomery reduction outweighs the cost of initially putting numbers in Montgomery form and converting back at the end of the algorithm.

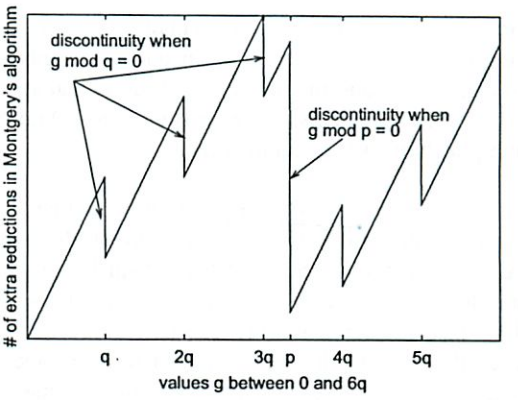


Figure 1: Number of extra reductions in a Montgomery reduction as a function (equation 1) of the input g.

The key relevant fact about a Montgomery reduction is at the end of the reduction one checks if the output cR is greater than q. If so, one subtracts q from the output, to ensure that the output cR is in the range $[0, q)$. This extra step is called an *extra reduction* and causes a timing difference for different inputs. Schindler noticed that the probability of an extra reduction during an exponentiation $g^d \bmod q$ is proportional to how close g is to q [18]. Schindler showed that the probability for an extra reduction is:

$$\Pr[\texttt{Extra Reduction}] = \frac{g \bmod q}{2R} \qquad (1)$$

Consequently, as g approaches either factor p or q from below, the number of extra reductions during the exponentiation algorithm greatly increases. At exact multiples of p or q, the number of extra reductions drops dramatically. Figure 1 shows this relationship, with the discontinuities appearing at multiples of p and q. By detecting timing differences that result from extra reductions we can tell how close g is to a multiple of one of the factors.

2.4 Multiplication Routines

RSA operations, including those using Montgomery's method, must make use of a multi-precision integer multiplication routine. OpenSSL implements two multiplication routines: Karatsuba (sometimes called recursive) and "normal". Multi-precision libraries represent large integers as a sequence of words. OpenSSL uses Karatsuba multiplication when multiplying two numbers with an equal number of words. Karatsuba multiplication takes time $O(n^{\log_2 3})$ which is $O(n^{1.58})$. OpenSSL uses

normal multiplication, which runs in time $O(nm)$, when multiplying two numbers with an unequal number of words of size n and m. Hence, for numbers that are approximately the same size (i.e. n is close to m) normal multiplication takes quadratic time.

Thus, OpenSSL's integer multiplication routine leaks important timing information. Since Karatsuba is typically faster, multiplication of two unequal size words takes longer than multiplication of two equal size words. Time measurements will reveal how frequently the operands given to the multiplication routine have the same length. We use this fact in the timing attack on OpenSSL.

In both algorithms, multiplication is ultimately done on individual words. The underlying word multiplication algorithm dominates the total time for a decryption. For example, in OpenSSL the underlying word multiplication routine typically takes 30% – 40% of the total runtime. The time to multiply individual words depends on the number of bits per word. As we will see in experiment 3 the exact architecture on which OpenSSL runs has an impact on timing measurements used for the attack. In our experiments the word size was 32 bits.

2.5 Comparison of Timing Differences

So far we identified two algorithmic data dependencies in OpenSSL that cause time variance in RSA decryption: (1) Schindler's observation on the number of extra reductions in a Montgomery reduction, and (2) the timing difference due to the choice of multiplication routine, i.e. Karatsuba vs. normal. Unfortunately, the effects of these optimizations counteract one another.

Consider a timing attack where we decrypt a ciphertext g. As g approaches a multiple of the factor q from below, equation (1) tells us that the number of extra reductions in a Montgomery reduction increases. When we are just over a multiple of q, the number of extra reductions decreases dramatically. In other words, decryption of $g < q$ should be slower than decryption of $g > q$.

The choice of Karatsuba vs. normal multiplication has the opposite effect. When g is just below a multiple of q, then OpenSSL almost always uses fast Karatsuba multiplication. When g is just over a multiple of q then $g \bmod q$ is small and consequently most multiplications will be of integers with different lengths. In this case, OpenSSL uses normal multiplication which is slower. In other words, decryption of $g < q$ should be faster than decryption of $g > q$ — the exact opposite of the effect of extra reductions in Montgomery's algorithm. Which effect dominates is determined by the exact environment. Our attack uses both effects, but each effect is dominant at a different phase of the attack.

3 A Timing Attack on OpenSSL

Our attack exposes the factorization of the RSA modulus. Let $N = pq$ with $q < p$. We build approximations to q that get progressively closer as the attack proceeds. We call these approximations guesses. We refine our guess by learning bits of q one at a time, from most significant to least. Thus, our attack can be viewed as a binary search for q. After recovering the half-most significant bits of q, we can use Coppersmith's algorithm [3] to retrieve the complete factorization.

Initially our guess g of q lies between 2^{512} (i.e. $2^{\log_2 N/2}$) and 2^{511} (i.e. $2^{\log_2(N/2)-1}$). We then time the decryption of all possible combinations of the top few bits (typically 2-3). When plotted, the decryption times will show two peaks: one for q and one for p. We pick the values that bound the first peak, which in OpenSSL will always be q.

Suppose we already recovered the top $i-1$ bits of q. Let g be an integer that has the same top $i-1$ bits as q and the remaining bits of g are 0. Then $g < q$. At a high level, we recover the i'th bit of q as follows:

- Step 1 - Let g_{hi} be the same value as g, with the i'th bit set to 1. If bit i of q is 1, then $g < g_{hi} < q$. Otherwise, $g < q < g_{hi}$.
- Step 2 - Compute $u_g = gR^{-1} \bmod N$ and $u_{g_{hi}} = g_{hi}R^{-1} \bmod N$. This step is needed because RSA decryption with Montgomery reduction will calculate $u_g R = g$ and $u_{g_{hi}} R = g_{hi}$ to put u_g and $u_{g_{hi}}$ in Montgomery form before exponentiation during decryption.
- Step 3 We measure the time to decrypt both u_g and $u_{g_{hi}}$. Let $t_1 = \text{DecryptTime}(u_g)$ and $t_2 = \text{DecryptTime}(u_{g_{hi}})$.
- Step 4 - We calculate the difference $\Delta = |t_1 - t_2|$. If $g < q < g_{hi}$ then, by Section 2.5, the difference Δ will be "large", and bit i of q is 0. If $g < g_{hi} < q$, the difference Δ will be "small", and bit i of q is 1. We use previous Δ values to know what to consider "large" and "small". Thus we use the value $|t_1 - t_2|$ as an indicator for the i'th bit of q.

When the i'th bit is 0, the "large" difference can either be negative or positive. In this case, if $t_1 - t_2$ is positive then DecryptTime(g) > DecryptTime(g_{hi}), and the Montgomery reductions dominated the time difference. If $t_1 - t_2$ is negative, then DecryptTime(g) < DecryptTime(g_{hi}), and the multi-precision multiplication dominated the time difference.

Formatting of RSA plaintext, e.g. PKCS 1, does not affect this timing attack. We also do not need the value of the decryption, only how long the decryption takes.

3.1 Exponentiation Revisited

We would like $|t_{g_1} - t_{g_2}| \gg |t_{g_3} - t_{g_4}|$ when $g_1 < q < g_2$ and $g_3 < g_4 < q$. Time measurements that have this property we call a strong indicator for bits of q, and those that do not are a weak indicator for bits of q. Square and multiply exponentiation results in a strong indicator because there are approximately $\frac{\log_2 d}{2}$ multiplications by g during decryption. However, in sliding windows with window size w ($w = 5$ in OpenSSL) the expected number of multiplications by g is only:

$$E[\texttt{\# multiply by g}] \approx \frac{\log_2 d}{2^{w-1}(w+1)}$$

resulting in a weak indicator.

To overcome this we query at a *neighborhood* of values $g, g+1, g+2, ..., g+n$, and use the result as the decrypt time for g (and similarly for g_{hi}). The total decryption time for g or g_{hi} is then:

$$T_g = \sum_{i=0}^{n} \texttt{DecryptTime}(g+i)$$

We define T_g as the time to compute g with sliding windows when considering a neighborhood of values. As n grows, $|T_g - T_{g_{hi}}|$ typically becomes a stronger indicator for a bit of q (at the cost of additional decryption queries).

4 Real-world scenarios

As mentioned in the introduction there are a number of scenarios where the timing attack applies to networked servers. We discuss an attack on SSL applications, such as stunnel [23] and an Apache web server with mod_SSL [12], and an attack on trusted computing projects such as Microsoft's NGSCB (formerly Palladium).

During a standard full SSL handshake the SSL server performs an RSA decryption using its private key. The SSL server decryption takes place after receiving the CLIENT-KEY-EXCHANGE message from the SSL client. The CLIENT-KEY-EXCHANGE message is composed on the client by encrypting a PKCS 1 padded random bytes with the server's public key. The randomness encrypted by the client is used by the client and server to compute a shared master secret for end-to-end encryption.

Upon receiving a CLIENT-KEY-EXCHANGE message from the client, the server first decrypts the message with its private key and then checks the resulting plaintext for proper PKCS 1 formatting. If the decrypted message is properly formatted, the client and server can compute a shared master secret. If the decrypted message is not properly formatted, the server generates its own random bytes for computing a master secret and continues the SSL protocol. Note that an improperly formatted CLIENT-KEY-EXCHANGE message prevents the client and server from computing the same master secret, ultimately leading the server to send an ALERT message to the client indicating the SSL handshake has failed.

In our attack, the client substitutes a properly formatted CLIENT-KEY-EXCHANGE message with our guess g. The server decrypts g as a normal CLIENT-KEY-EXCHANGE message, and then checks the resulting plaintext for proper PKCS 1 padding. Since the decryption of g will not be properly formatted, the server and client will not compute the same master secret, and the client will ultimately receive an ALERT message from the server. The attacking client computes the time difference from sending g as the CLIENT-KEY-EXCHANGE message to receiving the response message from the server as the time to decrypt g. The client repeats this process for each value of of g and g_{hi} needed to calculate T_g and $T_{g_{hi}}$.

Our experiments are also relevant to trusted computing efforts such as NGSCB. One goal of NGSCB is to provide sealed storage. Sealed storage allows an application to encrypt data to disk using keys unavailable to the user. The timing attack shows that by asking NGSCB to decrypt data in sealed storage a user may learn the secret application key. Therefore, it is essential that the secure storage mechanism provided by projects such as NGSCB defend against this timing attack.

As mentioned in the introduction, RSA applications (and subsequently SSL applications using RSA for key exchange) using a hardware crypto accelerator are not vulnerable since most crypto accelerators implement defenses against the timing attack. Our attack applies to software based RSA implementations that do not defend against timing attacks as discussed in section 6.

5 Experiments

We performed a series of experiments to demonstrate the effectiveness of our attack on OpenSSL. In each case we show the factorization of the RSA modulus N is vulnerable. We show that a number of factors affect the efficiency of our timing attack.

Our experiments consisted of:

1. Test the effects of increasing the number of decryption requests, both for the same ciphertext and a neighborhood of ciphertexts.
2. Compare the effectiveness of the attack based upon different keys.
3. Compare the effectiveness of the attack based upon machine architecture and common compile-time optimizations.
4. Compare the effectiveness of the attack based upon source-based optimizations.
5. Compare inter-process vs. local network attacks.
6. Compare the effectiveness of the attack against two common SSL applications: an Apache web server with mod_SSL and stunnel.

The first four experiments were carried out inter-process via TCP, and directly characterize the vulnerability of OpenSSL's RSA decryption routine. The fifth experiment demonstrates our attack succeeds on the local network. The last experiment demonstrates our attack succeeds on the local network against common SSL-enabled applications.

5.1 Experiment Setup

Our attack was performed against OpenSSL 0.9.7, which does not blind RSA operations by default. All tests were run under RedHat Linux 7.3 on a 2.4 GHz Pentium 4 processor with 1 GB of RAM, using gcc 2.96 (RedHat). All keys were generated at random via OpenSSL's key generation routine.

For the first 5 experiments we implemented a simple TCP server that read an ASCII string, converted the string to OpenSSL's internal multi-precision representation, then performed the RSA decryption. The server returned 0 to signify the end of decryption. The TCP client measured the time from writing the ciphertext over the socket to receiving the reply.

Our timing attack requires a clock with fine resolution. We use the Pentium cycle counter on the attacking machine as such a clock, giving us a time resolution of 2.4 billion ticks per second. The cycle counter increments once per clock tick, regardless of the actual instruction issued. Thus, the decryption time is the cycle counter difference between sending the ciphertext to receiving the reply. The cycle counter is accessible via the "rdtsc" instruction, which returns the 64-bit cycle count since CPU initialization. The high 32 bits are returned into the EDX register, and the low 32 bits into the EAX register. As recommended in [7], we use the "cpuid" instruction to serialize the processor to prevent out-of-order execution from changing our timing measurements. Note that cpuid and rdtsc are only used by the attacking client, and that neither instruction is a privileged operation. Other architectures have a similar a counter, such as the UltraSparc %tick register.

OpenSSL generates RSA moduli $N = pq$ where $q < p$. In each case we target the smaller factor, q. Once q is known, the RSA modulus is factored and, consequently, the server's private key is exposed.

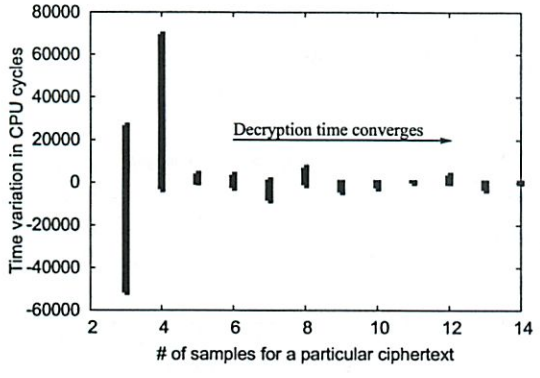
5.2 Experiment 1 - Number of Ciphertexts

This experiment explores the parameters that determine the number of queries needed to expose a single bit of an RSA factor. For any particular bit of q, the number of queries for guess g is determined by two parameters: neighborhood size and sample size.

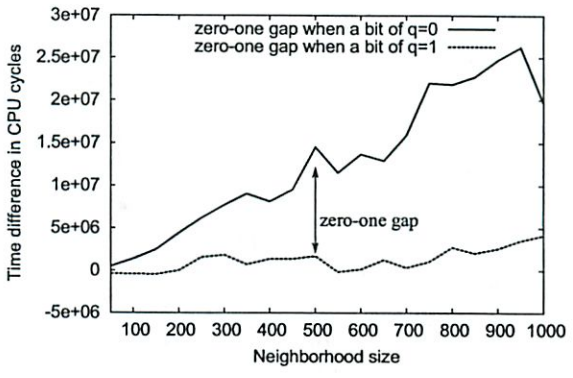
Neighborhood size. For every bit of q we measure the decryption time for a neighborhood of values $g, g+1, g+2, ..., g+n$. We denote this neighborhood size by n.

Sample size. For each value $g + i$ in a neighborhood we sample the decryption time multiple times and compute the mean decryption time. The number of times we query on each value $g + i$ is called the sample size and is denoted by s.

The total number of queries needed to compute T_g is then $s * n$.



(a) The time variance for decrypting a particular ciphertext decreases as we increase the number of samples taken.



(b) By increasing the neighborhood size we increase the zero-one gap between a bit of q that is 0 and a bit of q that is 1.

Figure 2: Parameters that affect the number of decryption queries of g needed to guess a bit of the RSA factor.

To overcome the effects of a multi-user environment, we repeatedly sample $g+k$ and use the median time value as the effective decryption time. Figure 2(a) shows the difference between median values as sample size increases. The number of samples required to reach a stable decryption time is surprising small, requiring only 5 samples to give a variation of under 20000 cycles (approximately 8 microseconds), well under that needed to perform a successful attack.

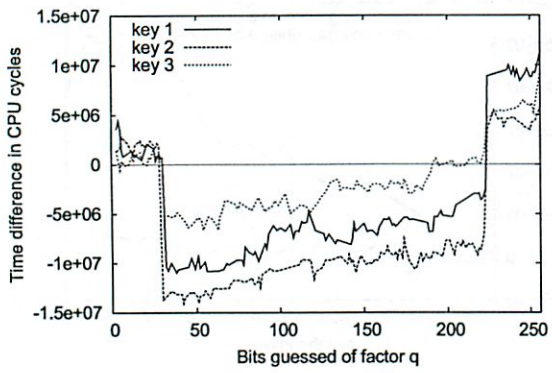
We call the gap between when a bit of q is 0 and 1 the *zero-one* gap. This gap is related to the difference $|T_g - T_{g_{hi}}|$, which we expect to be large when a bit of q is 0 and small otherwise. The larger the gap, the stronger the indicator that bit i is 0, and the smaller chance of error. Figure 2(b) shows that increasing the neighborhood size increases the size of the zero-one gap when a bit of q is 0, but is steady when a bit of q is 1.

The total number of queries to recover a factor is $2ns * \log_2 N/4$, where N is the RSA public modulus. Unless explicitly stated otherwise, we use a sample size of 7 and a neighborhood size of 400 on all subsequent experiments, resulting in 1433600 total queries. With these parameters a typical attack takes approximately 2 hours. In practice, an effective attack may need far fewer samples, as the neighborhood size can be adjusted dynamically to give a clear zero-one gap in the smallest number of queries.

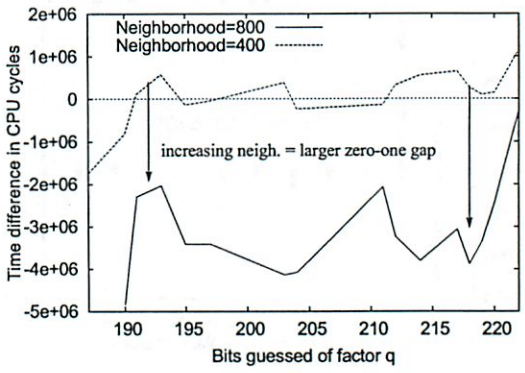
5.3 Experiment 2 - Different Keys

We attacked several 1024-bit keys, each randomly generated, to determine the ease of breaking different moduli. In each case we were able to recover the factorization of N. Figure 3(a) shows our results for 3 different keys. For clarity, we include only bits of q that are 0, as bits of q that are 1 are close to the x-axis. In all our figures the time difference $T_g - T_{g_{hi}}$ is the zero-one gap. When the zero-one gap for bit i is far from the x-axis we can correctly deduce that bit i is 0.

With all keys the zero-one gap is positive for about the first 32 bits due to Montgomery reductions, since both g and g_{hi} use Karatsuba multiplication. After bit 32, the difference between Karatsuba and normal multiplication dominate until overcome by the sheer size difference between $\log_2(g \bmod q) - \log_2(g_{hi} \bmod q)$. The size difference alters the zero-one gaps because as bits of q are guessed, g_{hi} becomes smaller while g remains $\approx \log_2 q$. The size difference counteracts the effects of Karatsuba vs. normal multiplication. Normally the resulting zero-one gap shift happens around multiples of 32 (224 for key 1, 191 for key 2 and 3), our machine word size. Thus, an attacker should be aware that the zero-one gap may flip signs when guessing bits that are around multiples of the machine word size.



(a) The zero-one gap $T_g - T_{g_{hi}}$ indicates that we can distinguish between bits that are 0 and 1 of the RSA factor q for 3 different randomly-generated keys. For clarity, bits of q that are 1 are omitted, as the x-axis can be used for reference for this case.



(b) When the neighborhood is 400, the zero-one gap is small for some bits in key 3, making it diffi cult to distinguish between the 0 and 1 bits of q. By increasing the neighborhood size to 800, the zero-one gap is increased and we can launch a successful attack.

Figure 3: Breaking 3 RSA Keys by looking at the zero-one gap time difference

As discussed previously we can increase the size of the neighborhood to increase $|T_g - T_{g_{hi}}|$, giving a stronger indicator. Figure 3(b) shows the effects of increasing the neighborhood size from 400 to 800 to increase the zero-one gap, resulting in a strong enough indicator to mount a successful attack on bits 190-220 of q in key 3.

The results of this experiment show that the factorization of each key is exposed by our timing attack by the zero-one gap created by the difference when a bit of q is 0 or 1. The zero-one gap can be increased by increasing the neighborhood size if hard-to-guess bits are encountered.

5.4 Experiment 3 - Architecture and Compile-Time Effects

In this experiment we show how the computer architecture and common compile-time optimizations can affect the zero-one gap in our attack. Previously, we have shown how algorithmically the number of extra Montgomery reductions and whether normal or Karatsuba multiplication is used results in a timing attack. However, the exact architecture on which decryption is performed can change the zero-one gap.

To show the effect of architecture on the timing attack, we begin by showing the total number of instructions retired agrees with our algorithmic analysis of OpenSSL's decryption routines. An instruction is retired when it completes and the results are written to the destination [8]. However, programs with similar retirement counts may have different execution profiles due to different run-time factors such as branch predictions, pipeline throughput, and the L1 and L2 cache behavior.

We show that minor changes in the code can change the timing attack in two programs: "regular" and "extra-inst". Both programs time local calls to the OpenSSL decryption routine, i.e. unlike other programs presented "regular" and "extra-inst" are not network clients attacking a network server. The "extra-inst" is identical to "regular" except 6 additional nop instructions inserted before timing decryptions. The nop's only change subsequent code offsets, including those in the linked OpenSSL library.

Table 1 shows the timing attack with both programs for two bits of q. Montgomery reductions cause a positive instruction retired difference for bit 30, as expected. The difference between Karatsuba and normal multiplication cause a negative instruction retired difference for bit 32, again as expected. However, the difference $T_g - T_{g_{hi}}$ does not follow the instructions retired difference. On bit 30, there is about a 4 million extra cycles difference between the "regular" and "extra-inst" programs, even though the instruction retired count decreases. For bit 32, the change is even more pronounced: the zero-one gap changes sign between the "normal" and "extra-inst" programs while the instructions retired are similar!

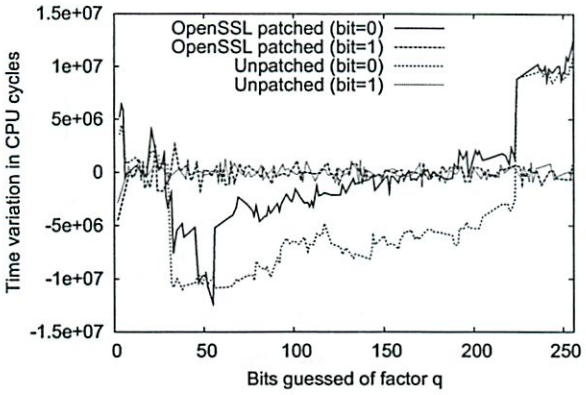


Figure 5: Minor source-based optimizations change the zero-one gap as well. As a consequence, code that doesn't appear initially vulnerable may become so as the source is patched.

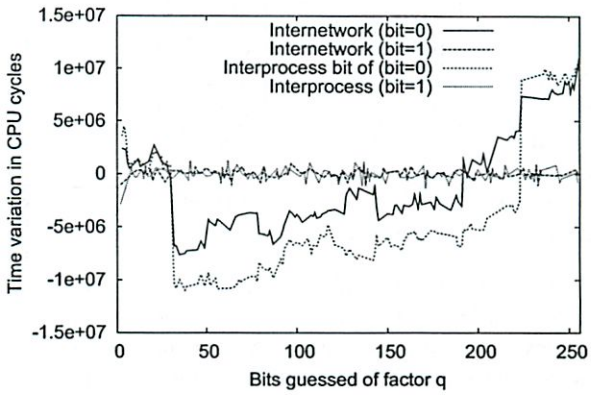


Figure 6: The timing attack succeeds over a local network. We contrast our results with the attack interprocess.

One conclusion we draw is that users of binary crypto libraries may find it hard to characterize their risk to our attack without complete understanding of the compile-time options and exact execution environment. Common flags such as enabling debugging support allow our attack to recover the factors of a 1024-bit modulus in about 1/3 million queries. We speculate that less complex architectures will be less affected by minor code changes, and have the zero-one gap as predicted by the OpenSSL algorithm analysis.

5.5 Experiment 4 - Source-based Optimizations

Source-based optimizations can also change the zero-one gap. RSA library developers may believe their code is not vulnerable to the timing attack based upon testing. However, subsequent patches may change the code profile resulting in a timing vulnerability. To show that minor source changes also affect our attack, we implemented a minor patch that improves the efficiency of the OpenSSL 0.9.7 CRT decryption check. Our patch has been accepted for future incorporation to OpenSSL (tracking ID 475).

After a CRT decryption, OpenSSL re-encrypts the result (mod N) and verifies the result is identical to the original ciphertext. This verification step prevents an incorrect CRT decryption from revealing the factors of the modulus [2]. By default, OpenSSL needlessly recalculates both Montgomery parameters R and R^{-1} mod N on every decryption. Our minor patch allows OpenSSL to cache both values between decryptions with the same key. Our patch does not affect any other aspect of the RSA decryption other than caching these values. Figure 5 shows the results of an attack both with and without the patch.

The zero-one gap is shifted because the resulting code will have a different execution profile, as discussed in the previous experiment. While our specific patch decreases the size of the zero-one gap, other patches may increase the zero-one gap. This shows the danger of assuming a specific application is not vulnerable due to timing attack tests, as even a small patch can change the run-time profile and either increase or decrease the zero-one gap. Developers should instead rely upon proper algorithmic defenses as discussed in section 6.

5.6 Experiment 5 - Interprocess vs. Local Network Attacks

To show that local network timing attacks are practical, we connected two computers via a 10/100 Mb Hawking switch, and compared the results of the attack interprocess vs. inter-network. Figure 6 shows that the network does not seriously diminish the effectiveness of the attack. The noise from the network is eliminated by repeated sampling, giving a similar zero-one gap to inter-process. We note that in our tests a zero-one gap of approximately 1 millisecond is sufficient to receive a strong indicator, enabling a successful attack. Thus, networks with less than 1ms of variance are vulnerable.

	$g - g_{hi}$ retired	$T_g - T_{g_{hi}}$ cycles
"regular" bit 30	4579248 (0.009%)	6323188 (0.057%)
"extra-inst" bit 30	7641653 (0.016%)	2392299 (0.022%)
"regular" bit 32	-14275879 (-0.029%)	-5429545 (-0.049%)
"extra-inst" bit 32	-13187257 (-0.027%)	1310809 (0.012%)

Table 1: Bit 30 of q for both "regular" and "extra-inst" (which has a few additional nop's) have a positive instructions retired difference due to Montgomery reductions. Similarly, bit 32 has a negative instruction difference due to normal vs. Karatsuba multiplication. However, the addition of a few nop instructions in the "extra-instr" program changes the timing profile, most notably for bit 32. The percentages given are the difference divided by either the total of instructions retired or cycles as appropriate.

Extensive profiling using Intel's VTune [6] shows no single cause for the timing differences. However, two of the most prevalent factors were the L1 and L2 cache behavior and the number of instructions speculatively executed incorrectly. For example, while the "regular" program suffers approximately 0.139% L1 and L2 cache misses per load from memory on average, "extra-inst" has approximately 0.151% L1 and L2 cache misses per load. Additionally, the "regular" program speculatively executed about 9 million micro-operations incorrectly. Since the timing difference detected in our attack is only about 0.05% of total execution time, we expect the runtime factors to heavily affect the zero-one gap. However, under normal circumstances some zero-one gap should be present due to the input data dependencies during decryption.

The total number of decryption queries required for a successful attack also depends upon how OpenSSL is compiled. The compile-time optimizations change both the number of instructions, and how efficiently instructions are executed on the hardware. To test the effects of compile-time optimizations, we compiled OpenSSL three different ways:

- **Optimized** (-O3 -fomit-frame-pointer -mcpu=pentium): The default OpenSSL flags for Intel. -O3 is the optimization level, -fomit-frame-pointer omits the frame pointer, thus freeing up an extra register, and -mcpu=pentium enables more sophisticated resource scheduling.

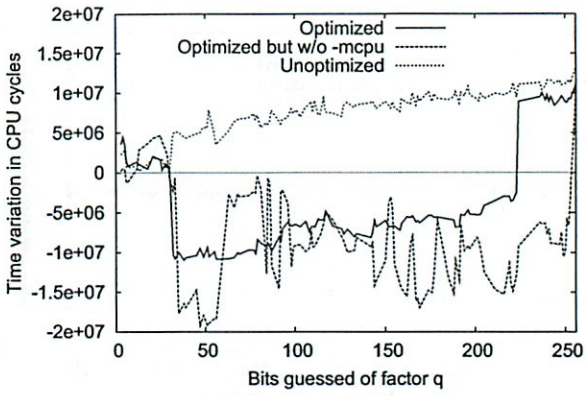


Figure 4: Different compile-time flags can shift the zero-one gap by changing the resulting code and how efficiently it can be executed.

- **No Pentium flag** (-O3 -fomit-frame-pointer): The same as the above, but without -mcpu sophisticated resource scheduling is not done, and an i386 architecture is assumed.
- **Unoptimized** (-g): Enable debugging support.

Each different compile-time optimization changed the zero-one gap. Figure 4 compares the results of each test. For readability, we only show the difference $T_g - T_{g_{hi}}$ when bit i of q is 0 ($g < q < g_{hi}$). The case where bit $i = 1$ shows little variance based upon the optimizations, and the x-axis can be used for reference.

Recall we expected Montgomery reductions to dominate when guessing the first 32 bits (with a positive zero-one gap), switching to Karatsuba vs. normal multiplication (with a negative zero-one gap) thereafter. Surprisingly, the unoptimized OpenSSL is unaffected by the Karatsuba vs. normal multiplication. Another surprising difference is the zero-one gap is more erratic when the -mcpu flag is omitted.

In these tests we again made about 1.4 million decryption queries. We note that without optimizations (-g), separate tests allowed us to recover the factorization with less than 359000 queries. This number could be reduced further by dynamically reducing the neighborhood size as bits of q are learned. Also, our tests of OpenSSL 0.9.6g were similar to the results of 0.9.7, suggesting previous versions of OpenSSL are also vulnerable.

to be decrypted. x is then decrypted as normal, followed by division by r, i.e. $x^e/r \bmod N$. Since r is random, x is random and timing the decryption should not reveal information about the key. Note that r should be a new random number for every decryption. According to [17] the performance penalty is $2\% - 10\%$, depending upon implementation. Netscape/Mozilla's NSS library uses blinding. Blinding is available in OpenSSL, but not enabled by default in versions prior to 0.9.7b. Figure 8 shows that blinding in OpenSSL 0.9.7b defeats our attack. We hope this paper demonstrates the necessity of enabling this defense.

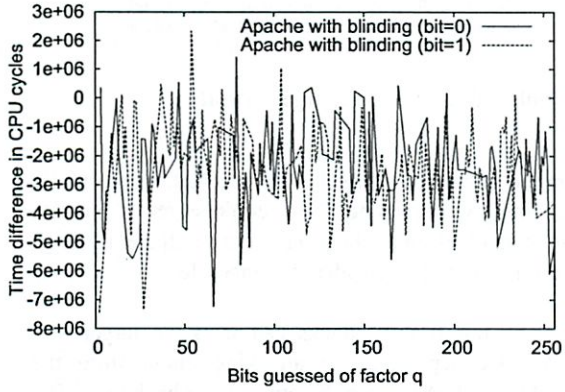


Figure 8: Our attack against Apache+mod_SSL using OpenSSL 0.9.7b is defeated because blinding is enabled by default.

Two other possible defenses are suggested often, but are a second choice to blinding. The first is to try and make all RSA decryptions not dependent upon the input ciphertext. In OpenSSL one would use only one multiplication routine and always carry out the extra reduction in Montgomery's algorithm, as proposed by Schindler in [18]. If an extra reduction is not needed, we carry out a "dummy" extra reduction and do not use the result. Karatsuba multiplication can always be used by calculating $c \bmod p_i * 2^m$, where c is the ciphertext, p_i is one of the RSA factors, and $m = \log_2 p_i - \log_2 (c \bmod p_i)$. After decryption, the result is divided by $2^{md} \bmod q$ to yield the plaintext. We believe it is harder to create and maintain code where the decryption time is not dependent upon the ciphertext. For example, since the result is never used from a dummy extra reduction during Montgomery reductions, it may inadvertently be optimized away by the compiler.

Another alternative is to require all RSA computations to be quantized, i.e. always take a multiple of some predefined time quantum. Matt Blaze's quantize library [1] is an example of this approach. Note that *all* decryptions must take the maximum time of *any* decryption, otherwise, timing information can still be used to leak information about the secret key.

Currently, the preferred method for protecting against timing attacks is to use RSA blinding. The immediate drawbacks to this solution is that a good source of randomness is needed to prevent attacks on the blinding factor, as well as the small performance degradation. In OpenSSL, neither drawback appears to be a significant problem.

7 Conclusion

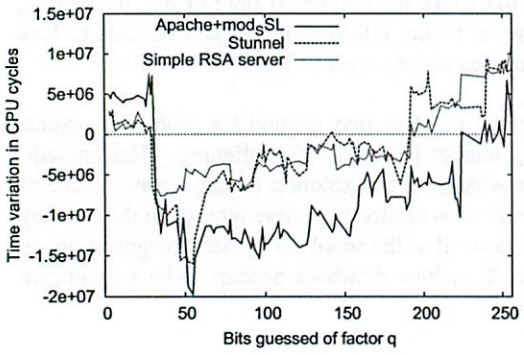
We devised and implemented a timing attack against OpenSSL — a library commonly used in web servers and other SSL applications. Our experiments show that, counter to current belief, the timing attack is effective when carried out between machines separated by multiple routers. Similarly, the timing attack is effective between two processes on the same machine and two Virtual Machines on the same computer. As a result of this work, several crypto libraries, including OpenSSL, now implement blinding by default as described in the previous section.

8 Acknowledgments

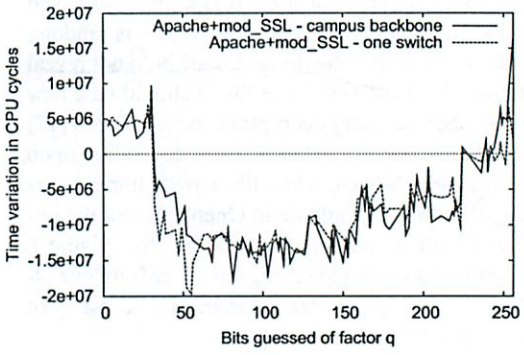
This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481 and the Packard Foundation. We thank the reviewers, Dr. Monica Lam, Ramesh Chandra, Constantine Sapuntzakis, Wei Dai, Art Manion and CERT/CC, and Dr. Werner Schindler for their comments while preparing this paper. We also thank Nelson Bolyard, Geoff Thorpe, Ben Laurie, Dr. Stephen Henson, Richard Levitte, and the rest of the OpenSSL, mod_SSL, and stunnel development teams for their help in preparing patches to enable and use RSA blinding.

References

[1] Matt Blaze. Quantize wrapper library. `http://islab.oregonstate.edu/documents/People/blaze`.



(a) The zero-one gaps when attacking Apache+mod_SSL and stunnel separated by one switch.



(b) The zero-one gap when attacking Apache+mod_SSL separated by several routers and a network backbone.

Figure 7: Applications using OpenSSL 0.9.7 are vulnerable, even on a large network.

Inter-network attacks allow an attacker to also take advantage of faster CPU speeds for increasing the accuracy of timing measurements. Consider machine 1 with a slower CPU than machine 2. Then if machine 2 attacks machine 1, the faster clock cycle allows for finer grained measurements of the decryption time on machine 1. Finer grained measurements should result in fewer queries for the attacker, as the zero-one gap will be more distinct.

5.7 Experiment 6 - Attacking SSL Applications on the Local Network

We show that OpenSSL applications are vulnerable to our attack from the network. We compiled Apache 1.3.27 + mod_SSL 2.8.12 and stunnel 4.04 per the respective "INSTALL" files accompanying the software. Apache+mod_SSL is a commonly used secure web server. stunnel allows TCP/IP connections to be tunneled through SSL.

We begin by showing servers connected by a single switch are vulnerable to our attack. This scenario is relevant when the attacker has access to a machine near the OpenSSL-based server. Figure 7(a) shows the result of attacking stunnel and mod_SSL where the attacking client is separated by a single switch. For reference, we also include the results for a similar attack against the simple RSA decryption server from the previous experiments.

Interestingly, the zero-one gap is larger for Apache+mod_SSL than either the simple RSA decryption server or stunnel. As a result, successfully attacking Apache+mod_SSL requires fewer queries than stunnel. Both applications have a sufficiently large zero-one gap to be considered vulnerable.

To show our timing attacks can work on larger networks, we separated the attacking client from the Apache+mod_SSL server by our campus backbone. The webserver was hosted in a separate building about a half mile away, separated by three routers and a number of switches on the network backbone. Figure 7(b) shows the effectiveness of our attack against Apache+mod_SSL on this larger LAN, contrasted with our previous experiment where the attacking client and server are separated by only one switch.

This experiment highlights the difficulty in determining the minimum number of queries for a successful attack. Even though both stunnel and mod_SSL use the exact same OpenSSL libraries and use the same parameters for negotiating the SSL handshake, the run-time differences result in different zero-one gaps. More importantly, our attack works even when the attacking client and application are separated by a large network.

6 Defenses

We discuss three possible defenses. The most widely accepted defense against timing attacks is to perform RSA blinding. The RSA blinding operation calculates $x = r^e g \bmod N$ before decryption, where r is random, e is the RSA encryption exponent, and g is the ciphertext

[2] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. *Lecture Notes in Computer Science*, 1233:37–51, 1997.

[3] D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology*, 10:233–260, 1997.

[4] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestre, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *CARDIS*, pages 167–182, 1998.

[5] Peter Gutmann. Cryptlib. `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`.

[6] Intel. Vtune performance analyzer for linux v1.1. `http://www.intel.com/software/products/vtune`.

[7] Intel. Using the RDTSC instruction for performance monitoring. Technical report, 1997.

[8] Intel. Ia-32 intel architecture optimization reference manual. Technical Report 248966-008, 2003.

[9] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis: Leaking secrets. In *Crypto 99*, pages 388–397, 1999.

[10] Paul Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. *Advances in Cryptology*, pages 104–113, 1996.

[11] Alfred Menezes, Paul Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.

[12] mod_SSL Project. mod_ssl. `http://www.modssl.org`.

[13] Peter Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[14] GNU Project. libgcrypt. `http://www.gnu.org/directory/security/libgcrypt.html`.

[15] OpenSSL Project. Openssl. `http://www.openssl.org`.

[16] Rao, Josyula, Rohatgi, and Pankaj. Empowering side-channel attacks. Technical Report 2001/037, 2001.

[17] RSA Press Release. `http://www.otn.net/onthenet/rsaqa.htm`, 1995.

[18] Werner Schindler. A timing attack against RSA with the chinese remainder theorem. In *CHES 2000*, pages 109–124, 2000.

[19] Werner Schindler. A combined timing and power attack. *Lecture Notes in Computer Science*, 2274:263–279, 2002.

[20] Werner Schindler. Optimized timing attacks against public key cryptosystems. *Statistics and Decisions*, 20:191–210, 2002.

[21] Werner Schindler, Franois Koeune, and Jean-Jacques Quisquater. Improving divide and conquer attacks against cryptosystems by better error detection/correction strategies. *Lecture Notes in Computer Science*, 2260:245–267, 2001.

[22] Werner Schindler, Franois Koeune, and Jean-Jacques Quisquater. Unleashing the full power of timing attack. Technical Report CG-2001/3, 2001.

[23] stunnel Project. stunnel. `http://www.stunnel.org`.

6.858: Computer Systems Security Fall 2012

How in all world able to do via network?

Home

General information

Schedule

Reference materials

Piazza discussion

Submission

2011 class materials

Paper Reading Questions

For each paper, your assignment is two-fold. By the start of lecture:

- Submit your answer for each lecture's paper question via the submission web site in a file named `lecn.txt`, and
- E-mail your own question about the paper (e.g., what you find most confusing about the paper or the paper's general context/problem) to 6.858-q@pdos.csail.mit.edu. You cannot use the question below. To the extent possible, during lecture we will try to answer questions submitted by the evening before.

Lecture 17

What are some other situations where an adversary may be able to learn confidential information by timing certain operations? Propose some ideas for how an application developer might mitigate such vulnerabilities.

On ssh password system
time of transmission
length of message
keyboard typing patterns
microphone

What else? packet contents
voice calls
process time/energy consumption

Questions or comments regarding 6.858? Send e-mail to the course staff at 6.858-staff@pdos.csail.mit.edu.

Top // **6.858 home** // *Last updated Friday, 12-Oct-2012 23:31:47 EDT*

11/18

Paper Question 17

Michael Plasmeier

This question made me think about the password systems we are analyzing as part of our final project.

One could measure the time of transmission of the message, or observe the length of the message.

One could possibly analyze the timing of typing by building inferences about the time gaps between letters by having the user type training data.

I think I remember seeing someone claim to do this for VOIP calls.

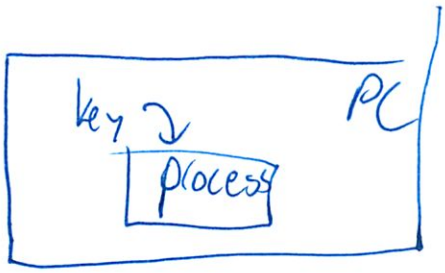
Tracking energy usage of a processor is closely related. This is particularly powerful for simple, specialized hardware.

6.858

11/14

Side Channel Attacks

Assuming too much about abstraction layer

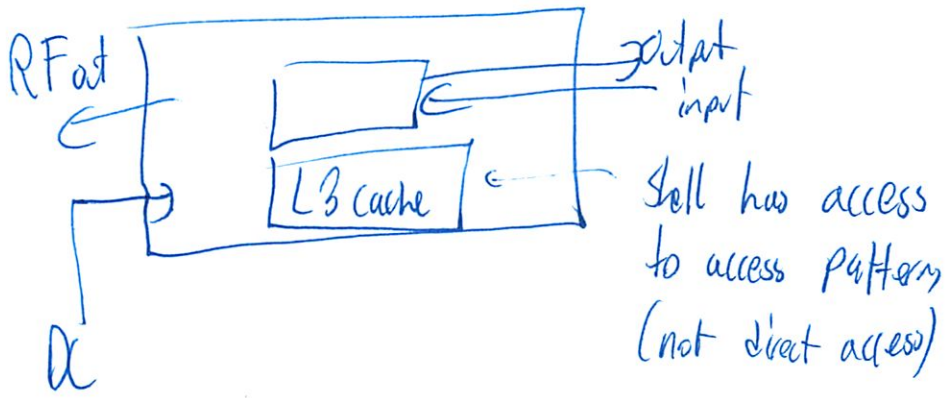


← is that containment well?

Could listen to RF emissions from CRTs

100 ft away

much harder w/ LCD



Hard to defend

esp in general case ...

(2)

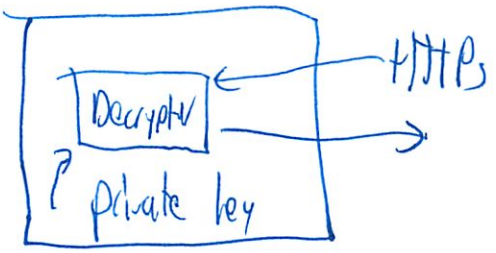
Can listen to sounds a dot matrix printer makes

Keys are very valuable

So people talk about it the most

Could also know how many friends someone has.
based on how long it takes req to process

Here



For network → average over hundreds of times

But very subtile

So almost impossible

(3)

Especially used in smart cards

RSA

Implementation Details Matter here

Pick random primes $p, q \approx 512$ bits

$n = p \cdot q$ — public (n, e)

$e = 65537$

$d = (512 \text{ bit value})$ — private (n, d)

$m \to c = m^e \bmod n$

$c \to m = c^d \bmod n$

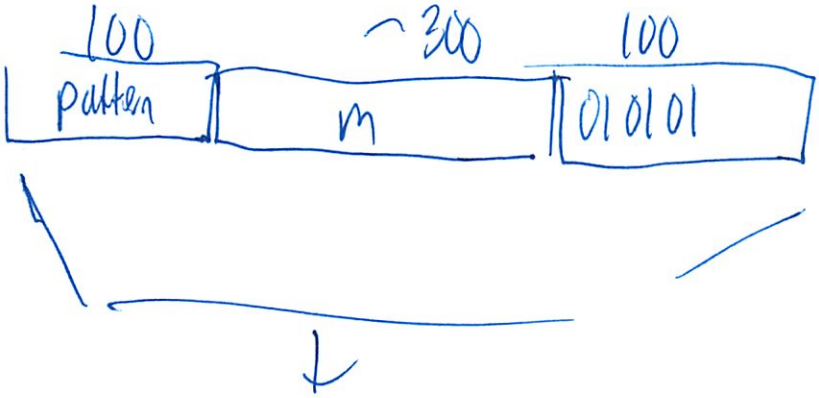
Lots of subtle things in implementation of RSA

So don't rely on these notes!

(4)

$$\left.\begin{aligned} c_0 &= m_0^e \bmod n \\ c_1 &= m_1^e \bmod n \end{aligned}\right\} \quad c_0 \cdot c_1 = (m_0 \cdot m_1)^e \bmod n$$

RSA-OAEP



lots of subtle padding
picking pq
e is 100...01 in binary
etc
not in original paper
but needed to protect the abstraction

(5)

So

$$(1024\text{ bit})^{(1024\text{ bit})} \bmod (1024\text{ bit})$$

How to do that fast?

Chinese Remainder Theorm

$$\left.\begin{array}{l} X: \quad X = a_1 \bmod p \\ \qquad\quad X = a_2 \bmod p \end{array}\right\} X = X_0 \bmod (pq)$$

To compute $c^d \bmod n$

$$\text{Compute} \quad \left.\begin{array}{l} c^d \bmod p \to m_0 \\ c^d \bmod q \to m_1 \end{array}\right) \; m \bmod (p \cdot q) = n$$

Many people add (n, d, p, q) to private key to make it faster

(6)

512 bit ops ($C^d \bmod p \to m_0$

512 bit ops ($C^d \bmod q \to m_1$

So 2x performance ↑

Since quadratic factors

---

2. $C^d \bmod p$

Don't multiply C over + over

instead <u>repeated squaring</u>

Can chop off one bit of d at a time

$C^{2x} = (C^x)^2$

$C^{2x+1} = C(C^x)^2$

squarings $|d|$

of multiplications by C: # of bits in d

(1)

So ~5/2 squarings
~25% multp by C

So faster

2b. Refinement: sliding windows

Can reduce # of multiplications

$C^{4x} = (C^x)^4$

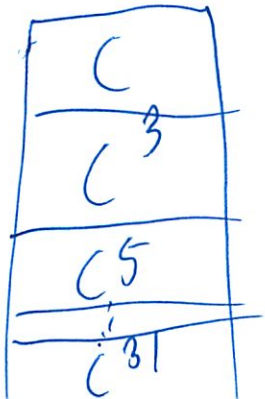
$C^{4x+1} = C(C^x)^4$

$C^{4x+2} = C^2(C^x)^4$

$C^{4x+3} = C^3(C^x)^4$

Can chop off 2 bits at atime

Can precompute



(not following ...)

(8)

Need to do a mod c reduction

C^2 $C \cdot C$ mod p

C ~512

C^2 ~1024

$C \cdot C^2$ ~1536

So might as well do the mod p

But still costly

Montgomery rep

regular

C ⇝

: square + multiply

C^d ←

Montgomery rep

CR

↓ mul spr

$C^d R$

3 solid lines less costly than 1 dotted line

(9)

We multiply everything by R

$a \bmod p \to aR \bmod p$ (~512 bits)

$b \bmod p \to bR \bmod p$ (~512 bits) (a)

want

$C = a \cdot b \to cR \bmod p$

~512 bits

So

$(aR) \cdot (bR) = abR^2$

So must $\frac{(aR)(bR)}{R}$ ~1024 bits

Super fast! →

But that seems like more work

But hasn't grown so fast

So no need to mod

(ii)

Make R a pow of 2

$R = 2^{512}$

Smaller example

$R = 2^4 = 10000$

$abR^2 = 26 = 11010$

$p = 7 = 111$

$abR^2 = abR^2 + p \bmod p$

$= abR^2 + qx \bmod p$

qx int

So add multiples of p

until low bits are 0

11010

↓ rest

10000

(11)

So $x = 2$ for $2p$

$$
\begin{array}{r}
11010 \\
\quad 1110 \\
\hline
101000 \\
111 \qquad\qquad 8p \\
\hline
1100000,
\end{array}
$$

So $abR^2 + 2p + 8p =$

$$\frac{abR^2}{R} = 110$$

~~Take ℓ and raise to $t-1$~~

$$x = \ell(-q^{-1}) \bmod R$$

(inverse)

$$\Downarrow$$

$abR^2 + x \cdot p$ will have low 0 bits

So multiplication is way faster!

if still > p, then sub p

a ~ 512 bit # = 16 × [32] [32] ... [32]

b ~ 512 bit # = 16 × [32] [32] ... [32]

Straight forward to multiply

↳ n^2 niave plan

But clever trick

Karatsuba - Russia '30s + '40s

$n^{1.585}$ time

for $n = 16$ $\quad n^2 = 256$

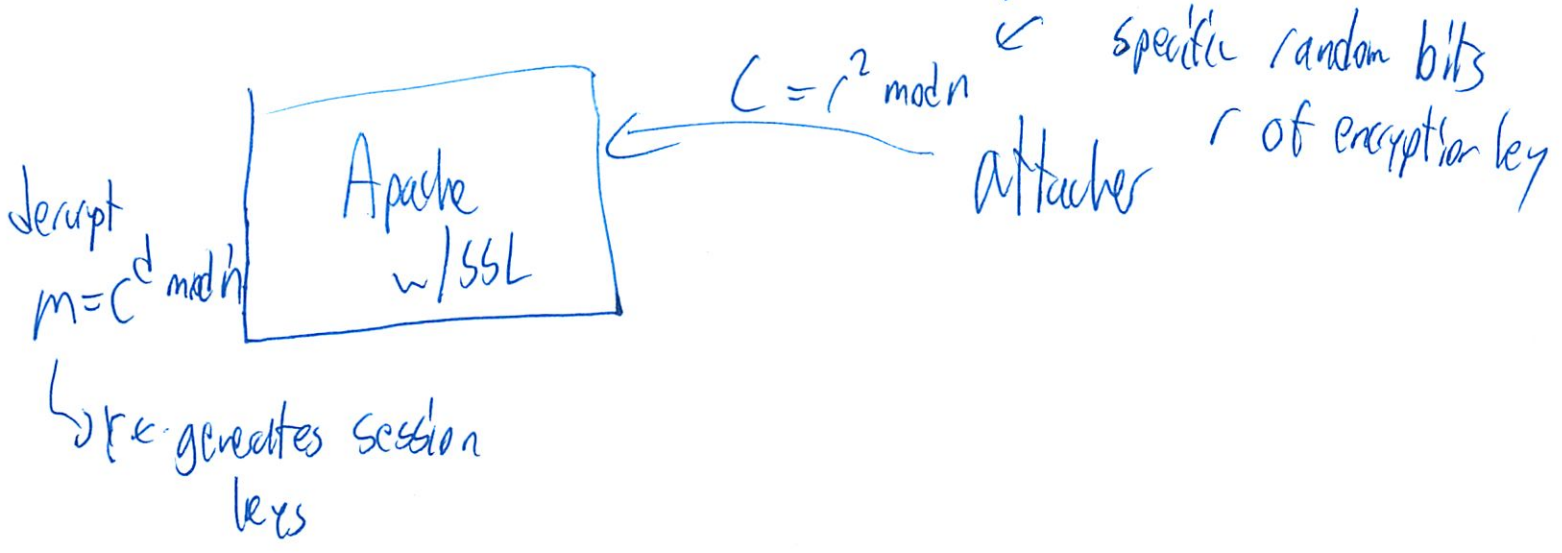
$n^{1.585} \approx 81$

But only works if those things are = length.

(12)

But must be of = lenght

So how does this all fit together?



measures time for server to decrypt correctly

If wrong → server sends error + closes connection

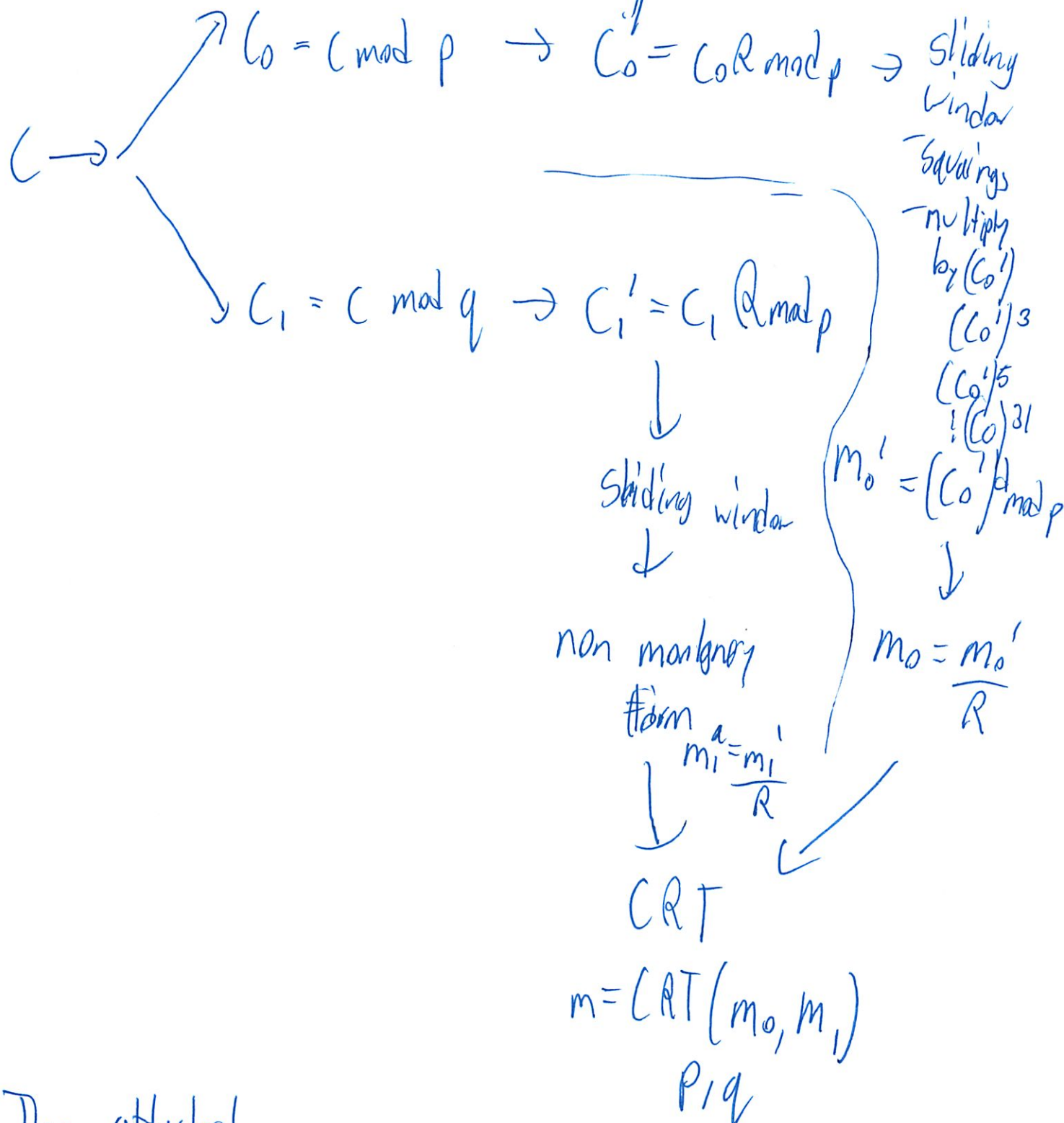
We can track that

Q Could we add a timeout/wait

Yes if could detect, add an avg amt of randomness

But assures specific attack strategy

(13) Break up into 2 pipelines



They attacked

- Squaring
  - extra reduction (-p)
- multiply $(C_0')(C_0')^3, (C_0')^5, \ldots (C_0')^{31}$
  - multiply karatsuba vs n^2

(14)

Figure out how often extra reductions happen

$$P[\text{extra reduction}] \approx \frac{c_0 \bmod p}{2R}$$

bigger c_0 is $\Rightarrow$ the greater chance you overshoot

---

will try to guess one value at a time

If can figure out p, q you can get d

They start guessing bits of p, q

Guessed k bits of p

$p = p_1 p_2 p_3 \cdots p_k \underbrace{\quad ? \quad}$

must figure out next bit

could be 1 or 0

Send $g^k \bmod n$

$g_{low} =$ || 000000 ...

$g_{hi} =$ || 11000 ...

(15)

Want to know time $g_{low}^{d} \bmod p$

$g_{high}^{d} \bmod p$

So how does this correlate?

If g below real value p

$(c')^{d} \bmod p$ will be large

So lots of extra reductions

but uses Karatsuba likely

If g above real value p

$(c')^{d} \bmod p$ will be low

few reductions

but uses n^2 likely

[redacted]

So measure how long take

(16)

if $t_{hi} = t_{lo}$

we know

both on same side of p

if $t_{hi} \neq t_{lo}$

p is b/w the two

then try the next bits

Can also factor out some bits

Only top 512 bits of p

figure out the rest the other way

512-bit

(17)

Each request ~ 5 milliseconds

Δ 10 microsecond difference

So average at requests to avoid noise

1. Use of network
   So send same guess 7 times
   Take median
      "Prof: Why median?
      Why not min?

2. Not many multiplications
   Want C_0^1
   But could be multiplied by $(C_0^1)^3$, $(C_0^1)^5$...
   So that could swamp the noise
   So send $g, g+1 \cdots g+399$

(18)

So if c_0 is prime leads to some small value

Exponents change all the time

So time of sliding window averages out

But high stays fine slow

Q: Identical hw?

They assume that

Could turn away repitions

(I didn't get much of this lecture - would need to recall & think about it - lots of subtlety)

Quantize (missed)

- Could do to time blocking
- Very much reduces info
- but subject to thought measurement
  - requests/sec servicing

(19)

would have to burn up CPU

But Intel chips also have AES built in HW carefully designed not to leak

11/19

Side-channel attacks on RSA

Side channel attacks: example setting

- A server (e.g., Apache) has an RSA private key.
- Server uses RSA private key (e.g., decrypt message from client).
- Something about the server's computation is leaked to the client.
- Many information leaks have been looked at:
  - How long it takes to decrypt
  - How decryption affects shared resources (cache, TLB, branch predictor)
  - Emissions from the CPU itself (RF, audio, power consumption, ..)
- Side-channel attacks don't have to be crypto-related.
  - E.g., operation time relates to which character of password was incorrect.
  - Or time related to how many common friends you + some user have on Facebook.
- Adversary can analyze information leaks, use it to reconstruct private key.
  - Currently, side-channel attacks on systems described in the paper are rare.
    - E.g., Apache web server running on some Internet-connected machine.
    - Often some other vulnerability exists and is easier to exploit.
    - Slowly becoming a bigger concern: new side-channels (VMs), better attacks.
  - Side-channel attacks are more commonly used to attack trusted/embedded hw.
    - E.g., chip running cryptographic operations on a smartcard.
    - Often these have a small attack surface, not many other ways to get in.
    - As paper mentions, some crypto coprocessors designed to avoid this attack.
- To understand how this works, first let's look at some internals of RSA..

RSA: high level plan

- Pick two random primes, p and q. Let n = p*q.
- A common key length, i.e., |n| or |d|, is 1024 or 2048 bits today.
- Euler's function phi(n): number of elements of Z_n^* relatively prime to n.
  - Theorem [no proof here]: $a^{\phi(n)} = 1 \bmod n$, for all a and n.
- So, how to encrypt and decrypt?
  - Pick two exponents d and e, such that $m^{e*d} = m \pmod n$, which means $e*d = 1 \bmod \phi(n)$.
  - Encryption will be $c = m^e \pmod n$; decryption will be $m = c^d \pmod n$.
- How to get such e and d?
  - For n=pq, phi(n) = (p-1)(q-1).
  - Easy to compute d=1/e, if we know phi(n).
  - In practice, pick small e (e.g., 65537), to make encryption fast.
- Public key is (n, e).
- Private key is, in principle, (n, d).
  - Note: p and q must be kept secret!
  - Otherwise, adversary can compute d from e, as we did above.
  - Knowing p and q also turns out to be helpful for fast decryption.
  - So, in practice, private key includes (p, q) as well.

RSA is tricky to use "securely" -- be careful if using RSA directly!

- Ciphertexts are multiplicative
  - $E(a)*E(b) = a^e * b^e = (ab)^e$.
  - Can allow adversary to manipulate encryptions, generate new ones.
- RSA is deterministic
  - Encrypting the same plaintext will generate the same ciphertext each time.
  - Adversary can tell when the same thing is being re-encrypted.
- Typically solved by "padding" messages before encryption [ref: OAEP]
  - Take plaintext message bits, add padding bits before and after plaintext.
  - Encrypt the combined bits (must be less than |n| bits total).
  - Padding includes randomness, as well as fixed bit patterns.
  - Helps detect tampering (e.g. ciphertext multiplication).

How to implement RSA?

- Key problem: fast modular exponentiation.
  - In general, quadratic complexity.
- Multiplying two 1024-bit numbers is slow.

```
  Computing the modulus for 1024-bit numbers is slow (1024-bit divison).

Optimization 1: Chinese Remainder Theorem (CRT).
  Recall what the CRT says:
    if x==a1 (mod p) and x==a2 (mod q), where p and q are relatively prime,
    then there's a unique solution x==a (mod pq).
    [ and, there's an efficient algorithm for computing a ]
  Suppose we want to compute m = c^d (mod pq).
  Can compute m1 = c^d (mod p), and m2 = c^d (mod q).
  Then use CRT to compute m = c^d (mod n) from m1, m2; it's unique and fast.
  Computing m1 (or m2) is 4x faster than computing m directly (quadratic).
  Computing m from m1 and m2 using CRT is ~negligible in comparison.
  So, roughly a 2x speedup.

Optimization 2: Repeated squaring and Sliding windows.
  Naive approach to computing c^d: multiply c by itself, d times.
  Better approach, called repeated squaring:
    c^(2x)   = (c^x)^2
    c^(2x+1) = (c^x)^2 * c
    To compute c^d, first compute c^(floor(d/2)), then use above for c^d.
    Recursively apply until the computation hits c^0 = 1.
    Number of squarings:          |d|
    Number of multiplications: number of 1 bits in d
  Better yet (sometimes), called sliding window:
    c^(2x)    = (c^x)^2
    c^(32x+1) = (c^x)^32 * c
    c^(32x+3) = (c^x)^32 * c^3
      ...
    c^(32x+z) = (c^x)^32 * c^z, generally [where z<31]
    Can pre-compute a table of all necessary c^z powers, store in memory.
    The choice of power-of-2 constant (e.g., 32) depends on usage.
      Costs: extra memory, extra time to pre-compute powers ahead of time.
    Note: only pre-compute odd powers of c (use first rule for even).
    OpenSSL uses 32 (table with 16 pre-computed entries).

Optimization 3: Montgomery representation.
  Reducing mod p each time (after square or multiply) is expensive.
    Typical implementation: do long division, find remainder.
    Hard to avoid reduction: otherwise, value grows exponentially.
  Idea (by Peter Montgomery): do computations in another representation.
    Shift the base (e.g., c) into different representation upfront.
    Perform modular operations in this representation (will be cheaper).
    Shift numbers back into original representation when done.
    Ideally, savings from reductions outweigh cost of shifting.
  Montgomery representation: multiply everything by some factor R.
    a mod q        <-> aR mod q
    b mod q        <-> bR mod q
    c = a*b mod q  <-> cR mod q = (aR * bR)/R mod q
    Each mul (or sqr) in Montgomery-space requires division by R.
  Why is modular multiplication cheaper in montgomery rep?
    Choose R so division by R is easy: R = 2^|q| (2^512 for 1024-bit keys).
    Because we divide by R, we will often not need to do mod q.
        |aR|          = |q|
        |bR|          = |q|
        |aR * bR|     = 2|q|
        |aR * bR / R| = |q|
    How do we divide by R cheaply?  Only works if lower bits are zero.
    Observation: since we care about value mod q, multiples of q don't matter.
    Trick: add multiples of q to the number being divided by R, make low bits 0.
      For example, suppose R=2^4 (10000), q=7 (111), divide x=26 (11010) by R.
        x+2q    = (binary)  101000
        x+2q+8q = (binary) 1100000
      Now, can easily divide by R: result is binary 110 (or 6).
```

```
      Generally, always possible:
        Low bit of q is 1 (q is prime), so can "shoot down" any bits.
        To "shoot down" bit k, add 2^k * q
        To shoot down low-order bits l, add q*(l*(-q^-1) mod R)
        Then, dividing by R means simply discarding low zero bits.
    One remaining problem: result will be < R, but might be > q.
      If the result happens to be greater than q, need to subtract q.
      This is called the "extra reduction".
      When computing x^d mod q, Pr[extra reduction] = (x mod q) / 2R.
        Here, x is assumed to be already in Montgomery form.
        Intuition: as we multiply bigger numbers, will overflow more often.

Optimization 4: Efficient multiplication.
  How to multiply 512-bit numbers?
  Representation: break up into 32-bit values (or whatever hardware supports).
  Naive approach: pair-wise multiplication of all 32-bit components.
    Same as if you were doing digit-wise multiplication of numbers on paper.
    Requires O(nm) time if two numbers have n and m components respectively.
    O(n^2) if the two numbers are close.
  Karatsuba multiplication: if both numbers have same number of components.
    O(n^log_3(2)) = O(n^1.585) time.
    Meaningfully faster (no hidden big constants)
      For 1024-bit keys, "n" here is 16 (512/32).
      n^2     = 256
      n^1.585 = 81

How does SSL use RSA?
  Server's SSL certificate contains public key.
  Server must use private key to prove its identity.
  Client sends random bits to server, encrypted with server's public key.
  Server decrypts client's message, uses these bits to generate session key.
    In reality, server also verifies message padding.
    However, can still measure time until server responds in some way.

Figure of decryption pipeline on the server:

                 CRT                To Montgomery                Modular exp
    -->  c_0 = c mod q  -->  c'_0 = c_0*R mod q  -->  m'_0 = (c'_0)^d mod q
   /
  /                                                 Use sliding window for bits
 /                                                    of the exponent d

c                                                   Karatsuba if c'_0 and q have
                                                      same number of 32-bit parts
 \
  \                                                 Extra reductions proportional
   \                                                  to ((c'_0)^z mod q) / 2R;
    -->  ...                                          z comes from sliding window

  Then, compute m_0 = m'_0/R mod q.
  Then, combine m_0 and m_1 using CRT to get m.
  Then verify padding in m.
  Finally, use payload in some way (SSL, etc).

Setup for the attack described in Brumley's paper.
  Victim Apache HTTPS web server using OpenSSL, has private key in memory.
  Connected to Stanford's campus network.
  Adversary controls some client machine on campus network.
  Adversary sends specially-constructed ciphertext in msg to server.
    Server decrypts ciphertext, finds garbage padding, returns an error.
    Client measures response time to get error message.
    Uses the response time to guess bits of q.
  Overall response time is on the order of 5 msec.
```

```
    Time difference between requests can be around 10 usec.
  What causes time variations?  Karatsuba vs normal; extra reductions.
  Once guessed enough bits of q, can factor n=p*q, compute d from e.
  About 1M queries seem enough to obtain 512-bit p and q for 1024-bit key.
    Only need to guess the top 256 bits of p and q, then use another algorithm.

Attack from Brumley's paper.
  Let q = q_0 q_1 .. q_N, where N = |q| (say, 512 bits for 1024-bit keys).
  Assume we know some number j of high-order bits of q (q_0 through q_j).
  Construct two approximations of q, guessing q_{j+1} is either 0 or 1:
    g    = q_0 q_1 .. q_j 0 0 0 .. 0
    g_hi = q_0 q_1 .. q_j 1 0 0 .. 0
  Get the server to perform modular exponentiation (g^d) for both guesses.
    We know g is necessarily less than q.
    If g and g_hi are both less than q, time taken shouldn't change much.
    If g_hi is greater than q, time taken might change noticeably.
      g_hi mod q is small.
      Less time: fewer extra reductions in Montgomery.
      More time: switch from Karatsuba to normal multiplication.
    Knowing the time taken can tell us if 0 or 1 was the right guess.
  How to get the server to perform modular exponentiation on our guess?
    Send our guess as if it were the encryption of randomness to server.
    One snag: server will convert our message to Montgomery form.
    Since Montgomery's R is known, send (g/R mod n) as message to server.
  How do we know if the time difference should be positive or negative?
    Paper seems to suggest it doesn't matter: just look for large diff.
    Figure 3a shows the measured time differences for each bit's guess.
    Karatsuba vs normal multiplication happens at 32-bit boundaries.
    First 32 bits: extra reductions dominate.
    Next bits: Karatsuba vs normal multiplication dominates.
    At some point, extra reductions start dominating again.
  How does the paper get accurate measurements?
    Client machine uses processor's timestamp counter (rdtsc on x86).
    Measure several times, take the median value.
      Not clear why median; min seems like it would be the true compute time.
    One snag: relatively few multiplications by g, due to sliding windows.
    Solution: get more multiplications by values close to g (+ same for g_hi).
    Specifically, probe a "neighborhood" of g (g, g+1, .., g+400).
  Why probe a 400-value neighborhood of g instead of measuring g 400 times?
    Consider the kinds of noise we are trying to deal with.
    1. Noise unrelated to computation (e.g. interrupts, network latency).
       This might go away when we measure the same thing many times.
    2. "Noise" related to computation.
       E.g., multiplying by g^3 and g_hi^3 in sliding window takes diff time.
       Repeated measurements will return the same value.
       Will not help determine whether mul by g or g_hi has more reductions.
    Neighborhood values average out 2nd kind of noise.
    Since neighborhood values are nearby, still has ~same # reductions.

How to avoid these attacks?
  Timing attack on decryption time: RSA blinding.
    Choose random r.
    Multiply ciphertext by r^e mod n: c' = c*r^e mod n.
    Due to multiplicative property of RSA, c' is an encryption of m*r.
    Decrypt ciphertext c' to get message m'.
    Divide plaintext by r: m = m'/r.
    About a 10% CPU overhead for OpenSSL, according to Brumley's paper.
  Make all code paths predictable in terms of execution time.
    Hard, compilers will strive to remove unnecessary operations.
    Effectively precludes efficient special-case algorithms.
  Can we take away access to precise clocks?
    Yes for single-threaded attackers on a machine we control.
    Can add noise to legitimate computation, but attacker might average.
```

```
    Can quantize legitimate computations, at some performance cost.
    But with "sleeping" quantization, throughput can still leak info.

How worried should we be about these attacks?
  Relatively tricky to develop an exploit.
  Relatively easy to notice attack on server (many connection requests).
  Often not the biggest security vulnerability, in many systems.
  Timing attacks: adversary has to be close by (not that big of a problem).
  However, if adversary mounts attack, effects are quite bad (key leaked).

Other types of timing attacks.
  Page-fault timing for password guessing [Tenex system]
    Suppose the kernel provides a system call to check user's password.
      Checks the password one byte at a time, returns error when finds mismatch.
    Adversary aligns password, so that first byte is at the end of a page,
      rest of password is on next page.
    Somehow arrange for the second page to be swapped out to disk.
      Or just unmap the next page entirely (using equivalent of mmap).
    Measure time to return an error when guessing password.
      If it took a long time, kernel had to read in the second page from disk.
      [ Or, if unmapped, if crashed, then kernel tried to read second page. ]
      Means first character was right!
    Can guess an N-character password in 256*N tries, rather than 256^N.
  Cache analysis attacks: processor's cache shared by all processes.
    E.g.: accessing one of the sliding-window multiples brings it in cache.
    Necessarily evicts something else in the cache.
    Malicious process could fill cache with large array, watch what's evicted.
    Guess parts of exponent (d) based on offsets being evicted.
  Cache attacks are potentially problematic with "mobile code".
    XFI modules, Javascript, Flash, etc running on your desktop or phone.
  Network traffic timing / analysis attacks.
    Even when data is encrypted, its ciphertext size remains ~same as plaintext.
    Recent papers show can infer a lot about SSL/VPN traffic by sizes, timing.
    E.g., Fidelity lets customers manage stocks through an SSL web site.
      Web site displays some kind of pie chart image for each stock.
      User's browser requests images for all of the user's stocks.
      Adversary can enumerate all stock pie chart images, knows sizes.
      Can tell what stocks a user has, based on sizes of data transfers.
    Similar to CRIME attack mentioned in guest lecture earlier this term.

References:
  http://css.csail.mit.edu/6.858/2012/readings/ht-cache.pdf
  http://tau.ac.il/~tromer/papers/cache-joc-20090619.pdf
  http://www.cs.unc.edu/~reiter/papers/2012/CCS.pdf
```

Read 11/13

AES-CBC + Elephant diffuser
A Disk Encryption Algorithm for Windows Vista

Niels Ferguson
Microsoft
niels@microsoft.com

August 2006

Abstract

The Bitlocker Drive Encryption feature of Windows Vista poses an interesting set of security and performance requirements on the encryption algorithm used for the disk data. We discuss why no existing cipher satisfies the requirements of this application and document our solution which consists of using AES in CBC mode with a dedicated diffuser to improve the security against manipulation attacks.

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

So not research
but protection

Contents

why not Truecrypt?

1 Introduction **1**

2 An overview of BitLocker Drive Encryption **1**

2.1 What it does 1

2.2 How it works 3

2.3 Disk encryption and authentication 4

2.3.1 Why not use a MAC? 4

2.4 Poor-man's authentication 5

2.5 Performance 6

2.6 BitLocker encryption algorithm requirements 7

2.7 Attack model 8

3 Existing ciphers **9**

3.1 Stream ciphers 9

3.2 AES-CBC 9

3.3 Bear and Lion 10

3.4 Beast 10

3.5 VIL 10

3.6 Mercy 11

3.7 LRW 11

3.8 CMC and EME 11

3.9 Conclusion 12

4 AES-CBC + diffuser **12**

4.1 Overview 12

4.2 AES-CBC 13

4.3 Sector key 14

4.4 Diffusers 14

4.5 About the name 15

5 Performance **16**

6 Analysis **16**

7 Use of AES-CBC + diffuser **16**

8 Acknowledgements **16**

A Sketch of a proof that AES-CBC + diffuser is as secure as AES-CBC **18**

1 Introduction

The Enterprize and Ultimate editions of Windows Vista contain a new feature called Bitlocker™ Drive Encryption which encrypts all the data on the system volume. BitLocker imposes some security requirements on the encryption algorithm that are not met by common encryption algorithms and modes. This creates a real problem: a new cipher cannot be trusted without many years of public review, and existing ciphers that satisfy the additional security requirements are either too slow or insufficiently analyzed.

We resolved this dilemma by combining a well-established cipher (AES in CBC mode) with a new component that we call the Elephant diffuser. The basic encryption security is provided by AES-CBC, which has been widely reviewed and is generally used in the industry for encryption. The diffuser layer adds some additional security properties that are desirable in the disk encryption setting but which are not provided by AES-CBC cipher methods.

This combination gives us the best of both worlds. All the security properties traditionally provided by encryption algorithms are provided by AES-CBC, which is an accepted cipher. We only depend on the diffuser for the additional security properties not provided by AES-CBC. The AES-CBC + diffuser approach is also faster than any of the alternatives, which is important for our application.

Fielding any kind of non-standard cryptographic algorithm like the diffuser is bound to be controversial. In this paper we explain the reasons why we made this choice, and document the diffuser so that the public cryptographic community can analyze it.

2 An overview of BitLocker Drive Encryption

Our design is driven by the particular requirements of BitLocker. Therefore, we will first describe BitLocker in its most common setting.

2.1 What it does

BitLocker is a security technology that targets a very specific scenario: the lost laptop.

Laptop computers regularly go missing, either because they are lost or because they are stolen. Our research indicates that typical laptop loss rates are around 1–2% per year. A large organization with 100,000 laptops loses several laptops each day.

These laptops contain confidential information, in the form of documents, presentations, emails, cached data, and network access credentials. This confidential information is typically far more valuable than the laptop hardware, if it reaches the right people. A laptop

is easy to replace at moderate cost; the cost of a data compromise can be many orders of magnitude higher.

BitLocker makes it harder to access this confidential information on a lost laptop. With the current generation of operating systems it is very simple to break into a laptop. One obvious way is to take the disk drive out of the laptop and connect it to a second machine as an auxiliary drive. All data can now be accessed using the administrator (root) privileges of the second machine. An even easier solution is to use a bootable floppy disk, CD, or USB key that contains a script that resets the Administrator (root) password. Such scripts and disks are available on-line, and anyone with an Internet connection and half an hour of time can download them. Once the Administrator password is reset, the laptop can be booted and the attacker can log in as the Administrator, giving him complete access to all information on the laptop.

The classic solution to this problem is to run a low-level disk encryption driver with the key provided by the user (passphrase), a token (smart card) or a combination of the two. The disadvantage of the classic solution is the additional user actions required each time the laptop is used. Most users are unwilling to go through these extra steps, and thus most laptops are unprotected.

BitLocker improves on the classic solution by allowing the user actions during boot or wake-up from hibernate to be eliminated. This is both a huge advantage and a limitation. Because of the ease of use, corporate IT administrators can enable BitLocker on the corporate laptops and deploy it without much user resistance. On the downside, this configuration of BitLocker can be defeated by hardware-based attacks.

Hardware attacks require the attacker to have significant skill and/or very specialized hardware, whereas software-only attacks can usually be automated, distributed over the internet, and carried out without any knowledge of the details of the system. We can therefore reasonably expect that the number of people that is capable and equipped to perform a hardware attack is far smaller than the number of people capable of performing a software-only attack. Thus, BitLocker significantly reduces the risk of data compromise, and makes it more likely that the laptop will simply be sold for the hardware value, rather than the value of the information on it.

The remaining vulnerability to hardware-based attacks seems fundamental for systems without user actions on boot. The cryptographic keys used to protect the confidential data must be available to the laptop during a normal boot, and can therefore be recovered by a hardware attack.

Stopping hardware attacks is possible, but requires the use of a token (e.g. USB key) and/or a user-memorized password or PIN. These options are fully supported by BitLocker, and they improve the security of the system. However, we expect that a large number of laptops will be used without PIN or USB key to avoid the need for user action on each reboot.

2.2 How it works

BitLocker obviously cannot be a software-only technology. Every software-only solution is vulnerable to software-only attacks.

BitLocker makes use of the TPM security chip which will be incorporated in most PCs in the near future. The TPM is a tamper-resistant chip mounted on the motherboard. Though the TPM has many functions [5], BitLocker uses only a few basic ones. Our description of the TPM is simplified and only covers those parts relevant for our purpose.

The TPM keeps several Platform Configuration Registers, or PCRs. At power-up the PCRs are set to zero. PCRs are only modified by the `extend` function which (effectively) sets a PCR to the hash of its old value and a supplied data string. We can think of a PCR as a hash over all the data strings provided in extend function calls for that PCR. There is no other way to set the value of a PCR, so if a PCR has value x after a sequence of extends, then the only way to reach the value x again is to perform the exact same sequence of extends after a power-up.

The seal/unseal functions of the TPM allow selective access to cryptographic keys based on PCR values. The seal function is used to encrypt a key into a string which can only be decrypted by that same TPM. Furthermore, the TPM will decrypt the string if and only if the selected PCRs have the value that was specified during the seal operation. In other words: we can store a key in an encrypted string so that it can only be accessed when selected PCRs have a particular value.

During the boot process the PCRs are used to keep track of the code that runs. The key used to encrypt the disk is sealed against a particular set of PCR values. During a normal boot the PCRs reach the same values, and the key can be unsealed by the TPM. If an attacker boots into any other operating system, the machine will be fully functional but the PCR values will be different and the TPM will not unseal the key. Thus, other operating systems cannot read the data on the disk, or find out how to modify the disk to reset the Administrator password.

In more detail, the process is the following: at power-up the processor starts running the BIOS from ROM. The first part of the BIOS cannot be modified. This part extends the BIOS PCR with the entire BIOS code and proceeds with the rest of the BIOS startup. After BIOS initialization the BIOS reads the Master Boot Record (MBR) of the hard disk, extends the boot sector PCR with the sector's data, and then executes the code in the boot sector. The boot sequence of a PC contains several more iterations, but in each case the newly-loaded code is first measured using an `extend` function before it is executed.

These functions do not interfere with the boot process of another operating system. Other operating systems can boot normally; but the TPM PCRs will have a different value. The PCRs merely report what software was run during the boot process.

Although extending PCRs provides good authentication of the code that is being run,

it is extremely inflexible. Any change to the code, for example for a patch or upgrade, leads to a different PCR value, and requires that the keys be re-sealed to the new PCR values. Therefore, the boot sequence switches to using BitLocker encryption at the first opportunity. Before the switch, PCRs are used to measure what code is running. At the switch point the TPM unseals the BitLocker volume encryption key. After the switch, all further data is read from the encrypted volume. Though BitLocker does not authenticate the data it reads from the disk in a cryptographic sense, it is very hard to perform a meaningful manipulation of the encrypted data.

2.3 Disk encryption and authentication

BitLocker encrypts almost all the data on the hard-disk. More precisely, it encrypts all the data on the operating system volume, which for most PCs spans almost the whole hard disk. The encryption protects the confidentiality of the data, which is a straightforward cryptographic problem, and can be solved with traditional ciphers.

However, to have a secure boot process we also need to authenticate the data from the disk. We don't want an attacker to modify the OS code in order to weaken the OS security. The normal cryptographic solution is to use an authentication code, but as we will see this is not practical. Therefore, we are left with using the encryption as a poor-man's authentication.

Imagine an attacker trying to break into the laptop using only software. He boots into some other OS. Because the PCRs are different he can't unseal the BitLocker key, so he can't read the encrypted volume. However, the attacker can modify the ciphertext on the hard disk in the hope of introducing a weakness in the OS. He then boots the machine normally, and exploits the created weakness during the boot or login process to gain access to the machine.

2.3.1 Why not use a MAC?

The obvious cryptographic solution to this problem is to add a Message Authentication Code (MAC) to each block of data on the disk. Disks store information in fixed-size sectors. Sectors are typically 512 bytes, though in the near future this will grow up to 4096 or even 8192 bytes. The operating system is designed to use sectors whose size is a power of two. Furthermore, the operating system accesses individual sectors of the disk in random order. These properties impose two constraints.

The first constraint is that the BitLocker encryption is done on a per-sector basis. Each sector is encrypted and decrypted independently of the other sectors. The second constraint is that the ciphertext cannot be larger than the plaintext. There is no extra room to store additional data. This means that we cannot store any nonce, IV, or MAC value with the

ciphertext. Together these constraints imply that the disk is effectively encrypted with a large block cipher in ECB mode where the block size is the sector size.

There are good engineering reasons behind these constraints. The encryption/decryption of one sector cannot depend on any other sector. There are applications, such as databases, that rely on the fact that they can write to sector x without danger of damaging sectors $x-1$ or $x+1$. They use this property to ensure that no information (other than possibly the sector that is being written) is lost in case of a crash or power failure. Suppose the encryption algorithm works in larger blocks than a single sector. To write sector x, the system first has to read the other sectors in the block, decrypt them, encrypt the new block, and then write all the sectors that make up the block. If the power fails when some of the sectors in the new block have been written but others have not, then the block has been corrupted. More importantly, unless the cipher works on a per-sector basis, writing sector x could corrupt some *other* sector. This violates the expectation of the application, and can lead to a loss of reliability of mission-critical applications. That is clearly unacceptable. (And re-writing all applications that depend on this property is impossible.)

It is tempting to add a nonce or MAC value to the ciphertext. However, making the ciphertext larger than the plaintext is not feasible. Both the disk and the operating system work in sectors whose size is a power of two. We could map a 512 byte OS sector into a 1024 byte disk-sector, but that would entail the loss of half the disk capacity, a price users are not willing to pay. We could reserve one in every 16 sectors to store the MAC and nonce values for the other 15 sectors, but this has several problems. First of all, writing to sector x means updating an additional sector that contains the nonce and MAC. This turns a write into a read-then-write with the associated performance loss. Furthermore, it could damage the nonce/MAC sector (e.g. if there is a power failure), which would lead to the loss of the other 14 data sectors; also unacceptable. Finally, for various usability, manageability, and deployment reasons, it must be possible to enable and disable BitLocker on an existing disk. (Think of a user upgrading to a new Windows version that includes BitLocker and enabling BitLocker on an existing disk.) Adding nonce/MAC sectors modifies the disk layout (and reduces the amount of available disk space) making it extremely complicated to enable and disable in-place. Add to that the various failure modes that can happen during the conversion, and it becomes infeasible to do this conversion in a reliable way.

The final conclusion is that we cannot add a MAC to each disk sector in a way that would be acceptable to most users.

2.4 Poor-man's authentication

That leaves us with an encryption algorithm that provides no authentication, yet we need authentication to provide a secure boot process. The best solution is to use poor-man's authentication: encrypt the data and trust to the fact that changes in the ciphertext do

not translate to semantically sensible changes to the plaintext.[1] For example, an attacker can change the ciphertext of an executable, but if the new plaintext is effectively random we can hope that there is a far higher chance that the changes will crash the machine or application rather than doing something the attacker wants.

We are not alone in reaching the conclusion that poor-man's authentication is the only practical solution to the authentication problem. All other disk-level encryption schemes that we are aware of either provide no authentication at all, or use poor-man's authentication.

To get the best possible poor-man's authentication we want the BitLocker encryption algorithm to behave like a block cipher with a block size of 512–8192 bytes. This way, if the attacker changes any part of the ciphertext, all of the plaintext for that sector is modified in a random way. We also want to prevent the attacker from moving the ciphertext of one sector to another sector, so the encryption algorithm should behave as a tweakable block cipher [8] with a slightly different algorithm for each sector.

For completeness we should mention that there are other authentication mechanisms that are used by the OS during the boot process, such as checking digital signatures on executables. Though these mechanisms are very valuable, they do not cover all of the data used during the boot and login process. From our point of view, these other mechanisms provide a second line of defense for some of the data, but we will ignore them for the rest of our discussion.

BitLocker also allows users to use a PIN that the TPM checks, or a USB key that contains a cryptographic key. Without the right PIN or USB key the laptop doesn't have the right information to even find the disk decryption key, so the information is safe unless the PIN is written on a post-it stuck to the machine, or the USB key is left in the laptop bag. In practice, we expect that many laptops will be used in the TPM-only mode and that scenario is the main driver for the disk cipher design.

2.5 Performance

The BitLocker disk cipher must be fast. If using BitLocker results in a significant and noticeable slowdown of the laptop there will be great user resistance to its deployment. When talking to customers about BitLocker, the question of performance is always one of the first questions they ask. Our analysis concluded that the performance loss of BitLocker must be minor for the feature to be used by a large number of customers. And given that Microsoft is not in the business of providing niche solutions, good performance was one of the hard requirements for BitLocker.

[1] This is not a compromise we like, but the only one that makes sense for the problem we're trying to solve.

Why? That sounds good...?

A typical desktop machine today has a 3 GHz P4 CPU and a hard disk that can read at about 50 MB/s. That means that the CPU has 60 clock cycles for each byte that the disk reads. Laptops have slower CPUs, often around the 1 GHz mark. Laptop disks are also slower but not by nearly as much. (For example, the Seagate Momentus 5400.2 laptop drive can read data at almost 50 MB/s.) Our data shows that laptops tend to have fewer CPU clock cycles per byte read from disk, down to 40 or even 30 cycles per byte. We cannot predict what the CPU/disk speed ratio will be for the actual hardware that BitLocker will run on, but these numbers are the best guidelines we have.

If decryption is slower than the peak data rate of the disk, the CPU becomes the bottleneck when reading large amounts of data. This is very noticeable, both because of the reduced performance and because of the reduced responsiveness of the UI when all CPU time is being used to decrypt data.[2] Therefore, decryption, including all overhead, must be faster than the disk to get an acceptable user experience.

BitLocker is carefully designed to overlap the reading of data from disk with the decryption of previously read data. This is only possible to a limited extent, and when the disk finishes reading the data, the CPU still has to decrypt (some of) the data. Thus the decryption time increases the latency of the disk request and reduces performance accordingly. This obviously argues for a fast decryption algorithm.

A software implementation of AES runs in around 20–25 cycles per byte on a P4 class CPU. (Synthetic benchmarks can achieve somewhat higher speeds, but they exclude various overheads encountered in real system implementations.) Other overhead adds around 5 cycles per byte for a total of 25–30 cycles per byte.

Based on this data, our performance analysis concluded that a single pass of AES, for example using AES in CBC mode, would have acceptable performance. An algorithm twice as slow as AES (45–55 cycles/byte) would be on the edge of being unacceptable, and a high-risk choice given the many uncertainties in the analysis. Anything slower than that would be unacceptable.

2.6 BitLocker encryption algorithm requirements

We get the following major requirements for our BitLocker encryption algorithm:

- It encrypts and decrypts disk sectors of size 512, 1024, 2048, 4096, or 8192 bytes.
- It takes the sector number as an extra parameter (the tweak) and implements different encryption/decryption algorithms for each sector.
- It protects confidentiality of the plaintext.

[2] There are many interesting issues involved in this analysis, including CPU scheduling, prioritization, and deadlocks, but we will not go into those details in this paper.

- It is fast enough that the slow-down of the laptop is acceptable to most users. Our best estimate is that a speed of 40 cycles/byte or faster will be acceptable.
- It has been validated by public scrutiny, and is generally considered safe.
- An attacker cannot control or predict any aspect of the plaintext changes if he modifies or replaces the ciphertext of a sector.

This is an extremely challenging set of requirements. Initially we tried to find a cipher that satisfies all the requirements, but we found no suitable cipher. The performance and public scrutiny requirements are especially difficult to satisfy.

2.7 Attack model

Normally, block ciphers are designed to withstand chosen plaintext and ciphertext attacks where the attacker has access to a black box that performs both the encryption and decryption operation. Also, the cipher is considered to be broken if it can be distinguished from a randomly chosen permutation.

Ideally our disk cipher (with the large 512–8192 byte block size) would satisfy these requirements, but we were unable to find a solution that achieves this and the other requirements too. To find a better solution we examined the BitLocker attack model in more detail to be able to take advantage of the specifics of the model.

In the BitLocker attack model we assume that the attacker has chosen some of the plaintext on the disk, and knows much of the rest of the plaintext. Furthermore, the attacker has access to all ciphertext, can modify the ciphertext, and can read some of the decrypted plaintext. (For example, the attacker can modify the ciphertext which stores the startup graphic, and read the corresponding plaintext off the screen during the boot process, though this would take a minute or so per attempt.) We also assume that the OS modifies some sectors in a predictable way during the boot sequence, and the attacker can observe the ciphertext changes.

However, the attacker cannot collect billions of plaintext/ciphertext pairs for a single sector. He cannot run chosen plaintext differences through the cipher. (He can choose many different plaintexts, but they are all for different sectors with different tweak values, so he cannot generate chosen plaintext differences on a single sector.) And finally, though this is not a cryptographic argument, to be useful the attack has to do more than just distinguish the cipher from a random permutation.

In short, we have the following attack model:

- The attacker has many known (but not chosen) plaintext/ciphertext pairs for different sectors.

- The attacker has the ciphertexts for a large number of chosen plaintexts for different sectors. The plaintexts are chosen before the attacker gets access to the laptop.
- The attacker has access to a slow decryption function for some of the sectors.
- The attacker gets several ciphertexts of plaintexts for the same sector with a known (but not chosen) difference.

The attacker succeeds if he can modify a ciphertext such that the corresponding plaintext change has some non-random property.

We don't want to argue that the standard attack model for block ciphers is wrong. Quite the opposite. We'd love to have a disk sector cipher that satisfies the standard requirements, but we have not been able to find one that satisfies the performance and validation requirements.

3 Existing ciphers

Before we dive into the details of our new design we'll discuss existing ciphers and why they are unsuitable.

3.1 Stream ciphers

There are many stream ciphers, but by their very nature, they allow the attacker to flip arbitrary bits in the plaintext. This lack of diffusion makes them entirely ineffective for poor-man's authentication.

3.2 AES-CBC

Any time you want to encrypt data, AES-CBC is a leading candidate. In this case it is not suitable, due to the lack of diffusion in the CBC decryption operation. If the attacker introduces a change Δ in ciphertext block i, then plaintext block i is randomized, but plaintext block $i+1$ is changed by Δ. In other words, the attacker can flip arbitrary bits in one block at the cost of randomizing the previous block. This can be used to attack executables. You can change the instructions at the start of a function at the cost of damaging whatever data is stored just before the function. With thousands of functions in the code, it should be relatively easy to mount an attack.

The current version of BitLocker implements an option that allows customers to use AES-CBC for the disk encryption. This option is aimed at those few customers that have formal requirements to only use government-approved encryption algorithms. Given the weakness of the poor-man's authentication in this solution, we do not recommend using it.

3.3 Bear and Lion

Bear and Lion are two large-block block ciphers proposed by Ross Andersen and Eli Biham [1]. Bear and Lion are very similar in construction. They split the data block into two unequal parts and create a 3-round Luby-Rackoff cipher by using a keyed hash function and a stream cipher to construct the round functions. The difference is that Bear uses two keyed hash function rounds and one stream cipher round whereas Lion uses one keyed hash function round and two stream cipher rounds.

Bear and Lion seem ideally suited, except for the fact that they are too slow. Both ciphers make three passes over the data. If we were to use SHA-256 for the hash function and AES-CTR for the stream cipher, the overall cipher would need close to 100 cycles/byte.

We tried several ways to make Bear/Lion suitable for BitLocker. However, the only way to make this solution fast enough is to use a fast stream cipher and a fast keyed hash function. We could not find suitable candidates that have had enough public review and have not yet been broken.

Our best attempt used AES-CBC for the keyed hash function, and AES-CTR for the stream cipher. Unfortunately, this solution is not fast enough.

3.4 Beast

Beast is a variation of Bear [9]. It is faster than Bear because it replaces the last round of Bear by a function that does not process the entire data block. Unfortunately, this change destroys the diffusion properties of the decryption function, making it unsuitable for our purpose.

We did consider using Beast upside-down, using the decryption function for encryption and the encryption function for decryption. This would seem to solve the immediate problem of the lack of diffusion. Even in this mode we do not feel comfortable with Beast. As far as we know Beast has not received any public review, and there has been no analysis of the upside-down mode. A final consideration is the speed of Beast. Though faster than Bear, it still requires two passes over the data; one with a hash function and one with a stream cipher. Our performance estimates for Beast with a well-established hash function and stream cipher were still slow enough to be a real problem.

3.5 VIL

VIL is a block cipher mode of operation that encrypts and decrypts arbitrary length messages [2]. This mode has received little attention, partly because it is patented.

For our purpose, the VIL mode is not suitable as most of the message is encrypted with a stream cipher. This provides no diffusion at all, and therefore very bad poor-man's-authentication properties.

3.6 Mercy

Mercy is a block cipher specifically designed for disk sector encryption [3]. Unfortunately, it was broken in 2001 by Scott Fluhrer [4], which eliminates it as a candidate.

3.7 LRW

LRW is a block cipher mode proposed by Liskov, Rivest, and Wagner. In LRW, a change to one block of the ciphertext results in a random change to the corresponding block of the plaintext, and no other changes. Effectively, it allows an attacker to randomize any block of plaintext.

LRW provides some level of poor-man's authentication, but the relatively small block size of AES (16 bytes) still leaves a lot of freedom for an attacker. For example, there could be a configuration file (or registry entry) with a value that, when set to 0, creates a security hole in the OS. On disk the setting looks something like `"enableSomeSecuritySetting = 1"`. If the start of the value falls on a 16-byte boundary and the attacker randomizes the plaintext value, there is a 2^{-16} chance that the first two bytes of the plaintext will be `0x30 0x00` which is a string that encodes the ASCII value '0'.

For BitLocker we want a block cipher whose block size is much larger. The same type of attack is still possible, but it is made harder by two factors: any particular attack point is far less likely to be on a suitable block boundary, and the attacker is forced to randomize more plaintext, increasing the likelihood that he will damage other parts of the system and crash the PC rather than open a usable hole.

3.8 CMC and EME

CMC and EME are two block cipher modes proposed by Halevi and Rogaway [7, 6]. They are directly targeted at our problem: encryption of disk sectors. CMC and EME have all the desired security properties, and are the leading contenders from the existing ciphers we identified.

However, they have not been widely studied or deployed, making them a relatively high-risk choice from a security point of view. (An earlier version of CMC was in fact broken.)

Furthermore, CMC and EME both require two AES encryptions for each block of data, making them a high-risk choice from a performance point of view.

A contributing factor in our decision not to use CMC or EME was that they are patented, and clearing the patent situation would require too much time.

3.9 Conclusion

In late 2004 we spent a lot of time looking for a suitable existing cipher, and failed to identify one. CMC and EME were the closest contenders, but for the reasons listed above we did not believe that they were suitable choices.

4 AES-CBC + diffuser

We finally chose to use AES-CBC for the primary encryption, and add a dedicated independently keyed diffuser to the plaintext side. This choice has advantages and disadvantages.

On the advantage side, it is trivial to see that AES-CBC + diffuser is at least as secure as AES-CBC, the industry workhorse algorithm for encryption. (See appendix A.) This provides a guaranteed security level for the confidentiality of the data, and one that customers can understand. The diffuser runs in about 10 clock cycles/byte so that the combination with AES-CBC satisfies our performance requirements.

On the disadvantage side, the diffuser is a new unproven algorithm, and this inevitably leads to questions. Without extensive public scrutiny and analysis of an algorithm there is a justified scepticism about its security. People are reluctant to trust new algorithms.

So why did we choose this option anyway? In our final analysis we decided this was the better choice for our product. The performance gain over the alternatives was important enough to outweigh the disadvantages of a new diffuser algorithm. Time will tell whether we made the right choice.

It bears repeating that even if the diffuser algorithm is utterly broken, all the data is also encrypted using AES-CBC and the confidentiality is still ensured. The only task of the diffuser is to make manipulation attacks harder by providing better poor-man's authentication than plain AES-CBC provides. Given the limited BitLocker attack model we consider it extremely unlikely that a practical attack against the diffuser will be found.

4.1 Overview

Figure 1 gives an overview of our solution. There are four separate operations in each encryption. The plaintext is exclusive-orred (xorred) with a sector key, then run through two (unkeyed) diffusers, and finally encrypted with AES in CBC mode.

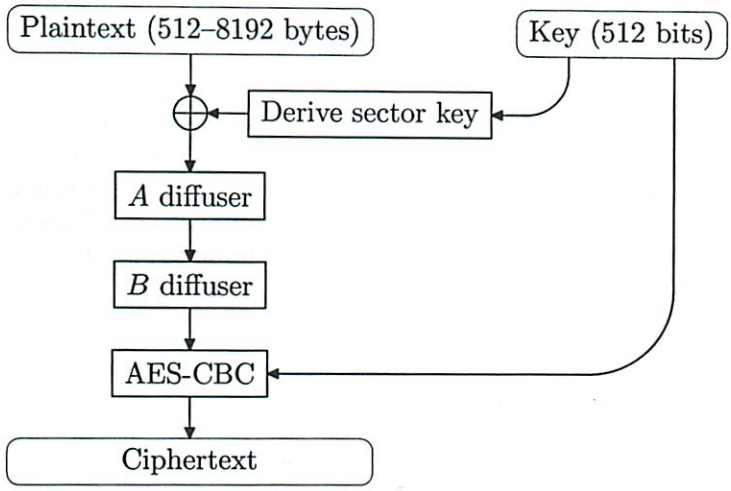


Figure 1: An overview of AES-CBC + diffuser

The sector key component and the AES-CBC component are independently keyed which allows an easy proof that our construction is at least as secure as AES-CBC (see appendix A). Both components are provided with 256 bits of key material, so that the full key is 512 bits. Depending on the selected version, the two keyed components can use fewer than 256 bits each, so some of the key bits may go unused. The full key is always 512 bits to support larger keys without any changes to the key management system. By default both the sector key and the AES-CBC layer use 128-bit AES keys; a version that uses 256 bits for each is also available.

The block size is variable: it can be any power of two within the range 512–8192 bytes (4096–65536 bits).

4.2 AES-CBC

The AES-CBC component is straightforward. The AES key K_{AES} is either 128 bits or 256 bits, depending on the selected version. The block size is a always a multiple of 16 bytes, so no padding is necessary. The IV for sector s is computed as:

$$IV_s := E(K_{\text{AES}}, e(s))$$

where $E()$ is the AES encryption function, and $e()$ is an encoding function that maps each sector number s into a unique 16-byte value. Note that IV_s depends on the key and the sector number, but not on the data.

The plaintext is encrypted using AES-CBC and the IV for the sector. Decryption is the obvious inverse function.

The result of $e()$ is the tweak value of this part of the cipher. The choice of $e()$ has no security implications (as long as it is an injection) and will vary with the application. For BitLocker the encoding function e is the simplest one for the implementation. The first 8 bytes of the result are the byte offset of the sector on the volume. This integer is encoded in least-significant-byte first encoding. The last 8 bytes of the result are always zero.

4.3 Sector key

The sector key for sector s is defined by:

$$K_s := E(K_{\text{sec}}, e(s)) \parallel E(K_{\text{sec}}, e'(s))$$

where $E()$ is the AES encryption function, K_{sec} is the 128 or 256-bit key for this component, $e()$ is the encoding function used in the AES-CBC layer, and $e'(s)$ is the same as $e(s)$ except that the last byte of the result has the value 128.

The sector key K_s is repeated as many times as necessary to get a key the size of the block, and the result is xorred into the plaintext.

4.4 Diffusers

The A and B diffusers are very similar, but work in opposite directions. Our core diffuser design has good diffusion properties in one direction and bad diffusion properties in the other direction. Having two diffusers provides good diffusion in both directions.

The diffusers have been designed in the decryption direction, as decryption is the more common operation. We will describe them first in the decryption direction, and later show the corresponding encryption function.

Each diffuser interprets the sector data as an array of 32-bit words, where each word is encoded using the least-significant-byte first convention. Let n be the number of words in the sector, and $(d_0, d_1, \ldots, d_{n-1})$ be the words of the sector. For index values outside the range we define $d_i := d_{i \bmod n}$ to allow easy wrap-around without confusing notation.

The decryption function of the A diffuser is given by:

$$\textbf{for } i = 0, 1, 2, \ldots, n \cdot A_{\text{cycles}} - 1$$
$$d_i \leftarrow d_i + (d_{i-2} \oplus (d_{i-5} \lll R^{(a)}_{i \bmod 4}))$$

The value i is a loop counter that goes around the data array A_{cycles} times. (Remember that all indices are modulo n, so the wrap-around is automatic.) The addition is modulo

2^{32}, $\lll$ is the rotate-left operator, and $R^{(a)} := [9, 0, 13, 0]$ is an array of 4 constants that specify the rotation amounts.

The corresponding encryption function of the A diffuser is easy to derive:

$$\begin{aligned}&\textbf{for } i = n \cdot A_{\text{cycles}} - 1, \ldots, 2, 1, 0\\ &\qquad d_i \leftarrow d_i - (d_{i-2} \oplus (d_{i-5} \lll R^{(a)}_{i \bmod 4}))\end{aligned}$$

The asymmetric diffusion properties are easy to see. If we look at the A diffuser decryption, the result of one iteration is used 2 and 5 iterations later which quickly propagates changes to the rest of the sector. In the encryption direction the output of one iteration is used $n - 5$ and $n - 2$ iterations later, which provides a much slower diffusion.

The B diffuser is very similar. It has good diffusion in the encryption direction. The B diffuser decryption function is given by

$$\begin{aligned}&\textbf{for } i = 0, 1, 2, \ldots, n \cdot B_{\text{cycles}} - 1\\ &\qquad d_i \leftarrow d_i + (d_{i+2} \oplus (d_{i+5} \lll R^{(b)}_{i \bmod 4}))\end{aligned}$$

where $R^{(b)} := [0, 10, 0, 25]$. The B diffuser encryption function is

$$\begin{aligned}&\textbf{for } i = n \cdot B_{\text{cycles}} - 1, \ldots, 2, 1, 0\\ &\qquad d_i \leftarrow d_i - (d_{i+2} \oplus (d_{i+5} \lll R^{(b)}_{i \bmod 4}))\end{aligned}$$

The constants A_{cycles} and B_{cycles} define how many times each of the diffusers loop around the sector, and are chosen as $A_{\text{cycles}} := 5$ and $B_{\text{cycles}} := 3$.

To choose the rotation amounts we ran experiments on the diffusion properties of our diffuser recurrence (in the high-diffusion direction). For performance reasons we only use a nonzero rotation amount every other iteration (the P4 processor has a very slow rotation instruction). We concentrated on how quickly a single bit difference diffuses through the 32-bit active word. Our results are that if we flip a single bit in d_i, each of the bits of d_{i+43} has a chance of at least 1/3 of flipping in a single forward cycle of the diffuser.[3] As a sector is at least 128 words long, we get full diffusion within the word in about one third of a cycle.

4.5 About the name

The name Elephant was chosen for the diffuser component to fit in with the Bear and Lion ciphers discussed earlier.

[3] This holds for the forward B diffuser. For the backward A diffuser a bit in d_i flips bits in d_{i-43} with probability 1/3.

5 Performance

Our AES implementation uses about 20 cycles/byte for AES-CBC on a a Pentium 4. The diffuser takes about 10 cycles/byte. The overall cipher speed is just over 30 cycles per byte, including various overhead. This implies that the cipher is faster than the peak data rate of a typical disk.

Our current BitLocker implementation manages to limit the loss of performance to around 5% averaged over our test cases. Our typical end-user test scenarios show an even smaller overhead. This is good enough to allow widespread adoption of this security technology.

6 Analysis

An extensive analysis of our construction is still ongoing, and will be published separately.

7 Use of AES-CBC + diffuser

This cipher is designed specifically for the role of disk sector encryption algorithm in the BitLocker setting. It is *not* a general-purpose block cipher, and should not be used in other settings without careful analysis. As a pure block cipher our construction has many weaknesses when analyzed in the standard block cipher attack model. For example, some of the key bits are unused, and guessing part of the key is enough to distinguish it from a random permutation.

8 Acknowledgements

I'd like to thank Josh Benaloh for his helpful comments on the design, and the System Integrity team at Microsoft for their tireless energy in bringing this to actual use.

References

[1] Ross Anderson and Eli Biham. Two practical and provable secure block ciphers: BEAR and LION. In Dieter Gollmann, editor, *Fast Software Encryption: Third International Workshop (FSE'96)*, LNCS 1039, pages 113–120. Springer Verlag, 1996.

[2] Mihir Bellare and Phillip Rogaway. On the construction of variable-input-length ciphers. In Lars Knudsen, editor, *Fast Software Encryption: 6th International Workshop, FSE'99*, LNCS 1636, pages 231–244. Springer Verlag, 1999.

[3] Paul Crowley. Mercy: a fast large block cipher for disk sector encryption. In Bruce Schneier, editor, *Fast Software Encryption: 7th International Workshop, FSE 2000*, LNCS 1978, pages 49–63. Springer Verlag, 2001.

[4] Scott R. Fluhrer. Cryptanalysis of the Mercy block cipher. In Mitsuru Matsui, editor, *Fast Software Encryption, 8th International Workshop, FSE 2001*, LNCS 2355, pages 28–36. Springer Verlag, 2002.

[5] Trusted Computing Group. *TCG TPM Specification Version 1.2.* Available from `www.trustedcomputinggroup.org`.

[6] Shai Halevi and Phillip Rogaway. A parallelizable enciphering mode. `http://eprint.iacr.org/2003/147`, 2003.

[7] Shai Halevi and Phillip Rogaway. A tweakable enciphering mode. `http://eprint.iacr.org/2003/148`, 2003.

[8] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In Moti Yung, editor, *Advances in Cryptology—CRYPTO 2002*, LNCS 2442, pages 31–46. Springer Verlag, 2002.

[9] Stefan Lucks. BEAST: A fast block cipher for arbitrary blocksizes. In Patrick Horster, editor, *Communications and Multimedia Security II, Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security*, IFIP Conference Proceedings 70, pages 144–153. Chapman & Hall, 1996.

A Sketch of a proof that AES-CBC + diffuser is as secure as AES-CBC

It is not possible to prove that a particular algorithm is secure. But there are techniques for showing security implications: proving that algorithm A is as secure as algorithm B.

In our case we want to show that AES-CBC + diffuser is at least as secure as using just AES-CBC. We won't do a formal proof, which requires lots of formal definitions, but we present a sketch of the argument.

Suppose an attacker is attacking two identical disks, one encrypted with AES-CBC and one with AES-CBC + diffuser. The exact attack model (what the attacker can do with the disks during the attack) is not important here, as long as it is the same for both disks.

The crucial observation is that giving the attacker more information cannot make it harder for him to perform the attack. The attacker is always free to ignore any additional information we give him. Providing more information can make it easier to attack the system, but never harder.

So here is what we do: we give the attacker the key K_{sec} used for the AES-CBC + diffuser disk. This is the AES key used to derive all the sector keys. This means that the attacker can now compute every sector key, and perform all the diffuser operations on any plaintext. In effect, the sector key and diffuser operations become transparent to the attacker, and the remaining problem is to attack the AES-CBC layer, which is exactly the problem of attacking a disk encrypted with only AES-CBC. We have helped the attacker significantly, and he is still faced with attacking a disk encrypted with AES-CBC. This shows that AES-CBC + diffuser cannot be easier to attack than just AES-CBC.

How come someone can't send the same key if inputs both? Or is that the HW vulnerability they mention?

6.858: Computer Systems Security

Fall 2012

Home

General information

Schedule

Reference materials

Piazza discussion

Submission

2011 class materials

Paper Reading Questions

For each paper, your assignment is two-fold. By the start of lecture:

- Submit your answer for each lecture's paper question via the submission web site in a file named `lecn.txt`, and
- E-mail your own question about the paper (e.g., what you find most confusing about the paper or the paper's general context/problem) to 6.858-q@pdos.csail.mit.edu. You cannot use the question below. To the extent possible, during lecture we will try to answer questions submitted by the evening before.

Lecture 18

Suppose an adversary steals a laptop that uses BitLocker disk encryption. In BitLocker's design, Windows has a key to decrypt the contents of the drive.

1. What prevents the adversary from extracting this key from Windows?
2. If the adversary cannot extract the key, what prevents him or her from simply using Windows to access files?

1. ~~That is a HW att~~ The TPM stores this key, and only activates it windows boots

2. Windows enforces pw

I want to know if I am right
I read it rather quickly...

Questions or comments regarding 6.858? Send e-mail to the course staff at 6.858-staff@pdos.csail.mit.edu.

Top // **6.858 home** // *Last updated Friday, 12-Oct-2012 23:31:47 EDT*

11/18

Paper Question 18

Michael Plasmeier

1. The TPM stores this key and can only access it if the proper series of `extend` commands are used.
2. Windows enforces the password using its normal method. (But isn't this insecure?)

Bitlocker
Disk encryption

11/21

(7 min late)

Laptop theft

Goals: confidentiality
integrity
freshness
deniability

Key: user enters pw
usability: since ask pw twice (Bios + Windows)
Ubuntu just uses a pw to encrypt the home directory
but had for Windows to add
to ensure compatability

Password is often weak
Windows can enforce special rules like
freeze for 10 min when 3 incorrect passwords
Or non pw: smart cards, photo, finger print

(2)

All which Bios might not support
or can't directly use to decrypt

USB key
high quality key
but will likely have them together

Better: Let OS enforce
Give key to OS

Ensure OS is legit

How to check the OS?

- Signed boot

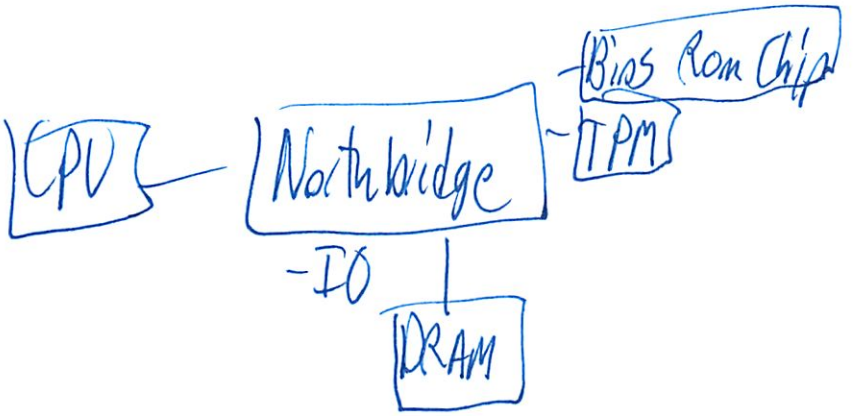
Only boot OS signed by key

- mobile phones do
- chrome OS
- game consoles
- but people worried its for commercial monopolies
- Not PCs
- single pt of failure: key

- Authenticated boot

Identify the OS (hash)
Then give it access to diff keys

(3)



TPM

A bunch of registers PCR_n

- platform config registers
- 160 bit long
- 20 of them
- $PCR_0 \ldots PCR_{19}$

TPM_extend(m,n):

$PCR_n \leftarrow H(PCR_n \| m)$

builds hash chain

Private key K

(9)

TPM_quote(m,n):

a way to convince someone across network that you are running authentic sw

$Sign_k(m, PCR_n)$

network must trust TPM is doing right thing

Bit Locker doesn't use quote

TPM_seal(m,n, PCRval)

Encrypt message for specific PCR value

encrypt m, so decrypt only if n = PCRval

$enc_k(m, n, PCR\ val)$

TPM_unseal(c)

m only if $PCR_n = PCR\ val$

(5)

(missed)

Slow TPM chips
Since cheap
not much tamper resistent mechanism either
Prof: keep in mind when using
Better than nothing
But not attack proof

Who calls extend:
Bootloaders in sequence must do this

Reset: TPM resets all PCRs ← base invariant
and CPU starts running BIOS

Then each do section:

Bios: m ← read(boot_sector)
TPM_extend(m, 0)
run(m)

(6)

Boot sector: m ← read (windows)
TPM_extend (n, 0)
run (n)

So must trust Bios, Boot sector, OS
if bug in boot loader could be doing wrong thing

Must trust HW to run Bios

So BK = Bitlocker key
Store w/ ← TPM_seal (BK, 0, hash of windows)
EBk ← store on disk

When Boot
TPM_Unseal (EBk)
Only if PCR0 has hash of windows

BK

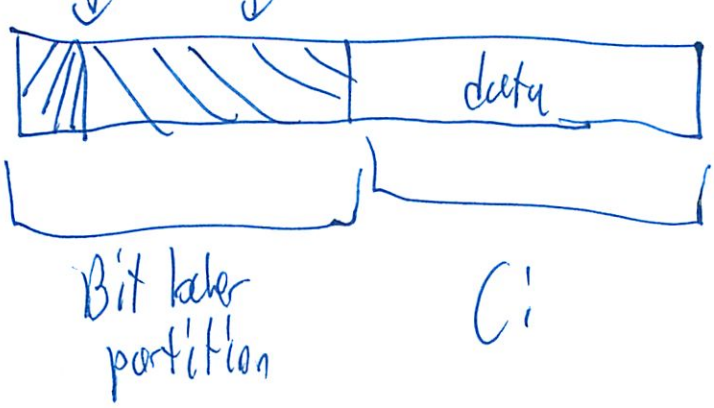
(7)

Attacks

If listening in on TPM or can disconnect it
Then can call extend yourself w/
all the right hashes (which are public)
That is the HW attack they talked about

"SSL" b/w CPU + TPM
└ Not; possibly, I forget
But must make sure channel is not replaced

BIOS → boot sector → Boot loader



(8)

But at some point they have a diff plan
Since if you write a file PCR changes!
So rely on as little as nessary

- Bitlocker running
  like little Windows kernel
  has key
  now how to decrypt the data..?

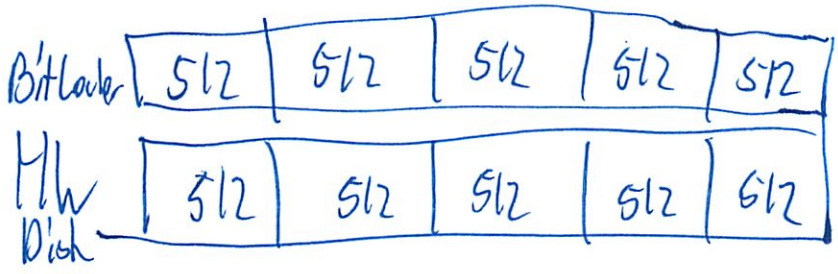
~~Sorry~~

If wanted GRUB
  would need to trust GRUB
  Otherwise it could just extend Windows
    w/ any hash, even if not the real hash of

(9)

Encrypt at block level

Since DBs rely on only reading/writing 1 block
- fast
- atomic



represented to file system
must store on disk

But how to encrypt + authenticate?

No extra metadata

So no space for - sigs, MACs, checksums
unless side table
- but read/2 things ← slow
- write " " ← non atomic

Need to encrypt 1:1

Net App does this
- 520 sector HDDs, very expensive $

(10)

(missed) Main goal → confidentiality

(missed) Want attacker not to modify windows to disable password checking → authentication

What attacks are we worried about?

Only need to find 1 sector to corrupt

(don't get)

Poor man's authentication

General encryption block cipher

ie AES.

E(k, block) → c

D(k, c) → block

Size(c) = size(block)

~ 128 bits

16 bytes

1. Fast 200 MB/sec
2. Fixed sized data

(11)

Running many times over is non trivial

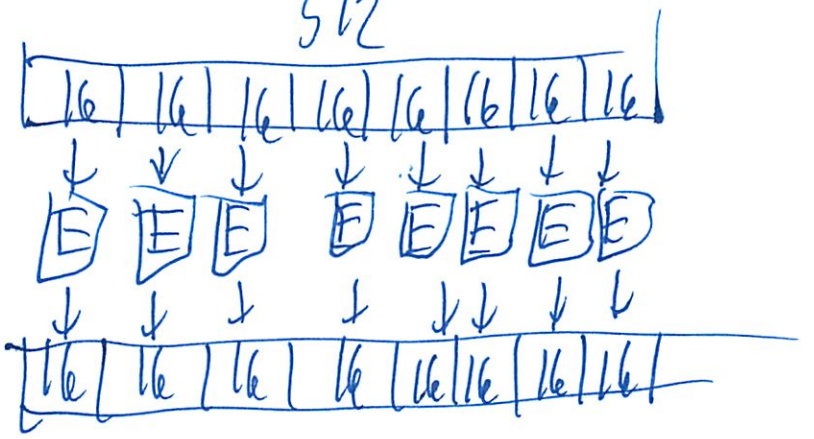
So <u>modes of operations</u> for Block ciphers

lots of these out there

2 we can talk about

1. <u>ECB</u> Electronic Code Book

magic lookup book



fast → in parallel

atomic

but disasteras → can corrupt precisely

easy to Cryptanalize

cipher
plain

- but AES makes it hard even if have lots of pairs

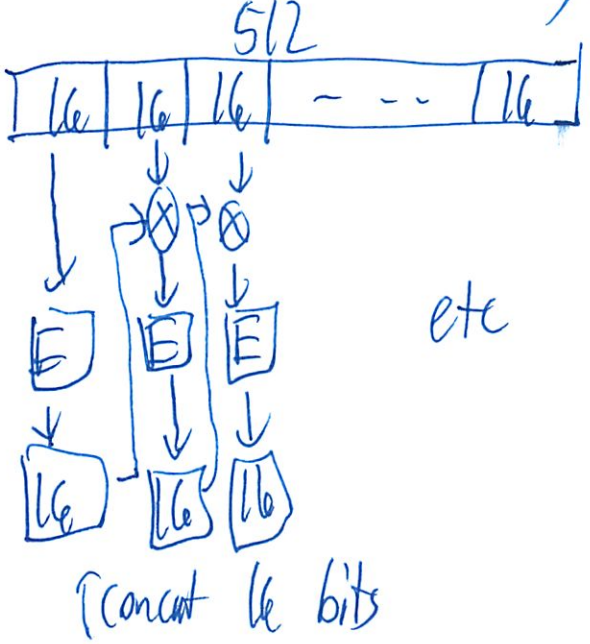
(12)

But often not secrecy

if same file has repeted values

Could have lists of SSNs and see if any overlaps

2. CBC Cipher block chaining

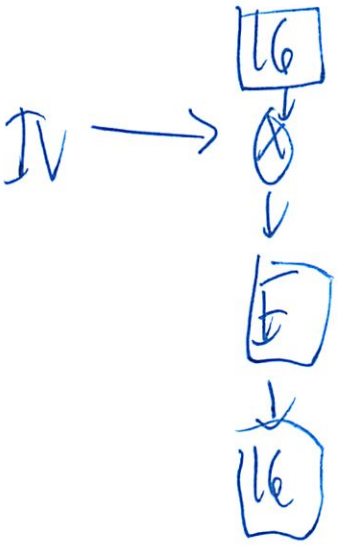


decrypt by doing this in reverse

all cipher text looks diff

(13)

People sometimes start w/ an IV
So same plain text looks diff



IV could be public, as long as diff for each sector
ie Sector #
or H(Sector #)

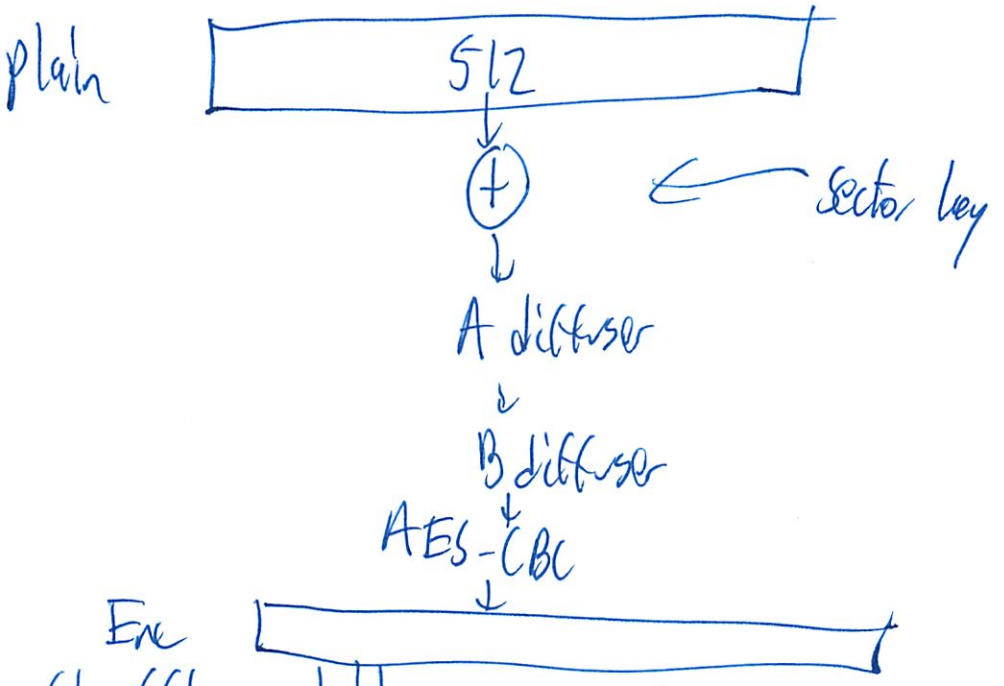
This is AES-CBC

But try to fix that last of block
Can get delibetly corrupting

Crypto is usually very subtle

Don't want to break it!

So add a diffuser



Shuffle bits around

deterministic - no key

So purposeful corruption will be spread around the disk

Want to make sure attacker can't carefully craft input to diffuser

(15)

Make sure don't reuse the AES key

Securely derive diffuser + AES key
Sector key

Blk → Sector key
→ AES key

But can you easily reset Windows Ph

Requires Windows to be secure

Enterprises set it up

They lock Windows down

Store key in Active Directory so IT
Can recover

Could also transplant TPM chip
prob in China they will do it

Also easy to read key from TPM chip
~ $100k to do
need super special probes

Performance

Must be faster than disk
SSDs are fast
But HW AES now

Diffuser could be AES both ways
but that would be slower

Still some performance hit since CPU
is decrypting

11/[illegible]2(

BitLocker

What's the problem this paper is trying to solve?

Prevent data from being stolen if an attacker physically steals a laptop.
Effectively, want some form of disk encryption.

What security properties might users want from disk encryption?

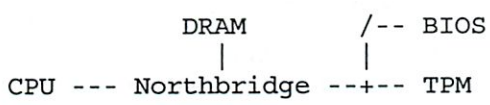
Data secrecy: adversary that gets access to disk cannot get data.
Data integrity: adversary cannot replace data on disk without detection.
Data freshness: adversary cannot roll back to an old version without det.
Deniability:
  Ensure that adversary can't tell you have certain data encrypted.
  Ensure that adversary can't tell if they have the real password.
  E.g., someone forces you to give up your password.
  [Related case: https://www.eff.org/cases/us-v-fricosu]

Where do encryption keys come from?

Simple approach: user provides key in some form.
  User might enter password (hashed to produce key, like in Kerberos).
    Problem: passwords are weak, adversary can try to guess the right one.
  User might plug in a USB drive containing the key.
    Problem: users don't want to carry around extra USB keys.
    Problem: users might lose the USB key along with the laptop.

Bitlocker: give key to legitimate OS, let OS verify user's credentials.

How to tell if legit OS is running, or attacker boots from his own CD?
One approach: only run software signed by some authority (signed boot).
  E.g., a PC only runs Windows signed by Microsoft.
  Even if adversary boots from CD, can only boot Windows.
  Windows will still enforce file permissions, password checks, etc.
  Sometimes works on specialized devices (game consoles, cell phones, ..)
  Too restrictive for general-purpose PCs.
Another approach: authenticated boot.
  Instead of restricting what software can boot, authenticate ("measure") it.
  Use a piece of trusted hardware: a TPM (Trusted Platform Module).



TPM chip has an ephemeral set of registers (PCR0, PCR1, ..), and a key.
Some supported operations (others also exist, but irrelevant here):
  TPM_extend(m): extend a PCR register, PCRn = SHA1(PCRn || m)
  TPM_quote(n, m): generate signature of (PCRn, m) with TPM's key
  TPM_seal(n, PCR_value, plaintext): return ciphertext.
  TPM_unseal(ciphertext): return plaintext, if PCRn matches PCR_value.

Who does the measurement for authenticated boot?

PCR values get reset to zero only when the entire computer is reset.
Important: CPU must jump to BIOS code, which is not tampered with.
BIOS code "measures" itself: extends PCR with hash of its code.
BIOS code loads boot loader (e.g., Linux grub), measures it
  (extends PCR with the hash of the boot loader), runs it.
Boot loader loads kernel, measures it (extends PCR with hash of
  kernel), runs it.

What can we infer if some PCRn corresponds to a particular chain of hashes?

Could be that the right software chain was loaded.
Or some software along the way had a bug, was exploited, and adversary
  issued their own extends from that point forward in the chain.
Or the CPU did not start with the BIOS code in the first place.

```
  Or the TPM hardware did not reset synchronously with the CPU.
    [ Turned out to be "easy" on some motherboards: just short out a pin. ]

What does this allow us to do?
  Can prove to others over the network that you're running some software.
    Use TPM_quote() to get the TPM to sign a message on your behalf.
    Assumption: remote party trusts your TPM (but not you directly).
    TPM has its own secret key, HW mfg signs public key, stores cert on TPM.
  Can encrypt data in a way that's only accessible to specific software.
    Use TPM_seal, TPM_unseal.
    Sealed data can be decrypted only by chosen recipient (PCR).
    Each TPM has its own randomly-generated key for encryption.
  Assumption: adversary does not tamper with CPU, TPM, or their link.

Bitlocker's TPM mode: key stored in TPM.
  Idea: store key in the TPM (or rather, seal it using the TPM).
    Advantage: no need for user to interact with the BIOS.
  What's the point of TPM-only mode?
    Key can only be obtained if the machine boots up the same OS (Windows).
    As a result, security boils down to whatever plan Windows has.
    One possibility: user has Windows password.
      Why is this better than the password-in-BIOS approach?
      1. No need to enter password twice: in BIOS and in Windows.
      2. Windows can rate-limit login attempts, prevent pw guessing.
    Another possibility: user cannot access sensitive data directly.
      User might have to access sensitive data via privileged process.
      Privileged process will not divulge entire data set.

What gets measured at boot in BitLocker?
  Two partitions on disk.
  First partition contains BitLocker's bootstrapping code.
  Second partition contains encrypted data ("OS volume").
  First partition measured at boot.
  BitLocker's key sealed with first partition's PCR measurement.
  Why not measure the second partition?
    Changes frequently.
    Expectation: adversary won't be able to meaningfully change it.
  What if we need to upgrade the first partition?
    One possibility: re-seal key with new PCR value before upgrade.
  What if we need to upgrade laptops?  Or, laptop died and need to recover?
    Disk encryption key is stored encrypted with a recovery password.
    (Or, stored in Active Directory encrypted with admin's password.)
    User can type in their recovery password to gain access to disk.

How do we encrypt the disk once we have the key?
  Encrypt disk blocks (sectors) one-at-a-time.
    Why one-at-a-time?  Atomicity, performance.
  Potential problem: integrity (adversary can modify sectors on disk).
    Why is this a problem for a disk encryption scheme?
    Why is it insufficient to do secure boot (check signatures on code)?
  What are the options for ensuring integrity?
    Ideally, store a MAC (~keyed hash) for the sector somewhere on disk.
    Recall, disks write sectors at a time: need one MAC per sector.
    Store MAC in adjacent sector: effectively cut space by a factor of 2.
    Store MAC in a table elsewhere: two seeks (and breaks atomicity).
    Store MACs for group of sectors nearby: breaks atomicity.
  Where can we store MACs for integrity?
    Buy really expensive disks (NetApp, EMC) that have jumbo sectors.
    "Enterprise" disks have 520-byte sectors, instead of standard 512.
    Extra 8 bytes used to store checksums, transaction IDs, etc.
    Could be used to store MAC.
    Not going to fly for common machines.
  BitLocker approach: "poor-man's authentication"
```

```
    Assume adversary cannot change the ciphertext in a "useful" fashion.
    I.e., cannot have a predictable effect on the plaintext.
  When would this work or not?
    Works if applications detect or crash when important data is garbled.
    Must be true at sector-level, which attacker can corrupt separately.
    Probably true for code: random instructions will raise an exception.
    Worst case for data: 1 bit (e.g., "require login?") alone in a sector.
    Adversary can guess random ciphertexts, see when that bit changes.
    If application doesn't notice other bits corrupted, game over.
    Hopefully this is not how the registry is constructed, so maybe OK..
  What about freshness?
    Harder to achieve: need to have some state that can't be rolled back.
    Strawman: hash all blocks, store hash in TPM.
    Problem: updates require re-hashing entire disk, slow.
    Idea: tree of hashes (Merkle tree).
    Even that is often too expensive to update.
  How do we achieve poor-man's authentication?
    Let's look at some encryption mechanisms.
    We consider symmetric-key encryption schemes (no need for public-key).
    Key sizes are typically on the order of 128 bits for symmetric-key.

Block ciphers.
  Encrypt entire block of data at a time; hopefully no direct bitwise deps.
  Standard block cipher today: AES, 128-bit block size (128- or 256-bit key).
  How to encrypt something larger than a block size?

Block cipher modes of operation
  Have plaintext blocks P_1 .. P_n
  Want ciphertext blocks C_1 .. C_n
  ECB: Apply block cipher to each block-sized chunk in turn.
    C_i = E_k(P_i)
    Advantage: simple.
    Disadvantage: attacker can permute blocks, find equal blocks, ..
  CBC: XOR each block's plaintext with previous block's ciphertext.
    C_i = E_k(P_i XOR C_{i-1})
    Can we decrypt this?  P_i = D_k(C_i) XOR C_{i-1}
    Advantage: attacker cannot permute blocks.
               different ciphertexts for matching plaintext blocks.
    Disadvantage: encryption hard to parallelize (but decryption can be).
    What if plaintext is changed by 1 bit?  All subsequent C's affected.
    What if ciphertext changed by 1 bit?  Only 2 plaintexts affected.
      Plaintext for changed ciphertext is fully garbled.
      Plaintext for next block is changed by 1 bit, in the same position!
    What's P_0 here, the initial XOR value?
      Called the initiatlization vector (IV).
      Needed for decryption, to decrypt the first block.
    What if IV is revealed after encryption?  Usually OK.
    What if IV is predictable or reused?  Usually a bad idea.
      Leaks information about the contents of first block.
      Watermarking attacks on file system.
        Adversary can create file patterns that are visible after enc.
    How to choose IVs for disk encryption?
      Typically want to choose different IVs for different sectors.
        Different encryptions for different sectors.
      Avoids watermarking, swapping of sectors by adversary.
      However, IV is typically reused, so ideally keep IV secret.
  Many more block cipher modes of operation exist.
    Can provide provable CCA2 security (UFE), authentication (OCB/CCM), ..

BitLocker's AES-CBC + Elephant mode
  Figure 1 in the paper.
  Start with basic AES-CBC: closest to what we want.
  Why is the IV derived from the sector number?
```

```
    Make encryption functions of each sector different, prevent switching.
  Why is the sector number encrypted in the IV?
    Make sure the IV is not predictable, avoid watermarking attacks.
  What are the diffusers doing?
    XORing bits of the block.
    Ensures that one bit change on either side will flip many other bits.
  Why do we have another sector-dependent transform upfront (sector key)?
    So that diffusers don't operate on predictable data on either side.
    Even if adversary learns the IV for a pair of sectors, cannot swap them.
  Sector key: why is it different from the AES key?
    Guarantees that sector-key operations cannot weaken AES.
    Even if the entire secret key is leaked, AES-CBC still intact.
    Good way to think of extending crypto schemes: never reuse keys!

Potential attacks on BitLocker?
  Not intended to be a perfect security solution, by design.
  Hardware attacks: DMA, cold boot attacks, ..
  Security vulnerabilities in Windows (buffer overflows, root access).
  Rolling back disk blocks to an old version (i.e., violate freshness).
    Adversary likely doesn't have interesting old blocks.
    Hard (expensive in terms of performance) to defend against.

Alternative to BitLocker's sector encryption: filesystem-level encryption.
  E.g., ecryptfs in Linux, used by Ubuntu's home directory encryption.
  FS can solve atomicity/consistency problems.
  FS can find space for extra MACs, random IVs to prevent IV reuse, etc.
  FS might require much more code to be "measured" into the TPM.
    FS comes up much later in the boot process.
    Requires TPM re-sealing for FS upgrades, driver upgrades, etc.
  FS might not interpose on swapping/paging.
  FS harder to deploy (changes to FS, cannot deploy incrementally).
```

TrInc: Small Trusted Hardware for Large Distributed Systems

Dave Levin, *University of Maryland*
John R. Douceur, *Microsoft Research*
Jacob R. Lorch, *Microsoft Research*
Thomas Moscibroda, *Microsoft Research*

Abstract

A simple yet remarkably powerful tool of selfish and malicious participants in a distributed system is "equivocation": making conflicting statements to others. We present TrInc, a small, trusted component that combats equivocation in large, distributed systems. Consisting fundamentally of only a non-decreasing counter and a key, TrInc provides a new primitive: unique, once-in-a-lifetime attestations.

We show that TrInc is practical, versatile, and easily applicable to a wide range of distributed systems. Its deployment is viable because it is simple and because its fundamental components—a trusted counter and a key—are already deployed in many new personal computers today. We demonstrate TrInc's versatility with three detailed case studies: attested append-only memory (A2M), PeerReview, and BitTorrent.

We have implemented TrInc and our three case studies using real, currently available trusted hardware. Our evaluation shows that TrInc eliminates most of the trusted storage needed to implement A2M, significantly reduces communication overhead in PeerReview, and solves an open incentives issue in BitTorrent. Microbenchmarks of our TrInc implementation indicate directions for the design of future trusted hardware.

1 Introduction

As wide-area systems grow in scale, so do their exposure to threats. Much of the interesting distributed-systems research of the past decade has focused on the issues of security and adversarial incentive that are inherent to large-scale systems. This research has addressed a wide range of applications, including storage [2, 16, 19, 22, 28], communication [4, 45, 30], databases [40], content distribution [15, 24, 32, 36], grid computation [12], and games [3, 10], in addition to generic infrastructure [1, 5, 9, 18, 23, 43]. Virtually all of this work shares a common supposition, namely that the individual components in the system are completely untrusted.

Recently, the necessity of this supposition has been called into question. The Attested Append-only Memory (A2M) system by Chun et al. [7] showed that a small trusted module in each distributed component can significantly improve system security. In addition to founding this important new research direction, A2M made two key contributions: First, they proposed a particular abstraction for such a module, namely a *trusted log*. Second, they showed specifically that their proposed abstraction could improve the degree of fault tolerance to Byzantine faults in the server components of client-server systems.

Despite our appreciation for this work, we are concerned that distributed-protocol designers may be reluctant to start assuming the availability of such trusted modules. We have two reasons for this concern: First, the abstraction of a trusted log may require more storage space and complexity than researchers are comfortable assuming, particularly for an embedded module inside a potentially hostile component. Second, designers may have difficulty appreciating how broadly applicable a trusted module can be to distributed protocols.

In this paper, we continue the research direction begun by A2M, with an eye toward addressing these two issues. First, we have developed a significantly smaller abstraction: Instead of a trusted log, we propose a *trusted incrementer* (*TrInc*), which is little more than a monotonic counter and a key. Second, we demonstrate a more inclusive set of architectures, running a broader range of protocols, yielding a wider set of benefits: Our architectures include not only client-server systems but also peer-to-peer systems. Our protocols include not only Byzantine-fault-tolerant protocols but also PeerReview [13] and BitTorrent [8]. Our demonstrated benefits include not only improving fault tolerance but also reducing communication overhead and solving an open incentive problem.

We show that TrInc has several benefits over A2M. First, its smaller size and simpler semantics make it easier to deploy, as we demonstrate by implementing it on real, currently available trusted hardware. Second, we observe that TrInc's core functional elements are included in the Trusted Platform Module (TPM) [38] found on many modern PCs, lending credence to the idea that such a component could become widespread. Third, TrInc makes use of a shared symmetric session key among all participants in a protocol instance, which significantly decreases the cryptographic overhead.

The rest of this paper is structured as follows. §2 provides background on the underlying problem addressed by TrInc (and by A2M), as well as a primer on trusted hardware. §3 then presents the design of TrInc, and §4 analyzes its security. §§5, 6, and 7 respectively describe several protocols we modified to use TrInc, our trusted hardware implementation, and our evaluation thereof.

Read 11/25

hmm?

So an append only log?

Still have not said what it does...

Property	Accountability layer		Trusted module	
	PeerReview [13]	Nysiad [14]	A2M [7]	TrInc
No centralized trust	✓*	✓*		
Easy to deploy	✓	✓		✓
Easy to apply to existing protocols	✓	✓†		✓‡
Immediate consistency		✓	✓	✓
No assumptions about protocol's determinism		✓†	✓	✓
No additional online assumptions			✓	✓
Additional communication overhead per protocol message, with witness sets of size W	$O(W^2)$	$O(W^2)$	$O(1)$	$O(1)$

Table 1: Summary of the properties of various equivocation-fighting systems. *While PeerReview and Nysiad do not require centralized trust, they do make use of a PKI. †Nysiad deals with nondeterminism by treating nondeterministic events as inputs; this requires protocol changes for nondeterministic state machines. ‡We found that, although TrInc requires a protocol redesign, the modifications are often localized, and vastly simplify security procedures.

2 Background and Related Work

2.1 Equivocation in distributed systems

Since 1982, it has been known that tolerating f Byzantine faults requires at least $3f + 1$ participants [20]. This stands in marked contrast to the case for f stopping faults, which more intuitively requires $2f + 1$ participants. A key insight behind A2M [7] was the observation that a single property of Byzantine faults is responsible for the difference between these two bounds. That property is *equivocation*, meaning the ability to make conflicting statements to different participants. A2M provides a mechanism that prevents participants from equivocating, thereby improving the fault tolerance of Byzantine protocols to f out of $2f + 1$.

We make the further observation that equivocation is a necessary property for many forms of cheating and fraud, not merely for classical Byzantine faults. For instance, in BitTorrent, recent work [21] has shown an exploit in which a peer can obtain an unfairly high download rate by lying about which chunks of a file it has received. This is equivocation, insofar as the peer acknowledges receiving a chunk from the peer that provided it, but then tells another peer that it does not have the chunk.

The following are three more brief examples:

- In a simultaneous-turn game, one can cheat by observing an opponent's move before making one's own move; this is equivocating about whether one has yet moved.
- In a distributed electronic currency system, one can counterfeit money by equivocating to different payees about whether one has spent a particular bill.
- In an election, the tallier can disrupt the vote by equivocating to a voter and an official about whether the voter's vote was recorded.

In §5.5, we will consider many other cases of malicious behavior that can be interpreted as equivocation.

2.2 Prior solutions to equivocation

Several recent efforts have addressed the problem of Byzantine faults in distributed systems. Although their approaches to the problem are very different, they have all effectively focused on the issue of equivocation. Table 1 summarizes our analysis of their properties.

PeerReview [13] is a system that employs witnesses to collect a tamper-evident record of all messages in a distributed system for subsequent checking against a reference implementation. Unlike the remaining approaches we will discuss, PeerReview does not provide fault tolerance. Instead, it provides eventual fault detection and localization, which the system's designers argue leads to fault deterrence. The tamper-evident record is a distributed collection of logs that are authenticated using hash chains. The purpose of the tamper-evidence is to detect equivocation about the messages recorded in a log. As shown in Table 1, the communication required to collectively manage the tamper-evident message log is quadratic in the size of the witness set.

Nysiad [14] is a mechanism that transforms crash-tolerant distributed systems into Byzantine-fault-tolerant ones. It does this by assigning a set of *guards* (comparable to witnesses) to each host in the system. The guards validate the messages sent by their associated hosts, using replicas of the hosts' execution engines. The potential for equivocation in Nysiad is that the host might send different messages to different guards or order its messages differently for different guards. To deal with this equivocation, the guards gossip among each other to agree on the order and content of messages sent by the host. As shown in Table 1, this gossip requires a count of messages that is quadratic in the number of guards. Relative to PeerReview, Nysiad has the benefit of immediate consistency, rather than eventual detection. Nysiad is also able to handle nondeterministic state machines, but doing so requires protocol changes to treat nondeterministic events as inputs.

Attested Append-only Memory, or A2M [7], is a

trusted module that is embedded in an untrusted machine, for the purpose of improving the fault tolerance of a distributed protocol. The A2M module provides the abstraction of a trusted log, which the machine can append to but not otherwise modify. This limitation prevents the machine from equivocating about whether it performed a particular action at a particular step, because once the action is recorded in the log, it cannot be overwritten. A2M uses cryptography to enforce its properties and to attest the log's contents to other machines. Relative to Nysiad and PeerReview, A2M does not require any additional online communication between machines beyond what is required in the base protocol. Consequently, the communication overhead is merely a constant factor due to the cryptographic attestations that accompany the protocol's messages.

As we will show in §3, TrInc is significantly smaller than A2M, making it easier to deploy. TrInc also has another advantage, namely that its use is less tightly coupled to the distributed protocol than use of A2M is. Specifically, because A2M's trusted log has finite storage, it provides a log-truncation operation, but opportunities to truncate the log may be limited by the protocol. Conversely, message sequencing in the protocol may be constrained by the available space in A2M's log. Perhaps in part to address this concern, A2M considered various implementations in addition to hardware, some of which would likely have plentiful storage for the log. These include a remote service, a software-isolated process, and a memory-isolated virtual machine. By contrast, the protocol modifications required to use TrInc tend to be quite localized. Furthermore, TrInc's use of a shared session key often simplifies the protocol.

2.3 Trusted hardware

There have been many trusted hardware designs that predate both TrInc and A2M. Perhaps most similar to TrInc is the abstraction of *virtual monotonic counters* [34]. These are similar to the four increment-only counters included in the current specification of the TPM [38]. Van Dijk et al. propose an algorithm by which to emulate multiple counters with a single trusted counter [39]. We believe a similar approach could ease TrInc's deployment by requiring fewer physical counters. Further, other systems have been proposed that make use of trusted hardware, such as for securing database systems [26] and auctions [31]. To the best of our knowledge, TrInc is the first trusted component designed to be used in large-scale, distributed systems.

3 TrInc Design

3.1 Design Goals

The fundamental security goal of TrInc is to remove participants' ability to equivocate. Consider the situation in which Mallory wishes to send conflicting messages to Alice and Bob. Common approaches to combating such equivocation require Alice and Bob to communicate with one another [13, 14, 20] or with a third party, so they can learn of the distinct messages sent to each. Unfortunately, this additional communication overhead can become a bottleneck for the overlying system, and constitutes the super-linear number of messages in PeerReview [13].

One goal of TrInc is to therefore minimize both communication overhead and the number of non-faulty participants required. With trusted hardware, it is possible to remove Mallory's ability to equivocate without *any* communication between Alice and Bob [7].

The other broad goal of TrInc is to be practical for distributed systems *today*. To be practical, a trusted component must be *small* so that it is feasible to manufacture and deploy. Arbitrary computation and a large amount of storage are difficult and costly to make tamper-resistant. Further, to be a practical primitive in distributed systems, the trusted component must have an API with which it is easy to build distributed systems.

3.2 Overview

To gain the benefits of TrInc, a user must attach a trusted piece of hardware we call a *trinket* to his computer. Unlike a typical TPM, which must attest to states of the associated computer, the trinket's API depends only on its internal state, so the trinket does not need access to the state of the computer. All it needs is an untrusted channel over which it can receive input and produce output, so even USB is quite sufficient.

When Mallory wishes to send a message m to Alice, she must include an attestation from her trinket that (1) binds m to a certain value of a counter, and (2) ensures Alice that no other message will ever be bound to that value of that counter, even messages sent to other users. A trinket enables such attestation by using a counter that monotonically increases with each new attestation. In this way, once Mallory has bound a message m to a certain counter value c, she will never be able to bind a different message m' to that value.

As we show in our case studies in §5, some protocols benefit from using multiple counters. In theory, anything done with multiple counters can be done with a single counter, but multiple counters allow certain performance optimizations and simplifications, such as assigning semantic meaning to a particular counter value. Furthermore, the user of a trinket may participate in multiple protocols, each requiring its own counter or counters. Therefore, a trinket provides the ability to allocate new counters. However, we must identify each of them uniquely so that a malicious user cannot create a new counter with the same identity as an old counter and thereby attest to a different message with the same counter identity and value.

As a performance optimization, TrInc allows its attestations to be signed with shared symmetric keys, which

vastly improves its performance over using asymmetric cryptography or even secure hashes. To ensure that participants cannot generate arbitrary attestations, the symmetric key is stored in trusted memory, so that users cannot read it directly. Symmetric keys are shared among trinkets using a mechanism that ensures they will not be exposed to untrusted parties.

3.3 Notation

We use the notation $\langle x \rangle_K$ to mean an attestation of x that could only be produced by an entity knowing K. If K is a symmetric key, then this attestation can be verified only by entities that know K; if K is a private key, then this attestation can be verified by anyone, or more accurately anyone who knows the corresponding public key. We use the notation $\{x\}_K$ to mean the value x encrypted with public key K, so that it can only be decrypted by entities knowing the corresponding private key.

3.4 TrInc state

Figure 1 describes the full internal state of a trinket, which we describe in more detail here. Each trinket is endowed by its manufacturer with a unique identity I and a public/private key pair $(K_{\text{pub}}, K_{\text{priv}})$. Typically, I will be the hash of K_{pub}. The manufacturer also includes in the trinket an attestation $\mathcal{A}$ that proves the values I and K_{pub} belong to a valid trusted trinket, and therefore that the corresponding private key is unknown to untrusted parties.

We leave open the question of what form $\mathcal{A}$ will take. This attestation is meant to be evaluated by users, not by trinkets, and so can be of various forms. For instance, it might be a certificate chain leading to a well-known authority trusted to oversee trinket production and ensure their secrets are well kept.

Another element of the trinket's state is the *meta-counter* M. Whenever the trinket creates a new counter, it increments M and gives the new counter identity M. This allows users to create new counters at will, without sacrificing the non-monotonicity of any particular counter. Because M only goes up, once a counter has been created it can never be recreated by a malicious user attempting to reset it.

Yet another element is Q, a limited-size FIFO queue containing the most recent few counter attestations generated by the trinket. It is useful for allowing users to recover from power failures, as we will describe later.

The final part of a trinket's state is an array of counters, not all of which have to be in use at a time. For each in-use counter, the state includes the counter's identity i, its current value c, and its associated key K. The identity i is, as described before, the value of the meta-counter when the counter was created. The value c is initialized to 0 at creation time and cannot go down. The key K contains a symmetric key to use for attestations of this counter; if $K = 0$, attestations will use the private key K_{priv} instead.

Global state:

Notation	Meaning
K_{priv}	Unique private key of this trinket
K_{pub}	Public key corresponding to K_{priv}
I	ID of this trinket, the hash of K_{pub}
$\mathcal{A}$	Attestation of this trinket's validity
M	Meta-counter: the number of counters this trinket has created so far
Q	Limited-size FIFO queue containing the most recent few counter attestations generated by this trinket

Per-counter state:

Notation	Meaning
i	Identity of this counter, i.e., the value of M when it was created
c	Current value of the counter (starts at 0, monotonically non-decreasing)
K	Key to use for attestations, or 0 if K_{priv} should be used instead

Figure 1: State of a trinket

3.5 TrInc API

Figure 2 shows the full API of a trinket, described in more detail in this subsection.

3.5.1 Generating attestations

The core of TrInc's API is `Attest`. `Attest` takes three parameters: i, c', and h. Here, i is the identity of a counter to use, c' is the requested new value for that counter, and h is a hash of the message m to which the user wishes to bind the counter value. `Attest` works as follows:

Algorithm 1 `Attest`(i, c', h, n)

1. Assert that i is the identity of a valid counter.
2. Let c be the value of that counter, and K be the key.
3. Assert no roll-over: $c \leq c'$.
4. If $K \neq 0$, then let $a \leftarrow \langle I, i, c, c', h \rangle_K$; otherwise let $a \leftarrow \langle I, i, c, c', h \rangle_{K_{\text{priv}}}$.
5. Insert a into Q, kicking out oldest value.
6. Update $c \leftarrow c'$.
7. Return a.

Note that `Attest` allows calls with $c' = c$. This is crucial to allowing peers to attest to what their current counter value is without incrementing it. To allow for this while still keeping peers from equivocating, TrInc includes both the prior counter value and the new one. One can easily differentiate attestations intended to learn a trinket's current counter value ($c = c'$) from attestations that bind new messages ($c < c'$).

3.5.2 Verifying attestations

Suppose a user Alice with trinket A wants to send a message to user Bob with trinket B. She first invokes

Function	Operation
`Attest`(i, c', h)	Verifies that i is a valid counter with some value c and key K. Verifies that $c \leq c'$. Creates an attestation $a = \langle \text{COUNTER}, I, i, c, c', h \rangle_K$; if $K = 0$, uses K_{priv} instead of K. Adds a to Q. Sets $c = c'$. Returns a.
`GetCertificate()`	Returns a certificate of this trinket's validity: $(I, K_{\text{pub}}, \mathcal{A})$.
`CheckAttestation`(a, i)	Returns a boolean indicating whether a is the output of invoking `Attest` on a trinket using the same symmetric key as the one associated with counter i.
`CreateCounter()`	Increments M. Creates a new counter with $i = M$, $c = 0$, and $K = 0$. Returns i.
`FreeCounter`(i)	If i is the identity of a valid counter, deletes that counter.
`ImportSymmetricKey`$(\mathcal{S}, i)$	Verifies that $\mathcal{S}$ is an encrypted symmetric key decryptable with K_{priv}. Decrypts it and installs the included key as K for counter i.
`GetRecentAttestations()`	Returns Q.

Figure 2: API of a trinket

`Attest` on her trinket using the message's hash, and thereby obtains an attestation a. Next, she sends the message to Bob along with this attestation. However, for Bob to accept this message, he needs to be convinced that the attestation was created by a valid trinket. There are two cases to consider: first, that the attestation used A's private key K^A_{priv}, and second, that the attestation used a shared symmetric key K.

In the first case, the API call `GetCertificate` will be useful. This call returns a certificate C of the form $(I, K_{\text{pub}}, \mathcal{A})$, where I is the trinket's identity, K_{pub} is its public key, and $\mathcal{A}$ is an attestation that I and K_{pub} belong to a valid trinket. Alice can call this API routine and send the resulting certificate C^A to Bob. Bob can then (a) learn Alice's public key K^A_{pub}, and (b) verify that this is a valid trinket's public key. After this, he can verify the attestation Alice attached to her message, and any future attestations she attaches to messages.

In the second case, the API call `CheckAttestation` is useful. When `CheckAttestation`(a, i) is invoked on a trinket, the trinket checks whether a is the output of invoking `Attest` on a trinket using the same symmetric key as the one associated with the local counter i. It returns a boolean indicating whether this is so. So, if Alice sends Bob an attestation signed with a shared symmetric key, Bob can invoke `CheckAttestation` on his trinket to learn whether the attestation is valid.

3.5.3 Allocating counters

Since a trinket may contain many counters, another important component of TrInc's API is the creation of these counters. TrInc creates new logical counters, and allows counters to be deleted, but never resets an existing counter. Logical counters are identified by a unique ID, generated using a non-deletable, monotonic *meta-counter* M. Every trinket has precisely one meta-counter, and when it expires, the trinket can no longer be used; we compensate for this by making M 64 bits, only incrementing M, and assigning no semantic meaning to M's value. TrInc exports a `CreateCounter` function that increments M; allocates a new counter with identity $i = M$, initial value 0, and initial key $K = 0$; and returns this new identity i. When the user no longer needs the counter, she may call `FreeCounter` to free it and thereby provide space in the trinket for a new counter.

3.5.4 Using symmetric keys

TrInc allows its attestations to be signed with shared symmetric keys, which vastly improves its performance over using asymmetric cryptography or even secure hashes. When a set of users are willing to use a single symmetric key for a certain purpose, we call this a *session*. Creating a session requires a *session administrator*, a user trusted by all participants to create a session key and keep it safe, i.e., to not reveal it to any untrusted parties.

To create a session, the session administrator simply generates a random, fresh symmetric key as the session key K. To allow a certain user to join the session, he asks that user for his trinket's certificate C. If the session administrator is satisfied that the certificate represents a valid trinket, he encrypts the key in a way that ensures it can only be decrypted by that trinket. Specifically, he creates $\{\text{KEY}, K\}_{K_{\text{pub}}}$, where K_{pub} is the public key in C. He then sends this encrypted session key to the user who wants to join the session.

Upon receipt of an encrypted session key, the user can join one of his counters to the session by using the API call `ImportSymmetricKey`$(\mathcal{S}, i)$. This call checks that $\mathcal{S}$ is a valid encrypted symmetric key, meant to be decrypted by the local private key. If so, it decrypts the session key and installs it as K for local counter i. From this point forward, attestations for this counter will use the symmetric key. Also, the user will be able to verify any trinket's attestation a using this symmetric key by invoking `CheckAttestation`(a, i).

3.5.5 Handling power failures

One practical concern is that of power failure. Unlike A2M, TrInc users need not query the trusted hardware to

obtain attestations. Instead, TrInc relies on the application (or a TrInc driver) to store attestations in untrusted, persistent storage. If there is a power failure between the time that the trinket advances its counter and the application writes it to disk, then the attestation is lost. This can be problematic for many protocols, which rely on the user being able to attest to a message with a particular counter value. For instance, if Charlie cannot produce an attestation for counter value v, Alice may suspect this is because Charlie has already told Bob about some message m associated with that counter value. Not wanting to be fooled about the absence of such a message, Alice may lose all willingness to trust Charlie.

To alleviate this, a trinket includes a queue Q containing the most recent attestations it has created. To limit the storage requirements, this queue only holds a certain fixed number k of entries, perhaps 10. In the event of a power failure, after recovery the user can invoke the API call `GetRecentAttestations` to retrieve the contents of Q. Thus, all a user must do to protect against power failure is make sure she writes a needed attestation to disk before she makes her kth next attestation request. As long as k is at least 1, the user can safely use the trinket for any application. Higher values of k are useful as a performance optimization, allowing greater pipelining between writing to disk and submitting attestations.

So far we have only discussed a power failure affecting the user, but a power failure can also affect the trinket. The `Attest` algorithm ensures that the attestation is inserted into the queue before the counter is updated, so the trinket cannot enter a situation where the counter has been updated but the attestation is unavailable. It can, however, enter the dangerous situation in which the attestation is in Q, and thus available to the user, but the counter has not been incremented. This window of vulnerability could potentially be exploited by a user to generate multiple attestations for the same counter value, if he could arrange to shut off power at precisely this intervening time. However, we guard against this case by having the trinket check Q whenever it starts up. At startup, before handling any requests, it checks all attestations in Q and removes any that refer to counter values beyond the current one.

3.5.6 A TrInc by any other name

The computational demands of a trinket are small. It must be able to do simple operations such as comparison, as well as cryptographic operations including hashing and both symmetric and asymmetric encryption and decryption. Such cryptographic operations are standard in trusted components such as the TPM [38]. However, we recognize that hardware manufacturers and users are often highly cost-conscious and may be willing to do without performance optimization to save hardware costs.

Therefore, we propose three versions of TrInc that make different trade-offs between cost and performance, summarized in Table 2. The bronze version simply offers correctness with no performance optimizations, by leaving out the ability to use symmetric keys. The silver version is as we have described it. The gold version adds one additional optimization: the use of fast persistent memory such as battery-backed RAM. This optimization makes attestations especially fast since they need not incur the cost of writing to the slow flash memory often found in modern TPMs.

	Persistent Memory	Asym. Crypto	Symm. Crypto	Fast Memory
Bronze TrInc	✓	✓		
Silver TrInc	✓	✓	✓	
Gold TrInc	✓	✓	✓	✓

Table 2: Versions of TrInc with different performance.

3.6 Local adversaries

Mutually distrusting principals on a single computer will share access to a single trinket, creating the potential for conflict between them. Although they cannot equivocate to remote parties, they can hurt each other. They can impersonate each other by using the same counter, and they can deny service to each other by exhausting shared resources within the trinket. Resource exhaustion attacks include allocating all available counters, submitting requests at a high rate, and rapidly filling the queue Q to prevent the pipelining performance optimization.

The operating system can solve this problem by mediating access to the trinket, just as it mediates access to other devices. In this way, the OS can prevent a principal from using counters allocated to other principals, and can use rate limiting and quotas to prevent resource exhaustion. Developing a detailed API and policy for such mediation is beyond the scope of this paper, and is left for future work. However, note that a remote party need not care about how or whether such local mediation is done. Equivocation to remote parties is impossible, even if an adversary has root access to the machine, since cryptography allows the trinket to communicate securely even over an untrusted channel.

4 Analysis of TrInc

We now present a brief discussion of why TrInc is sufficient for a broad class of distributed protocols and why it is nearly minimal in size.

4.1 Equivocation

When a trinket creates an attestation with distinct old and new counter values of c and c', we say that attestation *covers* the half-open interval $(c, c']$. TrInc prevents equivocation by ensuring that no two attestations will cover overlapping intervals. This property could be violated only if:

- the counter is decremented,
- the cryptosystem is broken,

- more than one counter has the same identity, or
- more than one trinket has the same identifier.

By construction, it is not possible to decrement the counter nor to assign the same identity to multiple counters. By hypothesis, cryptographic primitives are effectively unbreakable. Finally, no two trinkets will be created with the same identifier, at least not by a trusted manufacturer; recall that users can verify whether the trinket comes from a trusted manufacturer by observing the certificate chain in $\mathcal{A}$.

4.2 Timeliness

When a trinket creates an attestation with the same old and new counter values, there is no change to the trinket's state; however, the attestation demonstrates the current value of the counter. Thus, if a machine attests to a value of a remotely supplied nonce, the remote machine can be certain that the attestation was generated after the nonce was supplied. Since this attestation carries the current counter value, the remote machine can thus also be sure that the local machine's counter is no lower than this value.

Therefore, when the local machine provides attestations of counter values up to the nonce-attested value, the remote machine can be certain that these attestations are timely.

4.3 Minimality

Suppose, during the execution of a protocol, a participant sends n messages requiring attestation, but her attesting module has fewer than $\log_2(n)$ bits of storage. The attesting module must be willing to provide all n attestations, or else it will cause the protocol to halt prematurely. However, since the module can be in fewer than n distinct states, by the pigeonhole principle it must be willing to attest to two different messages while in the same state. Since this state is as it was before the first message, it cannot reflect the trinket's having attested to the first message. This means a malicious user could take advantage of the trinket's inability to remember its first attestation when requesting the second attestation, and thereby obtain an attestation inconsistent with the earlier one. This is clearly inconsistent with the goals of a trusted module, so we come to a contradiction, and conclude that such a module requires at least $\log_2(n)$ bits of storage. In other words, it needs sufficient storage to accommodate a message counter.

Furthermore, an attesting module needs for its attestations to be unforgeable. Otherwise, the user could generate attestations without using the module, and thereby attest to both sides of an equivocation. TrInc achieves this unforgeability with simple cryptographic primitives.

In summary, the core components of TrInc, a counter and cryptography, seem to be essential for equivocation prevention.

5 Designing Systems with TrInc

5.1 Overview

When designing a protocol that incorporates TrInc, we find it important to address the following questions:

5.1.1 What does TrInc's counter represent?

In the applications we have considered, TrInc's counter represents a natural "progression" of the system. In BitTorrent, for instance, the counter represents the number of blocks a given peer has received, a value which is naturally monotonically increasing. In Byzantine Fault Tolerance (BFT), the counter represents which view a replica is in. Ultimately, the choice of what the counter represents is dependent on what data peers will need to attest to.

5.1.2 To what data do peers attest?

There are two broad types of attestations that TrInc offers. *Advance* attestations increase the trinket's counter, thus binding a message to a counter. *Status* attestations attest to the current counter without advancing it.

Advance attestations Advance attestations are largely protocol-dependent, including such elements as the set of pieces received in BitTorrent, or the root of a Merkle tree of file hashes in a file server. The specific data to which to attest often requires a careful analysis of the selfish or malicious ways in which peers could equivocate. It is important to ensure that the impossibility of equivocating about what was assigned to a particular counter value translates into the impossibility of equivocating at the higher desired semantic level.

For instance, suppose an attestation consists solely of a number n of pieces received in BitTorrent and a list of n peers. In this case, a participant Mallory can cheat in the following way. After receiving the first piece a from Alice, she replies with an attestation that her one-piece set contains only a. Next, after receiving her next two pieces b from Bob and c from Charlie, she sends them both an identical attestation that her two-piece set is b and c. In this way, Mallory gets away with hiding the fact that she has received piece a, despite not being able to get different attestations for the same value of $n = 2$. As we will see later, in §5.4, we prevent this by having an attestation include the last piece received.

Status attestations Most distributed systems do not have an implicit system-wide "counter." Rather, peers progress at varying rates: BitTorrent peers download at rates largely dependent on their own upload rates, DHT peers store varying amounts of data, and so on. Status attestations enable peers to determine others' current counter values. The data in a status attestation is generally a nonce, to ensure freshness in peers' reports of their counters. Coupled with a counter that has semantic meaning, status attestations can provide peers with up-to-date information about their neighbors. In BitTorrent, for instance, knowing how much of a file a neighbor has downloaded can help determine whether to bootstrap him

Algorithm 2 Implementation of A2M with TrInc

Init()

1. Create low and high counters:
$$\mathcal{L}_q \leftarrow \texttt{CreateCounter}(); \ \mathcal{H}_q \leftarrow \texttt{CreateCounter}()$$
2. Return $\{\mathcal{L}_q, \mathcal{H}_q\}$

Append(queue q, value x)

1. Bind $h(x)$ to a unique counter (the current "high counter"):
$$a \leftarrow \texttt{Attest}(\mathcal{H}_q.\text{id}, \ \mathcal{H}_q.\text{ctr} + 1, \ h(x))$$
2. Store the attestation in untrusted memory:
$$q.\texttt{append}(a, x)$$

Lookup(queue q, sequence number n, nonce z)

1. If $n < \mathcal{L}_q$, the entry was truncated. Attest to this by returning an attestation of the supplied nonce using the low-counter:
$$\texttt{Attest}(\mathcal{L}_q.\text{id}, \ \mathcal{L}_q.\text{ctr}, \ h(\text{FORGOTTEN}||z))$$
2. If $n > \mathcal{H}_q$, the query is too early. Attest to this by returning an attestation of the supplied nonce using the high-counter:
$$\texttt{Attest}(\mathcal{H}_q.\text{id}, \ \mathcal{H}_q.\text{ctr}, \ h(\text{TOOEARLY}||z))$$
3. Otherwise, return the entry in q that spans n, i.e., the one such that $a.c < n \leq a.c'$. Note that if $n < a.c'$, this means n was skipped by an `Advance`.

End(queue q, sequence number n, nonce z)

1. Retrieve the latest entry from the given log:
$$\{a, x\} \leftarrow q.\texttt{end}()$$
2. Attest that this is the latest entry with a high-counter attestation of the supplied nonce:
$$a' \leftarrow \texttt{Attest}(\mathcal{H}_q.\text{id}, \ \mathcal{H}_q.\text{ctr}, \ z)$$
3. Return $\{a', \{a, x\}\}$

Truncate(queue q, sequence number n)

1. Remove the entries from untrusted memory:
$$q.\texttt{truncate}(n)$$
2. Move up the low counter:
$$a \leftarrow \texttt{Attest}(\mathcal{L}_q.\text{id}, \ n, \ \text{FORGOTTEN})$$

Advance(queue q, sequence number n, value x)

1. Append a new item with sequence number n:
$$a \leftarrow \texttt{Attest}(\mathcal{H}_q.\text{id}, \ n, \ h(x))$$
2. Store the attestation in untrusted memory:
$$q.\texttt{append}(a, x)$$

with free pieces (because he is new to the swarm) or to initiate a trade with him (because he has many interesting pieces of the file).

5.2 Case study 1: A2M

Attested Append-only Memory (A2M) [7] is another proposed trusted hardware design with the intent of combating equivocation. A2M offers *trusted logs*, to which users can only append. The fundamental difference between the designs of A2M and TrInc are in the amount of state and computation required from the trusted hardware. To demonstrate that TrInc's decreased complexity is enough, we present, as our first case study, how to build A2M using TrInc.

5.2.1 A2M overview

A2M's state consists of a set of logs, each containing entries with monotonically increasing sequence numbers. A2M supports operations to add (`append` and `advance`), retrieve (`lookup` and `end`), and delete (`truncate`) items from its logs. The basis of A2M's resilience to equivocation is `append`, which binds a message to a unique sequence number. For each log q, A2M stores the lowest sequence number, $\mathcal{L}_q$, and the highest sequence number, $\mathcal{H}_q$, stored in q. A2M appends an entry to log q by incrementing the sequence number $\mathcal{H}_q$ and setting the new entry's sequence number to be this incremented value. The low and high sequence numbers allow A2M to attest to failed lookups; for instance, if a user requests an item with sequence number $s > \mathcal{H}_q$, A2M returns an attestation of $\mathcal{H}_q$.

5.2.2 Trusted logs with TrInc

In our TrInc-based design of A2M, we store logs in untrusted memory, as opposed to within a trinket. As in A2M, we make use of two counters per log, representing the highest ($\mathcal{H}_q$) and lowest ($\mathcal{L}_q$) sequence number in the respective log q.

We present the detailed protocol in Algorithm 2, and summarize some of its characteristics here. Note the power of TrInc's simple API; our design is built predominately on calls to a trinket's `Attest` function. Our protocol uses advance attestations for moving the high sequence number when appending to the log, and for moving the low sequence number when deleting from the log. We perform status attestations of the low counter value to attest to failed lookups, and of the high counter to attest to the end of the log. No additional attestations are necessary for a successful `lookup`, even if the `lookup` is to a skipped entry. Conversely, A2M requires calls to the trusted hardware even for successful lookups.

5.2.3 Properties of a TrInc-based A2M

Chun et al. [7] demonstrate how to apply A2M to BFT [20], SUNDR [22], and Q/U [1]. Our implementation of A2M in TrInc demonstrates that TrInc, too, can be applied to these systems.

Implementing trusted logs using TrInc has several benefits over a completely in-hardware design like A2M. Because TrInc stores the logs in untrusted storage, we decouple the usage demand of the trusted log from the amount of available trusted storage. Conversely, limited by the amount of trusted storage, A2M must make

more frequent calls to `truncate` to keep the logs small. Some systems, such as PeerReview [13], benefit from large logs, making TrInc a more suitable addition, which we consider next.

5.3 Case study 2: PeerReview

Accountability systems, such as PeerReview [13] and Nysiad [14], strive to augment existing protocols to make them tolerant to Byzantine faults. This is a powerful approach, as it allows system designers to focus on the system at hand, rather than consider Byzantine faults at all layers of the system. The general approach is to have participants in the system communicate with and audit one another, resulting in what is sometimes, unfortunately, a massive amount of additional communication overhead.

Our main observation in this case study is that the means by which these systems combat equivocation constitutes the bulk of their communication overhead. By applying TrInc to PeerReview, we are able to vastly reduce PeerReview's communication overhead.

5.3.1 PeerReview review

PeerReview [13] is a system that enables accountability in general distributed protocols. Unlike BFT, which ensures that bad behavior never has an effect, PeerReview allows bad behavior to affect the system but ensures that the improper act will eventually be detected. This allows a system to correct for bad behavior after the fact, and also deters bad behavior to begin with.

PeerReview works on any protocol in which each participant acts according to a deterministic state machine. PeerReview assigns each participant a set of *witnesses*, machines whose job it is to detect bad behavior by that participant. The participant is required to log all of the messages it sends and receives, and report these to the witnesses. The witnesses then run the participant's state machine to ensure the participant's outgoing messages were consistent with proper operation.

A participant might try to cheat by sending different messages to the witnesses than it sends to other participants. For this reason, when a participant receives a message from another, it forwards this message to the sender's witnesses, so they can ensure this message actually appears in the sender's log.

As a practical matter, full messages do not have to be transmitted to witnesses thanks to the use of a *tamper-evident log*. Each log entry is associated with a sequence number, and the log itself is represented by a recursive hash reflecting all log entries. When a participant sends a message, it includes a signed statement that this message has a particular sequence number and that the log had a particular recursive hash when this message was logged. In this way, the receiver only needs to report this authenticator to the witness.

PeerReview's tamper-evident log has another important use. When a participant or witness discovers bad behavior in a participant, the authenticators signed by the malefactor stand as clear proof of the misbehavior. Thus, a faulty witness cannot improperly accuse a participant, and an incompletely trusted witness can be believed when it presents evidence of a participant's misbehavior.

5.3.2 Simplifying PeerReview with TrInc

By augmenting PeerReview with TrInc, we are able to simplify much of PeerReview's protocol. We detail here the modifications we make to PeerReview in augmenting it with TrInc.

Trusted logs As demonstrated with A2M, TrInc can easily supply a trusted log without the assistance of a witness set. Our first modification is to include such a trusted log. Whenever a participant sends or receives a message, it logs that message with an attestation from its trinket. A participant should only process a received message if it is accompanied by an attestation that the message has been logged by the sender's trinket.

Audits Each witness w for a participant p keeps track of n, a log sequence number, and s, the state that p should have been in after sending or receiving the message in log entry n. It initializes n to 0 and s to the initial state of participant p.

Whenever w wants to audit p, it sends it n and a nonce. The participant returns an attestation of its current log entry number n' using the nonce, and also returns a log entry and attestation for every index i such that $n < i \leq n'$. Note that witnesses need only obtain these entries directly from p, and not from other peers with whom p has communicated. The witness then runs the reference implementation, starting at state s, and progressing through the log entries between n and n'. If the reference implementation sends the same messages that are in the log, then the witness simply updates n to n' and updates s to the state of the reference implementation at that point. If not, then the witness has proof it can present of the participant's failure to act properly.

5.3.3 Properties of a TrInc-enabled PeerReview

The benefits from applying TrInc to PeerReview are evident when considering what the protocol *no longer has to do*.

Challenge/response Enabled with TrInc, PeerReview's challenge/response protocol is no longer needed for a participant to verify a hash chain of log entries. The fact that TrInc signs the messages is sufficient. The only time a participant i has to challenge another participant j is when it sends participant j a message and receives no acknowledgment of it. In this case, the challenge works as in regular PeerReview.

Consistency TrInc further removes the need for witness-to-witness communication. In PeerReview, if p receives an authenticator from q, then p's witnesses must forward it to q's witnesses. This is not necessary in a TrInc-augmented PeerReview because there would be no way for those other participants to avoid sending the au-

thenticators themselves to their witnesses. Another way to look at it is that it is not necessary for a participant to pass on authenticators it receives to witnesses, so it is not necessary for a witness to do this on behalf of participants.

To summarize, we find that by applying TrInc to PeerReview, we are able to vastly decrease the amount of communication overhead. We demonstrate this empirically in Section 7.

5.4 Case study 3: BitTorrent

The previous two systems demonstrate that TrInc is a minimal counterpart to a related trusted component, and that it can reduce the overhead of achieving accountability in a distributed setting. Our third case study demonstrates TrInc's versatility. We show how TrInc can be applied to solving an open incentive problem [21] in the immensely popular BitTorrent system [8].

5.4.1 A brief overview of BitTorrent

BitTorrent [8] is a decentralized file swarming system whose goal is to disseminate large files to a large number of downloaders. Rather than rely on a highly provisioned server, BitTorrent peers trade small *pieces* of a file with one another, thereby contributing to the system while gaining from it. *Bitfields* represent which pieces of a file a peer has. Peers trade bitfields in order to gain one another's interest: a peer is *interested* in peers who have pieces that it does not. Since peers only upload to peers in whom they are interested, peers have incentive to be as interesting to as many others as possible.

5.4.2 Piece under-reporting

BitTorrent peers can sometimes have incentive to under-report what pieces they have to their neighbors, since by doing so they can limit the degree to which their neighbors find interest in one another [21]. For instance, suppose peer i has neighbors j and k, both of whom want pieces p and q from i. If i were to tell them both about both pieces, one might demand p and the other might demand q. After obtaining them, they might gain interest in one another and exchange p and q among themselves, thus *decoupling* from i. Thus, i may prefer to under-report by sending to j and k a bitfield that contains p but not q. As a result, both neighbors request and obtain p, gaining no interest in one another; only then does i reveal that he also has piece q, forcing j and k to download it from i.

Such under-reporting leads to a tragedy of the commons, since although strategic under-reporters' download times improve, the system as a whole suffers [21]. Since its recent discovery, strategic under-reporting has yet to be solved; we demonstrate how to solve it with TrInc.

5.4.3 Solving under-reporting with TrInc

We observe that *under-reporting in file swarming systems is an act of equivocation*. Using the above example, when peer i received piece q from peer ℓ, i must have sent an acknowledgment, stating to ℓ that he received the piece. However, by under-reporting q to peers j and k, i is effectively contradicting a statement he made earlier to ℓ.

Algorithm 3 Fighting equivocation in BitTorrent

Upon receipt of piece p:
1. Add p to bitfield B
2. $a_{curr} \leftarrow \texttt{Attest}(i, |B|, h(p, B))$

Upon sending piece p to neighbor j:
1. Request an attestation from j with a random nonce.
2. Do not send any piece other than p to j until j admits to having p.

Periodically, for each neighbor j:
1. Request an attestation of j's current bitfield with a random nonce.

Upon receiving an attestation request with nonce z:
1. $a_{tmp} \leftarrow \texttt{Attest}(i, |B|, z)$.
2. Reply with (a_{curr}, a_{tmp}).

Our goal is therefore to remove BitTorrent peers' ability to undetectably equivocate. We present in Algorithm 3 a TrInc-based protocol for fighting equivocation in BitTorrent. In this protocol, a peer attests to his bitfield, incrementing a trinket counter for each piece he receives. Also, peers periodically request up-to-date attestations from their neighbors, to maintain fresh state.

Because they join the swarm at different times and download at different rates, peers' counters are not synchronized. In Algorithm 3, the TrInc counter does not correspond to some system-wide "round" the protocol is in, as it would in, say, BFT machine replication. Instead, peer i's counter represents how many pieces i has downloaded. This is a natural fit for the counter, because it is a monotonically increasing number, and because the type of malicious behavior we want to prevent corresponds to pretending it is not monotonic.

Algorithm 3 demonstrates the importance of choosing the correct data to which to attest. Suppose, for instance, peers were to attest *only* to their bitfields. Clearly, when s sends an attested bitfield to neighbor n, s must include the piece n sent him, p_n, in the bitfield, otherwise n will observe an under-report. Were s to attest only to the bitfield, then s could under-report as follows, where B_{old} represents the bitfield before receiving pieces p_a, p_b, and p_c, and $\oplus$ denotes adding a piece to the bitfield:

- To a: $B_{old} \oplus p_a$
- To b and c: $B_{old} \oplus p_b \oplus p_c$

The problem arises because the data to which s is attesting does not enforce monotonicity at the semantic level we desire. Specifically, though the counter cannot decrease, it does not have to correspond to the number of

distinct pieces acknowledged, allowing a malicious participant to misstate the number of distinct pieces he has acknowledged.

In our solution, a peer attests not only to the hash of his bitfield B, but also to the most recent piece he has received, p. Neighbor n therefore expects an advance attestation including both p_n and a bitfield containing p_n. As a result, every piece must have a unique advance attestation, ensuring that s's counter must be as large as the number of pieces he has acknowledged receiving.

5.4.4 Properties of a TrInc-augmented BitTorrent

Our TrInc-based solution to equivocation in BitTorrent solves two difficult incentives-related problems. First, peers have incentive to truthfully reveal the pieces they have whenever they are asked to. TrInc removes the ability to equivocate, and step-omission failures (remaining silent) result in getting no further pieces from a neighbor. Peers can therefore obtain long-lived trades with others only by truthfully reporting their pieces.

Second, our solution adds additional security to BitTorrent's bootstrapping mechanism. In BitTorrent, peers *optimistically unchoke* new participants, sending them pieces without requiring anything in return, to introduce them into the system. BitThief [24] exploits this by pretending not to be able to make progress [35]. However, such artifice is not possible with TrInc since with it a peer cannot hide the rate at which he is downloading pieces.

Note, however, that what we propose is not a complete solution to problems with bootstrapping. Even with TrInc-enabled BitTorrent, a peer can steal a single piece from each other peer. Our goal of applying TrInc here is to ensure truthfulness in long-lived peerings, which (surprisingly) does not arise automatically.

5.5 Other applications

We see many other potential applications for TrInc. We briefly described three such apps in Section 2.1: simultaneous-turn games, electronic currency, and elections. Here, we detail several others:

Secure DNS is intended to protect the integrity of the Internet domain name system. One identified threat [6] is that a resolving name server could be compromised and forge incorrect responses. The official solution to this threat is *data origin identification* in the DNS Security Extensions (DNSSEC), which uses public-key signatures to authenticate name updates. However, this solution does not address a threat in which the compromised name server replies to a query with out-of-date data, which would still bear a valid signature. Modifying DNSSEC with TrInc could address this problem by preventing the resolving name server from equivocating about whether it has received an update. Once it acknowledges receipt to the authoritative name server, it can no longer pretend it has not received the update.

Secure Origin BGP (soBGP) [44] is intended to protect the integrity of Internet routing updates. Like DNSSEC, soBGP uses public-key signatures to authenticate updates. Also like DNSSEC, soBGP is vulnerable to a threat in which a compromised router advertises out-of-date routes, which would still bear valid signatures. TrInc could address this problem by preventing a router from equivocating about whether it has received a routing update.

Distributed hash tables (DHTs), such as Chord [37], Bamboo [33], and Kademlia [27], are vulnerable to misbehaving nodes. In particular, a node can lie about which region of the keyspace it is responsible for. As nodes join and leave the DHT, these regions of responsibility change (sometimes quite rapidly [33]) in response to reconfiguration messages. A node can equivocate about whether it has received a particular message, which may allow it to claim responsibility for a region of the keyspace it does not own. TrInc could be used to prevent this equivocation.

Version control systems, such as CVS [41] and Subversion [29] are often run on remote servers. Thus, they are vulnerable to a threat model in which the server presents different views of the repository to different clients. Although this threat could be addressed at the block-store level [22], it might be more efficient to address it at the application level, in which case TrInc could prevent this equivocation.

Distributed auctions [42] are vulnerable to cheating participants. A bidder can try to manipulate others' bids by equivocating about the value of his current bid. An auctioneer can try to manipulate the bidding by equivocating about her reserve price for a particular auction. TrInc could protect against both of these classes of cheating, by preventing both bidders and auctioneers from equivocating.

Leader election protocols [25] rely on a quorum of participants to agree on a choice of leader. For a quorum of size q, it can legitimately happen that two groups of size $q - 1$ will nominate different leaders. In this case, one participant can equivocate about which leader to nominate, causing the protocol to select two leaders concurrently. TrInc could be used to prevent this equivocation.

Digital signatures are used in many cryptographic protocols, but commonly use slow asymmetric key operations [17]. However, TrInc allows faster symmetric key operations to be used instead. To do so, a signer merely has to have his trinket attest to the hash of the message to be signed using a shared symmetric key. Since this attestation can only be generated by a party with access to the symmetric key, and since the hardware includes the ID in any attestation, no other party (except the trusted session administrator) can have generated the attestation. Thus, it functions effectively as a digital signature, verifiable by anyone whose trinket has the same symmetric key installed.

Operation		Time (msec)
Noop		6.14 ± 0.15
Attest	(asymmetric, advance > 0)	230.24 ± 0.28
	(asymmetric, advance $= 0$)	198.21 ± 0.10
	(symmetric, advance > 0)	128.95 ± 0.08
	(symmetric, advance $= 0$)	105.90 ± 0.08
Verify Symmetric Attestation		85.81 ± 0.11

Table 3: TrInc microbenchmarks on a Gemalto .NET Smartcard, with 95% confidence intervals.

Operation	Time (msec) TrInc	A2M
Noop	6.99 ± 0.01	
Append	187.60 ± 0.15	551.93 ± 154
Lookup (Successful)	0.0122 ± 0.02	304.14 ± 6.87
Lookup (TooEarly)	162.24 ± 0.08	289.68 ± 2.23
Lookup (Forgotten)	162.35 ± 0.10	350.51 ± 1.43
End	162.31 ± 0.11	294.16 ± 2.04
Truncate	187.94 ± 0.10	28.99 ± 0.02
Advance	187.81 ± 0.12	288.20 ± 11.4

Table 4: TrInc-A2M microbenchmarks, with 95% confidence intervals.

6 TrInc Implementation

The application case studies demonstrate the strong theoretical properties of TrIncs. In this section, we study the performance of TrIncs *today*. To this end, we have implemented TrInc on Gemalto .NET SmartCards [11], and present microbenchmarks that measure TrInc's performance on these widely available pieces of trusted hardware.

6.1 Microbenchmarks

Our experimental setup consists of an Intel Core 2 Duo 1.6GHz machine with 3GB of RAM, and a smartcard connected via a USB card reader. We present our microbenchmarks in Table 3, with results averaged over 1,000 runs. In addition to TrInc's API, we include a noop to essentially measure the round-trip time between PC and smartcard.

Compare the `Attest` results on the card to those on the untrusted PC, where 3-DES took 0.017 ± 0.008 msec, and RSA took 8.6 ± 0.67 msec. It is no surprise that a smartcard does not perform as well, but the difference in relative performance between symmetric and asymmetric encryption is striking. On the PC, they differ by a factor of over 500, while on the card they differ by less than a factor of 2. While using symmetric instead of asymmetric operations improves TrInc's performance, we were surprised to see it was by this small a factor.

6.2 Why so slow?

The conclusion is clear: today's trusted hardware is *slow*! Indeed, it is much slower than would be allowed by most components of a distributed system. But why is it slow, and why do current applications that use trusted hardware not suffer as a result?

We believe this is attributable to the fact that *TrInc uses trusted hardware in a fundamentally different way than that for which the hardware is currently designed.* Today's trusted hardware is designed to bootstrap software, generally performing few operations during a machine's boot cycle. Conversely, TrInc makes use of trusted hardware *during* operation, in some cases multiple times for each message sent.

We proposed several versions in §3.5.6 that we believe would be viable directions for future designs of trusted hardware to take. In the interim, a logical solution is to design protocols that limit the number of necessary attestations, but such approaches are beyond the scope of this paper. Nevertheless, our empirical results in the following section indicate that making trusted hardware more suitable for use in distributed systems *today* is a valuable area of future work.

7 Application Evaluation

We now turn to macrobenchmarks, evaluating TrInc as it applies to our three case studies: A2M, PeerReview, and BitTorrent.

7.1 TrInc-A2M

In Section 5.2, we proposed a way to build A2M using TrInc. While demonstrating TrInc's ease of use and versatility, it also allows us to compare the two trusted-component designs. To this end, we have implemented A2M in the Gemalto .NET SmartCard, and a TrInc library—run on an untrusted machine—that accesses TrInc as prescribed in Algorithm 2.

We present microbenchmark comparisons in Table 4. As expected, TrInc performs `Appends` much more quickly, as it does not require as many writes to trusted storage. Where TrInc offers vast speed improvements over A2M is in successful `Lookups`; since these do not have to be either stored in trusted hardware or attested, they are merely local operations. Interestingly, A2M improves with `Truncate`, since A2M simply increases the log's low counter and postpones the attestation of the operation until a lookup that needs to return FORGOTTEN. TrInc amortizes this cost, in the expectation that there will be more FORGOTTEN lookups than truncations.

These results demonstrate that TrInc performs better on *today's* trusted hardware. As trusted components improve, particularly in terms of memory writes and cryptographic operations, it is likely that A2M and TrInc will perform comparably well. However, the slowness of today's trusted hardware brings to light the difference in complexity between A2M and TrInc. We believe TrInc's relative simplicity makes it a more suitable candidate even with future designs of trusted hardware.

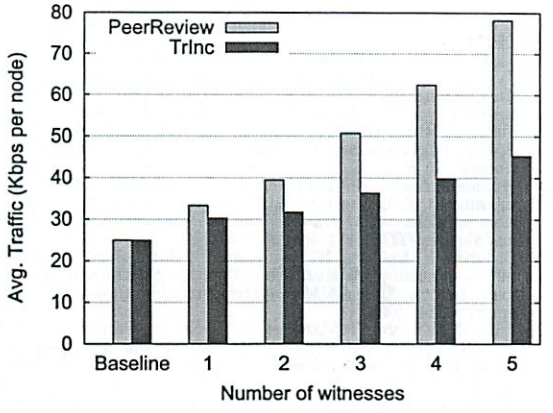


Figure 3: Reduction in PeerReview's message overhead due to TrInc.

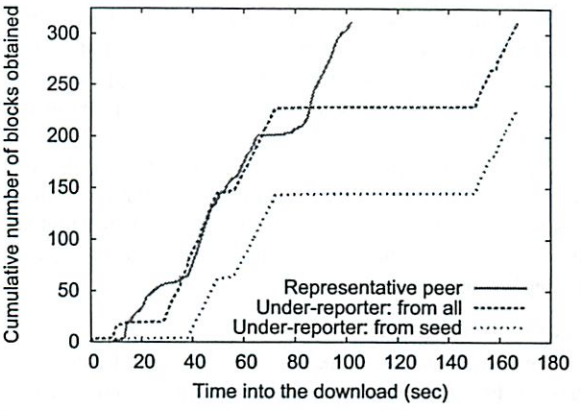


Figure 4: Rate of progress for various BitTorrent clients when TrInc is used.

7.2 TrInc-PeerReview

In Section 5.3, we demonstrated how including TrInc into the design of an accountability system such as PeerReview can decrease the amount of communication required between participants. This represents one of the fundamental strengths of including a small, trusted component into an otherwise untrusted system.

Applying TrInc to PeerReview removes the requirement for a peer p to communicate with the witness set of any other peer q, unless, of course, p happens in q's witness set. Using data from the original PeerReview study [13], we demonstrate in Figure 3 the extent to which TrInc reduces PeerReview's communication overhead. TrInc effectively removes the $O(W^2)$ witness-set-to-witness-set communication, for reasons described in Section 5.3. As a result, the amount of additional communication overhead scales linearly rather than quadratically with the size of the witness sets.

7.3 TrInc-BitTorrent

To evaluate our TrInc-based solution for BitTorrent, we simulated using a "gold-standard" trinket in the Azureus BitTorrent client. To do so, we modified BitTorrent's Have messages to include attestations to counters. We observed that Have messages, originally intended simply to inform others when a peer receives a piece, come frequently enough in practice to also satisfy peers' continual need for fresh attestations.

We modified the BitTorrent code to recognize these new messages, and to cut off peers thereby discovered to be under-reporting. However, we never have the seeder punish a peer in this way. It seems reasonable to have such a forgiving seeder since otherwise peers who suffer failures—for example, from a corrupted disk—could never request blocks after they have attested to them.

We ran our experiments on a local cluster consisting of 23 leechers, each with upload bandwidth capped at 50Kbps, and one seeder, with upload bandwidth capped at 80Kbps. We chose one host to act as a strategic piece revealer using an algorithm from a prior study [21]. We chose this host arbitrarily since, on the local cluster, we found them to be virtually indistinguishable in terms of performance.

Our experiments demonstrated a clear loss in performance from under-reporting. In a representative run, the under-reporting peer took 27% longer to download the file than the other peers did on average, and 33% longer than the median.

The under-reporter's download times would have been much worse if not for the forgiving seeder. We show in Figure 4 the total number of blocks the under-reporter received over time, compared to the number of blocks he received from the seeder. We plot a representative, truthful peer from the swarm as a point of comparison. Because other peers refused to send to the under-reporter until he revealed all the pieces in his possession, the seeder became the under-reporter's only remaining option. Indeed, the under-reporting peer obtained more pieces (73%) from the seeder than any other peer in the swarm (11% on average, 6% median).

These results indicate the power of applying a small amount of trust, and small attestations piggybacked on existing protocol messages, to a large-scale decentralized system.

8 Conclusions

In this paper, we presented TrInc, a simple yet powerful abstraction for improving security in distributed systems. TrInc is a trusted hardware module that holds a non-decreasing counter and a hidden cryptographic key. This combination, along with the computational machinery to support it, yields an abstraction that significantly improves various aspects of security in distributed systems.

TrInc was inspired by the seminal work of A2M, which introduced the idea of a trusted log for improv-

ing system security. Relative to A2M, TrInc has a significantly simpler abstraction: a counter instead of a log. We have also demonstrated a wider range of applications for, and benefits from, a trusted module than previously shown.

We have implemented TrInc on real, currently available trusted hardware. We have performed three detailed case studies of TrInc as applied to different distributed protocols. Our results show that this abstraction is easy to deploy, powerful, and versatile.

Acknowledgments

We would like to thank Josh Benaloh, Paul England, Sandro Forin, Atul Singh, Talha Bin Tariq, and Gideon Yuval for helpful discussions and assistance in evaluating TrInc on real trusted hardware. We also thank Adam Bender, Stefan Saroiu, the anonymous reviewers, and our shepherd, Greg Minshall, for their helpful comments on improving the presentation of this paper. Dave Levin was supported in part by a Microsoft Live Labs fellowship.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 59–74, 2005.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 1–14, 2002.

[3] N. E. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *Proc. Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, pp. 104–113, 2001.

[4] A. Blanc, Y.-K. Liu, and A. Vahdat. Designing incentives for peer-to-peer routing. In *Proc. NetEcon*, 2004.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[6] R. Chandramouli and S. Rose. Secure domain name system (DNS) deployment guide. *Special Publication 800-81, NIST*, 2006.

[7] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 189–204, 2007.

[8] B. Cohen. Incentives build robustness in BitTorrent. In *P2PEcon*, 2003.

[9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 177–190, 2006.

[10] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *NOSSDAV*, 2004.

[11] Gemalto. http://www.gemalto.com/.

[12] R. Gupta and A. K. Somani. CompuP2P: An architecture for sharing of compute power in peer-to-peer networks with selfish nodes. In *Proc. NetEcon*, 2004.

[13] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 175–188, 2007.

[14] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 175–188, 2008.

[15] D. Hughes, G. Coulson, and J. Walkerdine. Free riding on Gnutella revisited: The bell tolls? *IEEE Distributed Systems Online*, 6(6):1–18, 2005.

[16] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pp. 29–42, 2003.

[17] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. CRC Press, 2007.

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 45–58, 2007.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[20] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[21] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. BitTorrent is an auction: Analyzing and improving BitTorrent's incentives. In *Proc. SIGCOMM Conference on Data Communication*, pp. 243–254, 2008.

[22] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 121–136, 2004.

[23] Q. Lian, Y. Peng, M. Yang, Z. Zhang, Y. Dai, and X. Li. Robust incentives via multi-level tit-for-tat. In *Proc. Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[24] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, pp. 85–90, 2006.

[25] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[26] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 135–150, 2000.

[27] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. Workshop on Peer-to-Peer Systems (IPTPS)*, pp. 109–114, 2002.

[28] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 31–44, 2002.

[29] W. Nagel. *Subversion Version Control: Using the Subversion Version Control System in Development Projects*. Prentice Hall, 2005.

[30] T.-W. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *Proc. NetEcon*, 2004.

[31] A. Perrig, S. Smith, D. Song, and J. D. Tygar. SAM: A flexible and secure auction architecture using trusted hardware. In *ICEC*, 2001.

[32] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 1–14, 2007.

[33] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. USENIX Annual Technical Conference*, pp. 127–140, 2004.

[34] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *Proc. Workshop on Scalable Trusted Computing (STC)*, 2006.

[35] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding in BitTorrent networks with the large view exploit. In *Proc. Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[36] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki. Dandelion: Cooperative content distribution with robust incentives. In *Proc. USENIX Annual Technical Conference*, pp. 157–170, 2007.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. SIGCOMM Conference on Data Communication*, pp. 149–160, 2001.

[38] Trusted Computing Group. Trusted Platform Module Specifications. Online at https://www.trustedcomputinggroup.org/specs/TPM/.

[39] M. van Dijk, L. F. G. Sarmenta, C. W. O'Donnell, and S. Devadas. Proof of freshness: How to efficiently use an online single secure clock to secure shared untrusted memory. Technical report, 2006.

[40] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 59–72, 2007.

[41] J. Vesperman. *Essential CVS, 2nd Edition*. O'Reilly, 2006.

[42] J. M. Vidal. Multiagent coordination using a distributed combinatorial auction. In *AAAI Workshop on Auction Mechanisms for Robot Coordination*, 2006.

[43] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A Secure Economic Framework for P2P Resource Sharing. In *Proc. NetEcon*, 2003.

[44] R. White. Securing BGP through Secure Origin BGP. *Internet Protocol Journal*, 6(3), 2003.

[45] S. Zhong, J. Chen, and Y. R. Yang. Sprite: A simple, cheat-proof, credit-based system for mobile ad-hoc networks. In *Proc. Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, 2003.

Algorithm 1: the "n" parameter to Attest should not be there;
the Attest function only takes the first three parameters.

Byzantine fault tolerance

From Wikipedia, the free encyclopedia

Byzantine fault tolerance is a sub-field of fault tolerance research inspired by the **Byzantine Generals' Problem**,[1] which is a generalized version of the Two Generals' Problem.

The object of Byzantine fault tolerance is to be able to defend against *Byzantine failures*, in which components of a system fail in arbitrary ways (i.e., not just by stopping or crashing but by processing requests incorrectly, corrupting their local state, and/or producing incorrect or inconsistent outputs.). Correctly functioning components of a Byzantine fault tolerant system will be able to correctly provide the system's service assuming there are not too many Byzantine faulty components.

Contents

- 1 Byzantine failures
- 2 Origin
- 3 Early solutions
- 4 Practical Byzantine fault tolerance
- 5 See also
- 6 References
- 7 Further reading
- 8 External links

Byzantine failures

A **Byzantine fault** is an arbitrary fault that occurs during the execution of an algorithm by a distributed system. It encompasses both **omission failures** (e.g., crash failures, failing to receive a request, or failing to send a response) and **commission failures** (e.g., processing a request incorrectly, corrupting local state, and/or sending an incorrect or inconsistent response to a request). When a Byzantine failure has occurred, the system may respond in any unpredictable way, unless it is designed to have Byzantine fault tolerance.

For example, if the output of one function is the input of another, then small round-off errors in the first function can produce much larger errors in the second. If the second function were fed into a third, the problem could grow even larger, until the values produced are worthless. Another example is in compiling source code. One minor syntactical error early on in the code can produce large numbers of perceived errors later, as the parser of the compiler gets out-of-phase with the lexical and syntactic information in the source program. Such failures have brought down major Internet services. For example, in 2008 Amazon S3 was brought down for several hours when a single-bit hardware error propagated through the system.[2]

In a Byzantine fault tolerant (BFT) algorithm, steps are taken by processes, the logical abstractions that represent the execution path of the algorithms. A faulty process is one that at some point exhibits any of the above failures. A process that is not faulty is correct.

The Byzantine failure assumption models real-world environments in which computers and networks may

behave in unexpected ways due to hardware failures, network congestion and disconnection, as well as malicious attacks. Byzantine failure-tolerant algorithms must cope with such failures and still satisfy the specifications of the problems they are designed to solve. Such algorithms are commonly characterized by their resilience t, the number of faulty processes with which an algorithm can cope.

Many classic agreement problems, such as the Byzantine Generals' Problem, have no solution unless $n > 3t$, where n is the number of processes in the system. In other words, the algorithm can ensure correct operation only if fewer than one third of the processes are faulty.

Origin

Byzantine refers to the Byzantine Generals' Problem, an agreement problem (first proposed by Marshall Pease, Robert Shostak, and Leslie Lamport in 1980)[3] in which generals of the Byzantine Empire's army must decide unanimously whether to attack some enemy army(The Byzantine Army was chosen as an example for the problem as the Byzantine state experienced frequent treachery among the high levels of its administration). The problem is complicated by the geographic separation of the generals, who must communicate by sending messengers to each other, and by the presence of traitors amongst the generals. These traitors can act arbitrarily in order to achieve the following aims: trick some generals into attacking; force a decision that is not consistent with the generals' desires, e.g. forcing an attack when no general wished to attack; or confusing some generals to the point that they are unable to make up their minds. If the traitors succeed in any of these goals, any resulting attack is doomed, as only a concerted effort can result in victory.

Byzantine fault tolerance can be achieved, if the loyal (non-faulty) generals have a unanimous agreement on their strategy. Note that if the source general is correct, all loyal generals must agree upon that value. Otherwise, the choice of strategy agreed upon is irrelevant.

Early solutions

Several solutions were originally described by Lamport, Shostak, and Pease in 1982.[1] They began by noting that the Generals' Problem can be reduced to solving a "Commander and Lieutenants" problem where Loyal Lieutenants must all act in unison and that their action must correspond to what the Commander ordered in the case that the Commander is Loyal. Roughly speaking, the Generals vote by treating each others' orders as votes.

- One solution considers scenarios in which messages may be forged, but which will be *Byzantine-fault-tolerant* as long as the number of traitorous generals does not equal or exceed one third. The impossibility of dealing with one-third or more traitors ultimately reduces to proving that the 1 Commander + 2 Lieutenants problem cannot be solved, if the Commander is traitorous. The reason is, if we have three commanders, A, B, and C, and A is the traitor: when A tells B to attack and C to retreat, and B and C send messages to each other, forwarding A's message, neither B nor C can figure out who is the traitor, since it isn't necessarily A – the other commander could have forged the message purportedly from A. It can be shown that if n is the number of generals in total, and t is the number of traitors in that n, then there are solutions to the problem only when n is greater than or equal to $3t + 1$.[4]
- A second solution requires unforgeable signatures (in modern computer systems, this may be

achieved in practice using public-key cryptography), but maintains Byzantine fault tolerance in the presence of an arbitrary number of traitorous generals.

- Also presented is a variation on the first two solutions allowing Byzantine-fault-tolerant behavior in some situations where not all generals can communicate directly with each other.

Practical Byzantine fault tolerance

Byzantine fault tolerant replication protocols were long considered too expensive to be practical. [*citation needed*] Then in 1999, Miguel Castro and Barbara Liskov introduced the "Practical Byzantine Fault Tolerance" (PBFT) algorithm,[5] which provides high-performance Byzantine state machine replication, processing thousands of requests per second with sub-millisecond increases in latency.

PBFT triggered a renaissance in BFT replication research, with protocols like Q/U,[6] HQ,[7], Zyzzyva,[8] and ABsTRACTs [9] working to lower costs and improve performance and protocols like Aardvark[10] working to improve robustness.

UpRight[11] is an open source library for constructing services that tolerate both crashes ("up") and Byzantine behaviors ("right") that incorporates many of these protocols' innovations.

One example of BFT in use is Bitcoin, a peer-to-peer digital currency system. The Bitcoin network works in parallel to generate a chain of Hashcash style proof-of-work. The proof-of-work chain is the key to solving the Byzantine Generals' Problem of synchronising the global view and generating computational proof of the majority consensus.[12] heard about this, but forgot details

Additionally to PBFT and Upright, there is also the BFT-SMaRt library[13] , a high-performance Byzantine fault-tolerant state machine replication library developed in Java. This library implements a protocol very similar to PBFT's, plus complementary protocols which offer state transfer and on-the-fly reconfiguration of hosts. BFT-SMaRt is the most recent effort to implement state machine replication, still being actively maintained.

See also

- Atomic commit
- Brooks–Iyengar algorithm
- Consensus (computer science)
- Quantum Byzantine agreement

References

1. ^ [a] [b] Lamport, L.; Shostak, R.; Pease, M. (July 1982). "The Byzantine Generals Problem" (http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf) . *ACM Transactions on Programming Languages and Systems* **4** (3): 382–401. doi:10.1145/357172.357176 (http://dx.doi.org/10.1145%2F357172.357176) . http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf.
2. ^ Amazon S3 Availability Event: July 20, 2008 (http://status.aws.amazon.com/s3-20080720.html)
3. ^ Pease, M.; Shostak, R.; Lamport, L. (April 1980). "Reaching Agreement in the Presence of Faults". *Journal of the ACM* **27** (2): 228–234. doi:10.1145/322186.322188 (http://dx.doi.org/10.1145%2F322186.322188) .

Qu: what problem is this trying to solve?

6.858: Computer Systems Security

Fall **2012**

Home

General information

Schedule

Reference materials

Piazza discussion

Submission

2011 class materials

Paper Reading Questions

For each paper, your assignment is two-fold. By the start of lecture:

- Submit your answer for each lecture's paper question via the submission web site in a file named `lecn.txt`, and
- E-mail your own question about the paper (e.g., what you find most confusing about the paper or the paper's general context/problem) to 6.858-q@pdos.csail.mit.edu. You cannot use the question below. To the extent possible, during lecture we will try to answer questions submitted by the evening before.

Lecture 19

Suppose you are building an online multi-person game. You are worried that a player can cheat in various ways by modifying the game software, since it runs on the player's own computer, or sending arbitrary network messages to your game server. What security properties could you get by using TrInc in your game (e.g., a trinket comes in the box when you buy a game)? What security problems cannot be solved with TrInc?

"Get it to attest the code running on your machine"
but how would this do that?
Could attest to commands sent
So you can't send other stuff
depending on the architecture of the game

Questions or comments regarding 6.858? Send e-mail to the course staff at 6.858-staff@pdos.csail.mit.edu.

Top // **6.858 home** // *Last updated Friday, 12-Oct-2012 23:31:47 EDT*

11/25

Paper Question 19

Michael Plasmeier

It seems that TrInc is best suited for a certain type of distributed multi-player game where you send commands to other individual users (not a central server). Using a PeerReview system, you could attest to the string of commands that you make and send to other users. That way you could eventually be found if you cheat by sending different users different actions that you have taken.

However, I don't see a way for this to attest that the game code running on your machine is original (so you can perform actions like jump to arbitrary places on the map as long as the other machines are not verifying those commands), or that there are no other programs running on your machine (for instance an automated agent that does boring things like farming gold for you).

Tr Inc

11/26

(5 min late)

Make sure not give 1 party an answer and give another party another

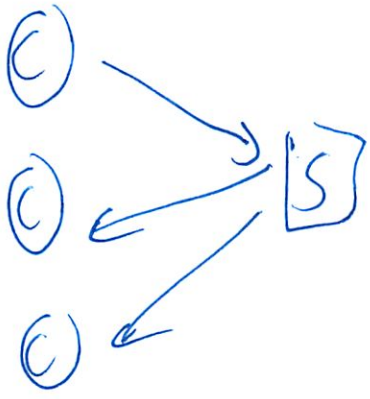
Storage - how secure

- - -

Hard to just do w/ crypto

Nieve

All messages go through server



But server is bottleneck

(7)

Need some trusted piece

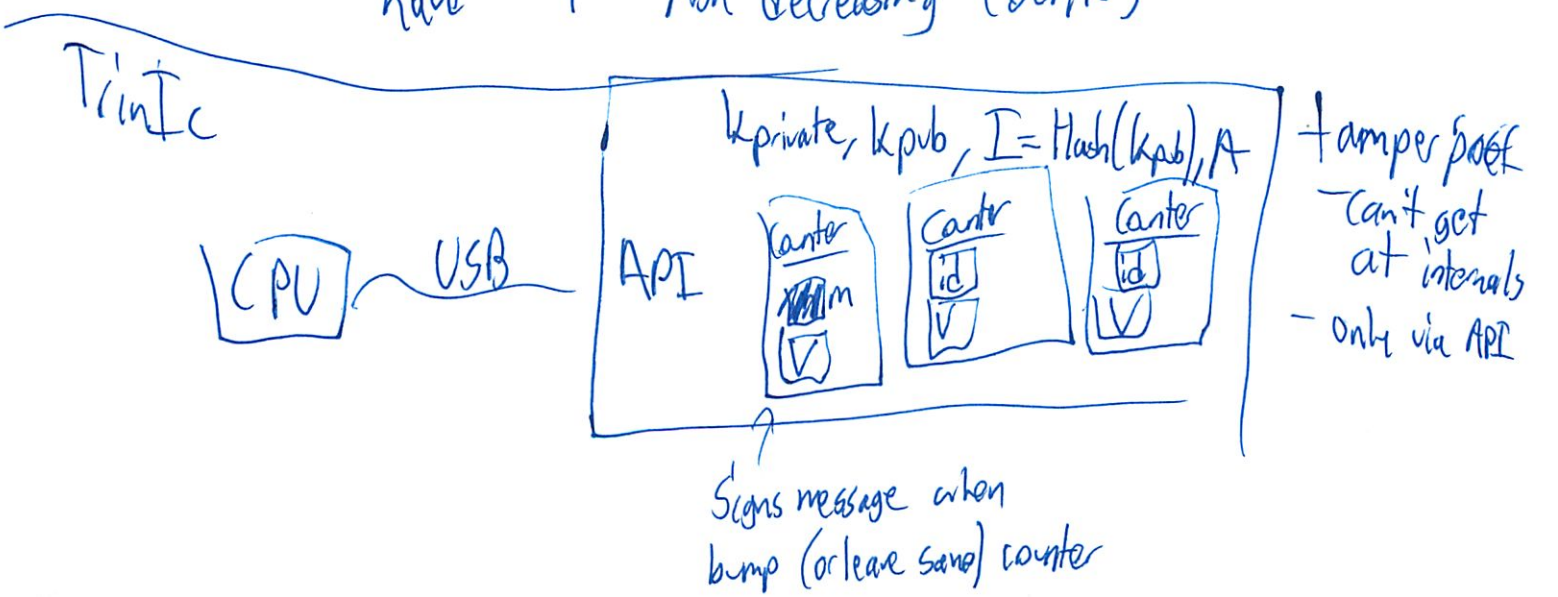
TPM Can attest to what code running on HW

Why need other HW?

- slow
- TPM need low level HW in - they don't need it

But could get it to work on a TPM

have 4 non decreasing counters



(3)

A = attestation from manuf
That TrInc is legit

but no where do they give an alg to verify
but if trust in HW why putting trust again in SW?

But if you broke down 1 - you would have
a kpriv and A
Could reuse
Perhaps they left vague so they could change
it later

If more counters than hw for
then each counter has an id m
m is used to allocate new counter ids
Can create new counter if other runs out
or want to reset to 0

(5)

(missed)

Crash Recovery

Attest

Should be atomic
- 1. Reply w/ attestation
- 2 Bump counter

So bump counter 1st

So can't get same attestation twice for same counter id

Could you skip #s?

In some protocols Counter only increment by 1

Verifier wants all your messages' attestations

So can't be missing attestations

?Solve in HW - w/ a mini battery

But can't reply if main CPU crashes,

So keep the last several replies to host in case host needs them

(6)

So

1. Put att in Q
2. Bump counter
3. Reply w/ attestation

How do ya use?

Requires some profile modifications

First → what is the meaning of a particular counter value

Second → what message to attest to

- increment counter
- or (- keep counter the same

$\langle I, i, C_{old}, C_{new}, H(message) \rangle_{k_{priv}}$

i ↑ counter id

3. How does counter id fit into application naming?

BitTorrent

Why?

BT ties to keep trades even

But others will send ya pices up front to start

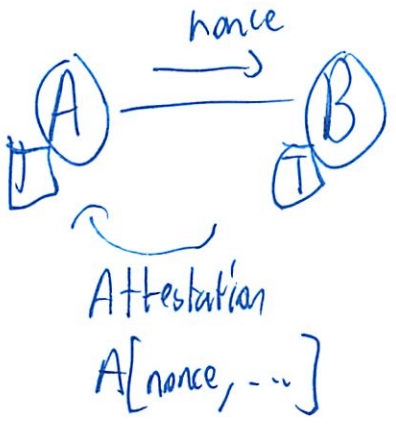
Some be nice bootstrap

So TrInc used to certify which files they have

Counter = # of pices/blocks ya have recieved

messages = Counter goes up; A uploads blocks to B

Counter stays same; A queries B's blocks



So A knows B's response is fresh

(B)

When A gets block, needs to send reciept to B which gets B to bump blah

Reciept

A[c → c+1, H(Block)]

Remember could make new ID

but ~~the~~ app must reject

One cheating → form as 100 sep members all one block

What prevents?

Unclear

Up to protocol?

; Register w/ phone #;

Need some notion of identity

(9)

Why must we include Hash (Block) in request

Then B commits to having that # block
if report wrong block
if ~~send~~ sent a duplicate block

They → then ↑ a counter
and gets over n (# of blocks)
so big disincentive not to lie

---

alt scheme → Or could force him to send all attestations
So can see all the counter blocks

Refuse to talk if no attestation

But can cheat each person once

So if # people > # blocks they could cheat

(10)

Payment System / E-Cash

last years quiz, not in Pape

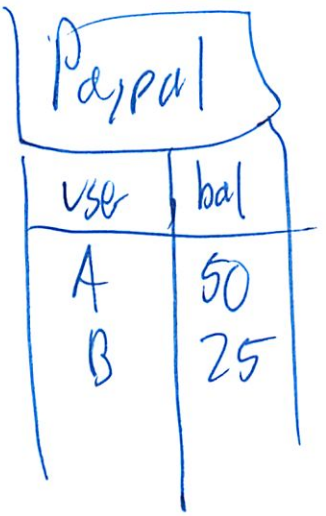
(A)

(C)

(B)

Regular crypto not good enough

Well Call use this mode



But through central 3rd party

Can we do this w/o it?

(11)

- for privacy
- and offline access

If just try w/ crypto people could double spend $

Coin → is $1

Coin: {$1}$_{\text{Bank}}$ ← signed by bank

But can make copies!

So use a Counter!

each id represents a counter
value is if coin is spent or not

0 → not spent
1 = spent
(: what if it comes back)

(12)

So

Give coin to B: Attest (id, c=1, "give coin to B")

But T.Inc has limited # of counters

So need to work around that...

But must tie id to counter

First must have B allocate a new counter

B.id, B.counter_id

(B.I, B.id)

How coins are created:

<"coin to (A.I, A.id>" }Bank

(missed)

Coin: - Initial msg from bank
- Chain of A H to cur counter
- A for each trinket in chain

13

BA this chain grows over time
And no anonimity

(How does BitCoin do it again?)

BitCoin has very poor anonimity
Can see entire graph of $ transfer

Lots of schems
But ~~nothing~~ nothing ideal

esp recovery + error handling
easy to steal BitCoins

Bank can do a audit scheme
Prof: I don't get why anyone uses BitTorrent

(14)

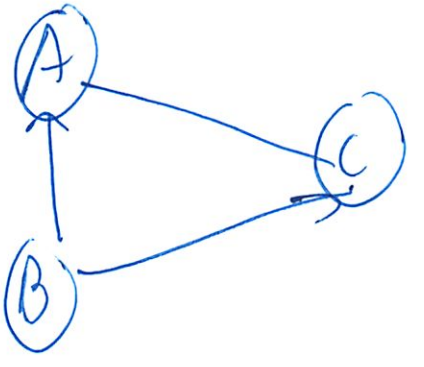
Games

P2P

No central server

Send msgs directly b/w nodes

But B could send diff messages to A, C



Could use counter to track # of messages sent

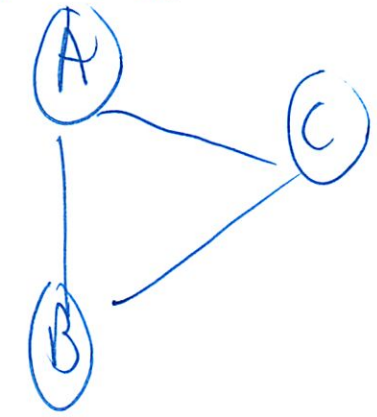
C recieves log of messages from B

Must be exactly +1

Must send same msg to everyone

Must be +1 from everyone

(15) B C_B
C C_C



A: C_A
B: C_B

T: C_B

B → A[$C_B \rightarrow (C_{B+1}, h(m))$]

Send to A, C

So must increment value

If B only send to C

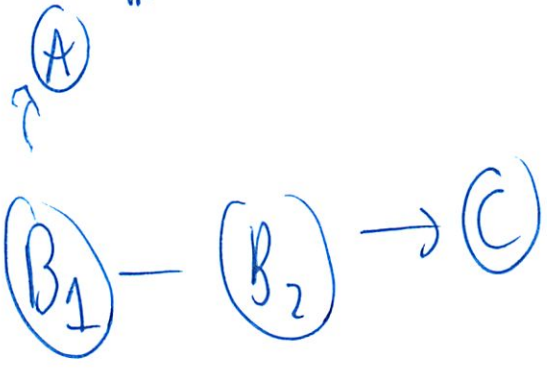
Then B can only attest $C_{B+1} \rightarrow C_{B+2}$

And A is looking only for increment by 1

(Ah that's it → only hashes once for a value and the increment by 1 value only)

16

Still vulnerable to the behind the scenes multiple IDs



must bootstrap some other identity plan to prevent other users

(missed)

Perhaps e-cash scheme for points in game

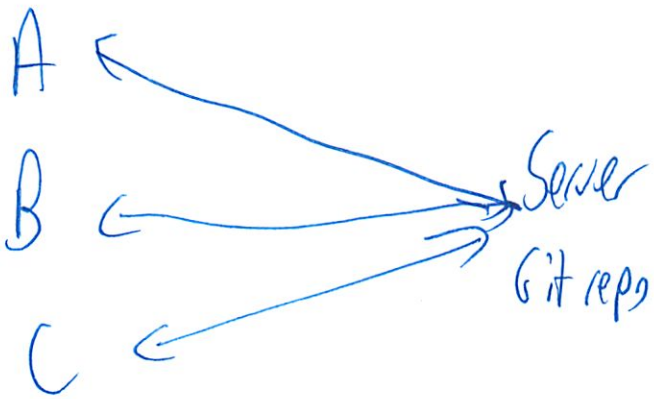
Version control

Storage

Git repo

(idea: file timestamp
- discovery
- patents)

(17)



H ~ commit
↓
commit
↓
commit
↓
base

Could all have key

- all save key
- ~~all~~ trust all clients
- but not server

Then $\{commit\}_K$

But server can still pretend not to have an up to date version

Server could also fork

Server Server'

Server' has diff copy of git repo

So they are off on their own branch
- divergent histories

If server has T can get it to commit to version
- # of commits = counter

Checkout → get attestation of counter
When commit get attestation it ↑

So A, B, C all same counter

(19)

Patents

No notion of time

Or order

- unless give to others

(I didn't get that last part - his ans to my q)

11/26

Trusted hardware

Administrivia.

- All labs graded.
- Final projects: 25+ groups.
- Two choices (both strictly enforced):
  - 4 minutes per group on Wednesday Dec 12, or
  - 8 minutes per group split over Monday Dec 10 + Wednesday Dec 12?
- Vote outcome: 4 minutes per group on Wednesday.

What's the problem this paper is trying to solve?

- Scenario: suppose we don't trust most parts of some computer system.
  - Controlled by potentially malicious user.
  - Typically distributed system communicating over the Internet.
  - Client machines could be ~arbitrary.
- Still want to have that system participate in a network protocol.
  - Many examples: BitTorrent clients, DNS servers, shared git repository, online game participants, ...
- How can we trust what some system (either client or server) is telling us?

Can we use crypto?

- Encrypt, sign all data.
- Untrusted machine can't tamper with data, but can store it, relay it, ..
- (Almost) works for simple applications like data storage.
- Doesn't let us reason about what operations the machine is doing.
  - E.g., received a chunk of data, sent some message, etc.

One approach: use a trusted server.

- All interactions go through this server.
- Server records what each client did.
- Just as capable as TrInc, but centralized server can be a bottleneck.

Alternative approach: trust some piece of hardware at the untrusted system.

- Trusted hardware component will generally sign messages on behalf of system.
- Trusted hardware constrains the kinds of messages it will sign in some way.
- Assume adversary can't tamper with trusted hw, it's implemented correctly, ..
- Construct system around primitive provided by trusted hardware.
  - As long as messages are signed correctly, means system is operating OK.

What should this trusted hardware look like?

- One version we've already seen: HW determines what code is running on machine.
  - TPM-based approach.
  - Advantages?
    - Quite general, can capture almost any property you might want.
    - TPMs are actually installed in many PCs today.
  - Drawbacks?
    - Large TCB, both in terms of software and hardware.
    - Weak threat model in terms of hardware attacks.
- Another version: TrInc.
  - More restricted kinds of properties.
  - Just think about network messages from a particular system.
  - Worried about "equivocation".
  - In principle, could check for equivocation without trusted HW.
  - Hard to do in a scalable way -- requires centralized component.

What does TrInc provide?

- State: Fig 1.
  - K_priv.
  - K_pub.
  - I = H(K_pub), A from manufacturer.
  - Several counters, each with their own count.

```
  Hardware is assumed to be tamper-proof.
    No way to tamper with the trinket's state except via API.
  API: Fig 2.

What happens when all counters are used up?
  Hardware has a fixed number of counters.
  Each hardware counter has both a counter value and an ID.
  Special counter, M, used to allocate new counter IDs.
  When counter gets a new ID, its value gets reset to zero.

How do TrInc's counters prevent equivocation?
  Trinket's <K_pub, counter_id> tuple is the identity for participant.
  Counter associated with identity cannot be decremented.
  Can be leveraged to map onto equivocation in the application's protocol.

Why does TrInc allow the counter at the same value?
  How do you tell if the counter was bumped?
  Attestation includes old, new value.

Crash recovery.
  What happens if counter is bumped before the attestation is written to disk?
  What if attestation is revealed before counter is bumped?
  How does TrInc deal with both?
    Write attestation to persistent queue, bump counter, return attestation.
    Provide an API call to read attestations from the queue.
    On startup, discard attestations that did not bump the counter.

What's the trust model?
  Each manufacturer includes a certificate along with each trinket.
    Certificate signs trinket's ID / public key.
    Certificate means only a legitimate trinket knows the private key.
  Certificates only verified by end-users.
    Authors don't want to prescribe a fixed certificate verification scheme.

Symmetric key crypto.
  Public-key signatures are slow.
  TrInc allows associating a symmetric key with a counter.
    Used to endorse & verify attestations for that counter.
  Where does the symmetric key come from?
    Some common party trusted by all participants in that protocol instance.
    This party verifies every participating trinket's attestation.
    Generates new symmetric key, encrypts for each trinket's public key.
  Why not use an arbitrary trinket as the trusted party?
    Trust model: no single prescribed way to verify a trinket's certificate.
    There's already a trusted party in many protocols.

TrInc's trinket can be implemented on top of a TPM chip.
  TPM also provides non-decreasing counters, although somewhat different.
  Need to use "late launch" functionality: hardware VM support, SKINIT.

What do we need to use TrInc in a protocol?
  Section 5.1.
  Figure out what the counter value represents.
    Should be that equivocation = counter rollback.
  Figure out what data should be in the attestation.
    Advance attestation (where counter is bumped).
    Status attestation (where counter stays the same).

  TrInc can only prevent equivocation (counter rollback) for its principal:
    <K_pub, counter_id>.
  Need to also map some application-level principals to TrInc counter names.

Applications of TrInc.
```

```
BitTorrent.
  Problem: peers may want to under-report blocks (why?)
  Using TrInc in BitTorrent:
    What's the participant in the system?  TrInc counter, for specific file.
    What does the counter map to?  Number of blocks received.
    Algorithm 3 in paper.
    Why do we need to include received block in attestation?
      Otherwise adversary could use one attestation for two blocks.
    How to figure out what blocks a peer has?
      Send nonce, require fresh TrInc attestation with current bitfield.
  How can an adversary cheat in TrInc-enabled BitTorrent?
    Refuse to acknowledge receipt of block.
    Multiple identities.

Games.
  Problems: if game is peer-to-peer, hard to know how to trust messages.
  How do you know if a participant sent different messages to me & others?
  Centralized server can solve problem, but may be inefficient.
  Can associate a TrInc counter with each player.
    Counter value: number of game steps / operations performed.
    Each attestation must bump counter to the next value.
    Others should expect monotonically increasing counter values in messages.
    Adversary cannot send different (valid) messages to two different players.
  Are there other security problems in a game we could solve with TrInc?
  What things cannot be solved with TrInc?

E-cash: from last year's quiz.
  General problem: double-spending.
  Counter + chain of attestations represents one coin (if counter is at 0).
  If A wants to give B a coin, first ask B for some counter ID.
  Then bump A's coin counter from 0 to 1 with message containing B's counter ID.
  Coin is valid if it was initially issued by the bank,
    all attestations correspond to a counter going from 0 to 1,
    and all counter IDs are "correct" (from previous message in chain).

Version control system: better git.
  Problem: adversarial server could lie about what versions it has.
  Git already names everything by hash: boils down to getting the right hash.
  What if we sign all updates, and all clients know about each others' keys?
    Prevents server from generating its own changes or versions.
    Cannot generate an arbitrary signed hash.
    Each client can check that its previous hash is in history of new hash.
    Forking attack: clients see different but consistent "forks" of server.
  How can we use TrInc to prevent forking attacks?
    Server's name is the name of one particular counter <K_pub, counter_id>.
    Simple plan: counter is number of check-ins in repository.
      On checkin, server attests to new (signed) hash & length of commit chain.
      Client waits for attestation of its checkin.
      Server keeps entire history of attestations for every check-in.
    To check out repo, client requests this history.
      Use nonce to get current server-side counter version.
      Verify that all versions are chained correctly.
      If check-in happened (client got attested response),
        that check-in must correspond to some counter#.
    Can we just record the last signed-hash & history-length in attestaton?
      Potential problem: server could maintain two hash chains in parallel.
      Odd counter numbers & even counter numbers correspond to two chains.
      Server can fork them as long as check-ins can skip counter numbers:
        client thinks counter was at N, uploads 2 checkins, counter at N+2.
      Can solve by forcing client to upload each checkin separately.
      By uploading checkin #N, client claims trinket was OK up to N-1.
  Of course, critical to agree on specific counter ID for repository.
```

```
DNSSEC.
  Problem: DNS records are signed, but old signed records are still valid.
  Associate TrInc counter name with DNS server.
  Simple plan: counter is current version number for all DNS records.
    Use TrInc as a trusted version number.
    Drawback: server must re-sign all records with new version number on bump.
  Alternative plan: git-style version history.
    Much more expensive for client to do a lookup now.
  Paper doesn't present an efficient design for how to use TrInc.

PeerReview / AVM.
  Hybrid of TPM-style trusting code and lighter-weight trusted hardware.
    Don't check exactly what code runs on untrusted client.
    Instead, run client software in deterministic virtual machine.
    Record all possible inputs to VM (keyboard, mouse, interrupts, packets).
    Another party can now verify whether output network packets were legit,
      by re-running the entire VM on recorded inputs.
  Can use TrInc to attest to these inputs.
    Prevents client from equivocating about what inputs
      (e.g., keypresses) they performed on application.
    Protects against variety of attacks on games.
      Outputs that couldn't have been generated by any legitimate inputs.
      Incompatible outputs (e.g., equivocation about user input).
    Cannot protect against introspection of application state.
    Cannot protect against automated input generators (e.g., bot game players).

Advantages of TrInc compared to TPM-style trusted HW?
  Small trusted hardware, fewer assumptions about hardware tampering.
  No assumptions about code correctness (at least at the "untrusted" end).
  Application guarantees map more directly to properties provided by hw.
  Can run many applications concurrently, TrInc counters "composable".

Disadvantages of TrInc compared to TPM-style trusted HW?
  Requires changing application/protocol.
  Research prototype -- not currently availble in commodity PCs.
  Only applicable to equivocation-type misbehavior.
    Cannot reason about internal state.
  Application may need to provide stronger form of identity.
    Easy to create multiple identities.
      Just allocate another TrInc counter in trinket.
      Or even buy another trinket.
    Adversary may be able to get its "compromised" clients/counters to collude.
      E.g., exchange arbitrary blocks w/ each other in BitTorrent.
    Was this a problem with the TPM?  Adversary could buy multiple TPMs.
      Yes, to some extent.
      Adversary could have multiple TPMs, run multiple BitTorrent clients.
      Harder to precisely control blocks exchanged between his multiple clients.
      More costly to allocate one TPM per client clone.
        But also advantage of TrInc: lightweight, supports concurrent apps.

References:
  http://sparrow.ece.cmu.edu/~adrian/projects/pioneer.pdf
  http://peerreview.mpi-sws.org/
  http://www.cis.upenn.edu/~ahae/papers/avm-osdi2010.pdf
```