InvisiType: Object-Oriented Security Policies

Jiwon Seo Monica S. Lam
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{jiwon, lam}@stanford.edu

Abstract

Many modern software platforms today, including browsers, middleware server architectures, cell phone operating systems, web application engines, support third-party software extensions. This paper proposes InvisiType*, an object-oriented approach that enables platform developers to efficiently enforce fine-grained safety checks on third-party extensions without requiring their cooperation. This allows us to harness the true power of third-party software by giving it access to sensitive data while ensuring that it does not leak data.*

In this approach, a platform developer encapsulates all safety checks in a policy class and selectively subjects objects at risk to these policies. The runtime enforces these policies simply by changing the types of these objects dynamically. It uses the virtual method dispatch mechanism to substitute the original methods and operations with code laced with safety checks efficiently. The runtime hides the type changes from application code so the original code can run unmodified.

We have incorporated the notion of InvisiType into the Python language. We have applied the technique to 4 real-world Python web applications totaling 156,000 lines of code. InvisiType policies greatly enhance the security of the web applications, including MoinMoin, a popular, 94,000-line Wiki Engine. MoinMoin has a large number of third-party extensions, which makes security enforcement important. With less than 150 lines of Python code, we found 16 security bugs in MoinMoin. This represents a significant reduction in developers' effort from a previous proposal, Flume, which required 1,000 lines of C++ code and modifications to 1,000 lines of Python code.

Our InvisiType policies successfully found 19 cross-site scripting vulnerabilities and 6 access control errors in total. The overhead of applying the policies is less than 4 percent, indicating that the technique is practical.

1 Introduction

Third-party software extensions play an important role in many software frameworks today. On the client side, important extensions include ActiveX, Javascript and Java applets embedded in web pages. Many "middleware" server platforms provide a base on which developers can build their applications: examples include servlets on web servers and extensions in wiki engines. Finally, there is an emerging class of server platforms that let end users supply their own code: these include application engines at Google, Facebook and MySpace. This paper describes a new object-oriented approach to allow software platform developers to efficiently impose fine-grained security policies on third-party software as well as platform software.

1.1 Motivation

Third-party code adds greatly to the difficulty of securing software. Even if we require that third-party code be signed by a trusted party, it is likely that third-party software has more security vulnerabilities than the underlying software platform. Not only are third-party coders typically less experienced, they may be ignorant of the implicit principles used in the design of the platform. As a case in point, extensions for the popular MoinMoin Wiki system often neglect access control checks for Wiki pages. In addition, third-party code seldom goes through the same level of review and testing as the platform itself.

Measures used today to guard against malicious software are too crude for third-party software extensions, which are responsible for providing much of the user-desired features on a platform. Android isolates applications from each other by executing them in different virtual machines. The Google App Engine constrains direct operating system accesses and limits resource usage with a quota system. The Java Development Kit (JDK) uses sand-boxing to restrict unsigned code from accessing certain system resources, and allows permissions to be granted to classes and objects belonging to protection domains [10]. After permissions are granted however, JDK users do not have control over how the resources are used or what objects are accessible. For example, once network access is granted, it is impossible to enforce that only non-sensitive data are sent over the network.

1.2 Object-Oriented Safety Checks

It has been found that software often have implicit design rules that govern the integrity of the program. Take for example SQL injection, an important class of security vulnerabilities. SQL injection [1, 16] can be eliminated by simply disallowing access of the database with *tainted* strings, input strings that have not been checked for malicious contents. A number of static and dynamic techniques have been proposed to find such kinds of errors in programs automatically [11, 12, 18]. A promising approach is to allow application writers to define application-specific safety checks. Aspect programming [14, 15] allows programmers to write down codes related to, say, a safety check in one place, and many lines of code scattered throughout the program may be changed syntactically [19, 20].

Going beyond a syntactic rewrite system, our goal is to extend the principles and benefits of object-oriented programming to safety checks. Safety checks of primitive types, such as null pointer and array bound checks, have been built into high-level programming languages. They improve software robustness as well as relieve programmers of the burden of inserting the error checking code. Analogously, classes often have class-specific safety checks, such as the prevention of data leakage via access right violations and security vulnerabilities through unchecked inputs.

Allowing class designers to specify safety checks in a class and having the runtime system enforce these checks efficiently, we bring automatic safety check benefits to a higher semantic level. Fine-grained control enables us to harness the true power of third-party software by giving it access to sensitive data and resources in the system without unduly exposing the platform to security attacks.

1.3 Introduction of InvisiType

Our approach to providing object-oriented security check builds upon the fundamental idea of class hierarchies in object-oriented programming. Roughly, our approach is to create subclasses that extend the original class definitions with virtual methods that contain safety checks. Through the virtual method dispatch mechanism, subclassing enables software to be extended efficiently with no changes to the original code.

More specifically, we introduce the notion of a new type called the *InvisiType*[1] to enforce object-oriented safety checks. An InvisiType is a class with special properties known to the language's runtime system.

Multiple inheritance. Safety-checking policies are encapsulated as *policy classes* derived from the InvisiType class. We can impose a policy on an ordinary class by creating a new subclass, called a *protected* class, that inherits from both the original class and the policy subclass itself.

Suppose we wish to keep track of input strings to guard against SQL injection vulnerabilities. We can create a taint policy, `TaintPolicy`, subclassed from InvisiType. A tainted string is an instance of a protected class derived from the `string` class and the `TaintPolicy` class.

Safety-checking rules. A policy class defines how all operations, including method invocations, built-in operators, and native function calls, in an ordinary class are to be augmented with safety checks. It also defines exception-handling code. The runtime system of the language automatically enforces the safety-checking rules and raises exceptions where necessary. The exceptions may be caught to dynamically recover from the error conditions.

Third-party code transparency. Protected classes are nameless and invisible to the application code, as suggested by the name InvisiType. This ensures that third-party code works without modification; even the `type()` function call returns the original type. Users of a class are oblivious to the safety checks, just like they are oblivious to the insertions of null checks.

Dynamic selection of policy enforcement. Unlike primitive safety checks, object-oriented safety checks can be complex and can incur higher overheads. It is necessary to allow application writers to choose dynamically the instances of a class to be subjected to safety policies. For example, only input strings need to be treated as tainted; furthermore, they should be classified as untainted after they are checked for malicious contents.

In our design, we make inheritance and disinheritance of safety policies a dynamic feature by generating protected classes and changing the protection type of an object dynamically. To this end, we introduce a pair of built-in functions, *demote* and *promote*. Demote puts an instance of an ordinary class under a safety policy by making the instance a member of the protected class derived from the instance's class and the policy class. A demoted object is also referred to as a *protected* object in this paper. Conversely, promote turns a protected instance back to an ordinary instance of the original class.

Efficient implementation. Unlike syntactic rewrite systems such as Aspect Programming [14, 15] that change the definition of methods for every instance, InvisiType allows

[1]The name InvisiType is inspired by Invisalign, which is a series of transparent teeth aligners used as an alternative to traditional metal dental braces.

method definitions to change only for selected instances, and only for the duration an instance is subject to safety policies. The safety checking rules are implemented by modifying the virtual method dispatch mechanism. As a result, the runtime system can enforce the safety checks with a negligible overhead. This makes InvisiType especially efficient for low-level safety checks that need to be applied only to selected objects.

1.4 Contributions of this Paper

This paper aims to minimize vulnerabilities in software caused by coding errors in third-party extensions as well as platform softwares. Examples of errors of this sort include SQL injection, cross-site scripting, and incorrect access control checking. It is not the intention of this paper to guard against malicious third-party developers.

The premise of this paper is that software platform developers can insert fine-grained control over third-party extension efficiently and transparently by encapsulating safety checks in an object-oriented manner. The specific contributions of this paper are described below.

Concept of InvisiType. We propose encapsulating object-oriented safety-checking rules as policy classes, which are subclasses of the InvisiType class. Ordinary object instances can be subjected to these policies selectively, dynamically, and efficiently. The policies are enforced by the runtime system, requiring no change be made to the third-party code to be protected.

Common security policies described using InvisiType. We use InvisiType to implement common security policies such as taint tracking or access control on objects. These policies are generally applicable and can enhance security of many applications.

An efficient implementation of InvisiType for Python. We have incorporated InvisiType into Python, a widely used programming language. The design of InvisiType requires a relatively small change to the runtime system. The overhead incurred is found to be negligible.

Enhanced security of Python web applications. We used the InvisiType technique to enhance the security of 4 widely used Python web applications. These applications totaled approximately 160,000 lines of source code, not including the library code. All the web applications have support for third-party extensions. One example is the MoinMoin Wiki Engine which has a large number of extensions, and is known to have many security bugs both in its own distribution and in third-party extensions. We successfully found two important classes of security bugs in the applications; in MoinMoin the two classes of security bugs account for more than 50% of all known security bugs. In total, we found 25 security bugs in the 4 applications.

1.5 Paper Organization

The rest of the paper is organized as follows. We first illustrate the concept of InvisiType by showing how to use it to eliminate cross-site scripting vulnerabilities in web applications in Section 2. Section 3 presents the rationale and design of InvisiType. We then discuss the various security policies we have implemented using InvisiType in Section 4. We describe our implementation of InvisiType in Python in Section 5. Section 6 presents the experimental results of applying InvisiType policies to a number of Python web applications. Finally, Section 7 discusses related work and Section 8 concludes.

2 Eliminating Cross-Site Scripting With An InvisiType Policy

Cross-site scripting (XSS) is one of the most common vulnerabilities that plague web server applications. Attackers can send code to a web browser client if a web application echoes back user-input strings directly as output. Like SQL injection and many other vulnerabilities due to unchecked inputs, XSS can be prevented with information flow control, a technique that controls how data flows inside the system to the outside world. In this section, we introduce InvisiType by way of showing how it can be used to control information flow to prevent XSS.

2.1 Cross-Site Scripting

Let us first consider a simplified example of a cross-site scripting vulnerability drawn from the login extension code in the MoinMoin Wiki Engine, as shown in Figure 1. The function `cgi.FieldStorage()` returns a Python dictionary-like object representing the HTML form data. The login extension retrieves the `name` and `password` fields from the form data. If there is no user with the given name, the code responds with an error message. In this case, the user-provided input name is echoed back unchanged. By inserting some malicious Javascript code in the name itself, the Javascript gets executed when it is echoed on the client's computer, thus launching a cross-site scripting attack.

We can avoid cross-site scripting using the notion of tainting, which is a basic form of information flow control. There are four basic constraints in tainting:

Taint Constraint 1. All input strings are considered tainted.

Taint Constraint 2. A string is considered tainted if it is a concatenation of one or more tainted strings.

```
form = cgi.FieldStorage()
...
name = form["name"]
password = form["password"]
if not user.exist(name):
  error = "Unknown user name:" + name
  ...
  request.http_headers()
  request.write(error)
```

Figure 1. Cross-site scripting example from MoinMoin

Taint Constraint 3. Tainted strings cannot be used as arguments in system calls.

Taint Constraint 4. Strings are safe to be used in system calls if they cannot be interpreted as executable Javascript. Strings can be made safe, sanitized, by replacing characters with special meaning in HTML, such as replacing "`&`" with "`&amp`", "<" with "`&lt`", and ">" with "`&gt`".

This taint policy is applicable across many applications.

2.2 The Taint Policy

Let us show how we use InvisiType in the context of the Python language to implement tainting. The `escape` function of the `cgi` module in the Python standard library has already implemented sanitization. Third-party code is expected to call `escape` on all input strings before echoing them back.

The framework developer encapsulates the taint policy in our system with the definition of `TaintPolicy`, a subclass of `InvisiType`, as shown in Figure 2. All objects subjected to this policy are considered tainted. The definition of `__syscall__` enforces Taint Constraint 3, which states that tainted objects are not allowed to be used in system calls. The rest of the definition enforces Taint Constraint 2, which states that any concatenation of tainted strings yields tainted strings. Details of the policy definition are presented in Section 3.

2.3 Applying the Policy

A framework designer next applies the taint policy by tainting all input strings until they are escaped. Each input string is tainted by calling `demote` on the string along with the `TaintPolicy` class. Figure 3 shows tainting is

```
class TaintPolicy(InvisiType):
  def __add__(handler, self, other):
    return TaintPolicy.propagate(handler,
                                  self, other)
  def __radd__(handler, self, other):
    return TaintPolicy.propagate(handler,
                                  self, other)
  def __getitem__(handler, self, other):
    return TaintPolicy.propagate(handler,
                                  self, other)
  def __nativecall__(handler, self, args):
    return TaintPolicy.propagate(handler,
                                     args)
  ...
  def propagate(handler, *args):
    result = InvisiType.call(handler, *args)
    # demotes result with TaintPolicy
    return taint(result)
  def __syscall__(handler, self):
    raise OperationError("Tainted object is"+
                  "used in a system call")
```

Figure 2. Definition of TaintPolicy

```
recv =  socket.recv
socket.recv = (lambda *args:
         demote(recv(*args), TaintPolicy))
```

Figure 3. Tainting an input string received from a socket

```
def escape(s, quote=None):
  s = s.replace("&", "&amp;")
  s = s.replace("<", "&lt;")
  s = s.replace(">", "&gt;")
  if quote:
    s = s.replace('"', "&quot;")
  promote(s, TaintPolicy)
  return s
```

Figure 4. Untainting a sanitized string in the `escape` function in the `cgi` module

applied to strings received from a socket. Strings are untainted in the `escape` method by calling `promote` on the sanitized string and the `TaintPolicy` class, as shown in Figure 4. The pair of `demote` and `promote` operation implements Taint Constraints 1 and 4.

2.4 Detecting Safety Violations

The runtime system of the language automatically ensures that there is no information flow from an input string to system calls. A tainted string in this system is a string that has been demoted with the `TaintPolicy` class. All input strings are marked as tainted; the taint is propagated through concatenation; taint is removed when `escape` is called; any system call on tainted strings generates an exception. This example illustrates how fine-grained information flow control can be provided in a way that requires no modification to third-party code, at either the source or binary level.

3 InvisiType Design

In this section, we present the design and rationale of InvisiType. We first describe the class hierarchy of policy and protected classes, then describe how to define the safety-checking rules. Finally, we describe how InvisiType enables the platform developers to distinguish between third-party and platform code using the notion of restricted code domains.

3.1 InvisiType Policy and Protected Classes

As discussed in Section 1, policies are expressed as subclasses of `InvisiType`. A *protected* class is derived dynamically and implicitly, via multiple inheritance, from the policy and the class of the object being put under the policy. A platform developer can choose to place individual object instances under a policy possibly only for a duration. Thus, the type of an object can change dynamically. An object's class hierarchy consists of the classes it belongs to ordinarily, followed by a dynamically modifiable *protected class hierarchy* consisting of one or more protected classes, as shown in Figure 5. Note that the protected class hierarchy is visible only to the runtime system and not to the application software.

The built-in `demote` and `promote` functions can be used to apply a policy to or remove a policy from an object (Figure 6). Upon demotion, a new protected class implementing the given safety-checking rules in a policy class is added as the lowest member in the instance's protected class hierarchy. Upon promotion, the protected subclass corresponding to the given policy is removed from the object's hierarchy.

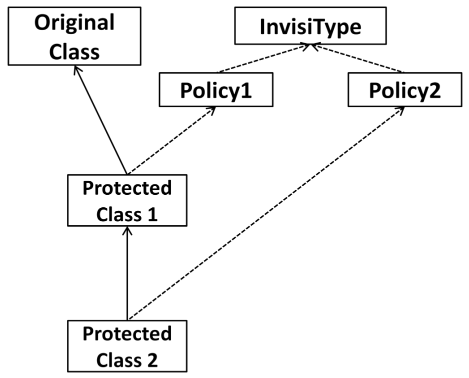

Figure 5. Class hierarchy for InvisiType subclasses: normal arrow indicates inheritance supported in an underlying language and a dotted arrow indicates inheritance supported by the InvisiType runtime.

3.2 Safety Checking In InvisiType Classes

InvisiType is designed as an extension of the underlying programming language to allow for application-specific safety checks. Safety checks are often required when the state of an object is read or modified and when methods are invoked on an object. Besides the code written in the object-oriented programming language itself, we also have to deal with native methods. Thus, in the following, we describe how we can add safety checks to each of the three categories in turn:

1. specific virtual methods
2. language primitives, and
3. native method call.

3.2.1 Method Invocations

A policy class can extend an ordinary class by defining new virtual methods with the same name. Methods overriding those in the original class are passed an extra argument by the runtime system: the method defined in the original class. This allows the overriding method to invoke the original method before or after safety checks.

Consider a common safety rule that requires access control be checked before confidential data can be accessed. Figure 7 illustrates how this safety rule can be enforced with `AccessControlPolicy`, a subclass of `InvisiType`. This policy simply overrides the

Function	Description
`demote(object, invisiType-class)`	Demote the object by adding the class of the given InvisiType policy to its protected class hierarchy
`promote(object, invisiType-class)`	Promote the object by removing the given InvisiType policy from its protected class hierarchy

Figure 6. Signature of the demote and promote functions

`accessConfidentialData` with one that raises an exception. All newly created clients are subjected to the AccessControlPolicy by demoting them with respect to the policy class. They are promoted back when they pass the `checkACL` method.

```
class AccessControlPolicy(InvisiType):
  def accessConfidentialData(method, self):
    raise Exception("Illegal Access")
class Client:
  def __init__(self):
    ...
    demote(self, AccessControlPolicy)
  def checkACL(self):
    ...
    if success == True:
      promote(self, AccessControlPolicy)
  def accessConfidentialData(self):
    # accessing confidential data
```

Figure 7. Over-riding a virtual method in AccessControlPolicy

3.2.2 Language Primitives

Dynamic languages like Python have a set of built-in operation handlers that can be overridden by the application. A policy class can insert fine-grained safety checks by overriding these operation handlers. For example, it can restrict access to certain attributes of a protected object by changing the `__getattr__` operation handler; or it can restrict the methods that can be invoked on protected objects by adding checks to the `__call__` operator handler. Like the above, the new handlers also take the policy object and the handler for its superclass as additional arguments.

Let us return to the TaintPolicy example defined in Figure 2. The policy class overrides operation handlers such as `__add__`, `__radd__` and `__getitem__`. When a binary add operation is applied to an object subjected to the `TaintPolicy`, the `__add__` method in the `TaintPolicy` class is called. Passed to the method are the the original handler of the add operation (`__add__` in string type) and original arguments to the `handler(self` and `other`). The `__add__` method calls the `propagate` method and passes all the arguments. `Propagate` delegates the operation to the `handler` and demotes the result object with `TaintPolicy`.

Taint propagation is implemented similarly for other operations such as `__getitem__` which returns a copy of the character at the given index from the string. Taint also propagates through string native methods such as `upper()` which returns a copy of the string converted to the upper case. The `__nativecall__` method is called for such native methods, and it calls the `propagate` method to perform taint propagation by demoting the return value of the native functions.

3.2.3 Native methods

We now describe how we handle native method implementations. System calls are a particularly important subset of native methods as they interface with the external world. We often have to perform special checks to avoid leakage of sensitive data from the application. We need to monitor which of these native methods are called, and check the protected parameters passed into such methods.

To handle native methods, the InvisiType class has pre-defined four methods that are invoked by the runtime system: `__syscall__`, `__nativecall__`, `__before_nativecall_arg__`, and `__after_nativecall_arg__`, as described in Table 1.

The InvisiType runtime invokes the `__syscall__` method *before* a system call is made. A policy class can override the `__syscall__` to add safety checks to system calls. The `__syscall__` method takes two arguments: the system call function and the demoted object on which the system call is invoked.

In the `TaintPolicy` example in Figure 2, the `__syscall__` method is overridden to throw an exception if any system call is to be attempted on a protected object. This prevents tainted strings from flowing into system calls.

Whenever the InvisiType runtime encounters a native method invocation to a protected object, it will call `__nativecall__`. Again, policy classes can perform safety checks by overriding the `__nativecall__` method. `__nativecall__` takes as arguments the native method

Method	Description
`__syscall__`	Method invoked when a protected object is used as an argument to a system call
`__nativecall__`	Method invoked when a natively implemented method of a protected object is called
`__before_nativecall_arg__`	Method invoked before a protected object is used as an argument to a natively implemented method
`__after_nativecall_arg__`	Method invoked after a protected object is used as an argument to a natively implemented method

Table 1. Pre-defined methods in the InvisiType class

called, the demoted object, the arguments, as well as a set of keywords summarizing the relevant semantic information in the native methods. For example, the keyword `readonly` specifies if the native method to be invoked only reads and not writes the object. This information allows the policy class to react accordingly.

Finally, whenever a protected object is passed as an argument to a native method of another object, the `__before_nativecall_arg__` and `__after_nativecall_arg__` methods are invoked before and after the native method, respectively. Overriding this method allows safety checks to be added before the object is used or after the native method returns.

3.3 Restricted Mode

To enable policies that differentiate between the third-party code versus the framework itself, the InvisiType system introduces the notion of a restricted mode. It has two built-in functions:

`import_restricted(modulename)` imports an untrusted or third-party module,

`in_restricted_mode()` returns true if the current frame is executing code from a restricted module. This can be used to restrict access in restricted modules.

We now demonstrate how `in_restricted_mode()` can be used to prevent untrusted functions from modifying protected objects. The `ReadonlyPolicy`, as shown in Figure 8, overrides two methods: `__setattr__` and `__nativecall__`. The former prevents restricted code from writing to a protected object's attributes directly. The latter prevents restricted code from invoking any native methods that do not possess the `readonly` property.

3.4 Language Compatibility

The InvisiType technique is applicable to object-oriented languages supporting class inheritance. Multiple inheritance is not required at the language level, as it is not exposed at the programming level. The only class having more than one immediate superclass is a protected class. Its inheritance from a policy class is set up by the InvisiType runtime system upon the invocation of a demote operation.

```
class ReadonlyPolicy(InvisiType):
  def __setattr__(handler, self, name, attr):
    if in_restricted_mode():
      raise OperationError(
        "Object %s is read-only" % self)
    return handler(self, name, attr)
  def __nativecall__(nativemethod, self,
                     args, readonly):
    if not readonly and in_restricted_mode():
      raise OperationError(
        "Object %s is read-only" % self)
    return nativemethod(*args)
```

Figure 8. ReadonlyPolicy: prevents restricted modules from changing protected objects

Highly reflective languages such as Smalltalk, Ruby, PHP and Javascript (which are also known as dynamic languages) are more suitable for InvisiType than less reflective languages. In those languages, every access to an object goes through the type dispatch mechanism, which may be intercepted by the InvisiType runtime to override a default behavior. This allows, for instance, the addition of safety checks for attribute access by overriding an attribute access operation.

It is not possible to utilize the full power of the InvisiType concept in less reflective languages like Java or C#. Since only method invocations use the type dispatch mechanism in these languages, the InvisiType idea is only applicable to adding safety checks to method invocations.

InvisiType is compatible with optimization techniques such as Just-In-Time (JIT) compilation. The JIT compiler dynamically compiles a method at runtime by specializing the method with the most common argument types. If the method is called with arguments of uncommon types, the JIT runtime falls back on the interpreter to execute the method. If an InvisiType instance is used as an argument for a method for the first time, the JIT runtime considers it as an unseen type and uses the interpreter to execute the method. When the argument of the InvisiType is used enough times, JIT will compile the method specializing with the InvisiType.

InvisiType Policy	Description
`TaintPolicy`	Restrict an object from being used in system calls and propagate taint
`ReadonlyPolicy`	Remove write access to an object in restricted mode
`WriteonlyPolicy`	Remove read access to an object in restricted mode
`NoAccessPolicy`	Remove read and write access to an object in restricted mode

Table 2. Pre-defined security policies in InvisiType

4 Security Policies

Throughout the paper, we have presented a number of security policies to illustrate the capability of the InvisiType system. These policies, summarized in Table 2, are significant in their own right because they can be used for many different applications.

The taint policy is useful not just for catching security vulnerabilities in web applications. It can be used for preventing data leakage of downloaded consumer applications. For example, a mobile application on a smart phone may wish to access some contact information. Today, an end user may be given a choice of whether to grant it access, which would also allow the application to export the information if it so pleases. With the taint policy, we can prevent the third-party application from exporting the contact information.

Here are some other scenarios in which the various policies can be used. The readonly policy can be used to ensure that a third-party application does not accidentally modify configuration information which may lead to security vulnerabilities such as allowing access to arbitrary files. The writeonly policy can be used to ensure that untrusted code can initialize a session object but not read it back. This policy guarantees that the untrusted code cannot leak information related to previous sessions to the outside. The NoAccessPolicy may be useful to hide top-secret information, like cryptographic keys, from third-party software.

5 An InvisiType Implementation

We have implemented a prototype of the InvisiType runtime system as part of the Python programming language.

5.1 PyPy Implementation

We implemented our InvisiType runtime system on PyPy. Designed to be flexible and support experimentation, PyPy is a Python interpreter written in Python itself [28]. The interpreter is written in RPython, a restricted subset of Python, which can be statically compiled. Some of the policies, `TaintPolicy` for instance, described in this paper are also written in RPython for performance consideration. The C-translated version of the PyPy interpreter we use has a performance comparable to CPython, a reference implementation for Python [27]. Our prototype is compatible with optimizations implemented in PyPy interpreter. For instance, as an optimization, PyPy interpreter emits special bytecodes for method access and built-in function access; the InvisiType runtime works with the optimizations effortlessly.

5.2 Protected Classes

As discussed in Section 3, a protected class inherits from a policy class and the class of the object it protects. A new data structure needs to be generated for each class of objects a policy protects, as illustrated in Figure 9. The protected classes are kept in a cache for the sake of reuse.

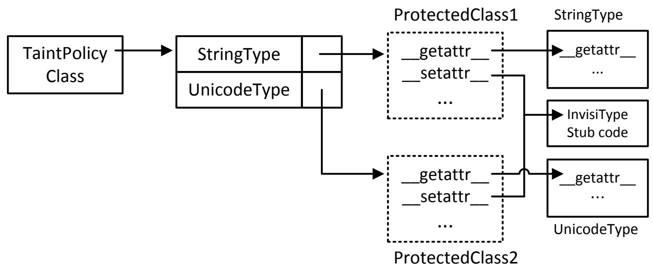

Figure 9. Examples of protected classes: normal ProtectedClass1 inherits from String type and TaintPolicy; ProtectedClass2 inherits from Unicode type and TaintPolicy. The generated protected classes are cached in the policy class.

In Python, a type has a table of handler functions for each of the standard operations in the language such as `__getattr__` and `__setattr__` for getting and setting an object attribute, respectively. We create a new table of operation handlers for each protected class. If the handler is not overridden by the policy, the handler entry simply points to the corresponding handler in the class being protected. If it is, the entry points to a stub code defined in `InvisiType`. This stub code is responsible for invoking the handler with an additional parameter: the overridden method. Similarly, the overridden virtual methods in the virtual method table for the class also point to the stub code, which again supplies the extra argument.

5.3 Demoted Objects

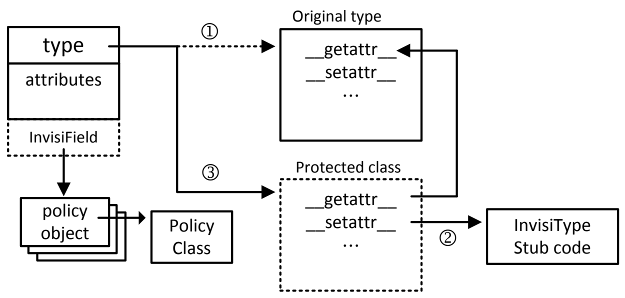

Figure 10. Object demoted to be an InvisiType instance: (1) Type pointer of the object originally points at its type. (2) When a policy is applied by calling the demote function, the InvisiType runtime creates a subclass inheriting from the applied policy class and the original type of the object. The runtime examines virtual methods in the policy class and adds stub code to the generated subclass. (3) Finally, the runtime changes the type pointer of the object to point at the generated InvisiType subclass.

Objects in our system have an additional field, called the `InvisiField`, which points to a list of policy objects that the object is currently subjected to, as shown in Figure 10. The type pointer of a demoted object points to the protected class of the lowest member of the protected class hierarchy.

A `demote` operation is implemented in the following way:

1. When an object is demoted, the runtime examines if attributes are defined in a given policy class. If so, the runtime instantiates a policy object and appends it to the list of policy objects.

2. The runtime changes the class hierarchy of a given object by inserting the protected class in the class hierarchy. The protected class is created if one does not already exists.

3. The type pointer of the object is changed to point to the protected class.

If a policy class defines attributes or overrides methods in an original class, attribute/method resolution order is modified to match the new class hierarchy. The runtime does this by overriding `__getattr__` operation handler, which handles attribute/method access.

A `promote` operation removes the given policy class from the class hierarchy as well as the corresponding policy object from the `InvisiField`.

5.4 Native Methods

The `__nativecall__` method is implemented by overriding the `__getattr__` operation handler. If the `__nativecall__` method is defined in a policy class, the protected class corresponding to this policy automatically includes an overriding `__getattr__`, the attribute access handler. When `__getattr__` detects that a native method of the object is accessed, it copies the method object and demotes it to override the `__call__` handler before returning it. The `__call__` handler invokes `__nativecall__` instead of the actual native method code.

To support the `__before_nativecall_arg__`, `__after_nativecall_arg__`, and `__syscall__` methods, we use Python's runtime type-checking mechanism. Since Python is a dynamically typed language, a native method has to check whether the arguments to the method have the correct type for the method. We augment the type-checking code to check if an argument of InvisiType subclass is passed to a native method and invoke the appropriate `__before_nativecall_arg__`, `__after_nativecall_arg__`, or `__syscall__` method accordingly.

5.5 Maintaining Transparency

We also modified the Python runtime system to hide the protected class hierarchy from the application writer. This involves making changes to only a few built-in functions such as `type(object)` and `isinstance(object, type)`.

5.6 Restricted Mode

Two built-in functions for restricted mode, `import_restricted()` and `in_restricted_mode()`, are implemented using the restricted execution support in Python. The Python Interpreter executes code in restricted mode unless the built-in namespace associated with the code is bound to the standard built-in module. `Import_restricted()` function binds built-in namespace to a cloned built-in module before importing a module, thus making the module code to run in restricted mode. Similar technique is being used to restrict capabilities in applications on Google AppEngine [31].

5.7 Summary

InvisiType allows programmers to encapsulate safety checks in an object-oriented manner, separating the concerns of safety checks from the representation and usage of the objects. `TaintPolicy`, for instance, expresses the security policy for general tainted objects. The same policy can be applied to different classes of data. Third-party software can be run with the `TaintPolicy` enforced without any code modifications.

It is interesting to contrast this approach to code rewriting systems that add instrumentation code to the bytecode. Our approach has the advantage that InvisiType adds overhead to only the selected object instances; bytecode rewriting adds overhead to the code invoked on all instances of the objects. Also, simplicity is another advantage of our approach. Not only is instrumentation at the bytecode level complex, but bytecode re-writing imposes a non-trivial overhead for dynamically loaded code.

6 Experimental Results

Besides implementing the security policies described above, we have applied InvisiType to four widely-used open-source Python web applications to find cross-site scripting vulnerabilities and access control errors. We successfully found bugs in every one of these applications with relatively little effort, while incurring a low runtime overhead.

6.1 Applications

We have chosen a set of widely-used Python web applications that allow third-party extensions. Table 3 shows the number of lines of code in the standard distributions of these platforms, which in some cases also include a few of the more popular extensions.

1. PyBlosxom is a personal weblog system. Over 90 extensions have been developed for this platform; the notable ones include Lucene extension which searches weblogs, and a photo-gallery extension that shows photos for a weblog entry.

2. ViewVC is a tool for viewing the contents of CVS and SVN repositories using a web browser. ViewVC has extensions for third-party source repository systems such as the popular, commercial Perforce system.

3. Roundup is a web-based issue/bug tracking system. It has well-defined interfaces for extensions to allow project-specific customization.

4. MoinMoin is a large and very popular Wiki system. MoinMoin provides a central repository for extensions [22]; over 200 third-party extensions have been developed to date.

Application	**Description**	**Source LOC**
PyBlosxom	A lightweight file-based weblog system	5,999
ViewVC	Web interface for version control repositories	14,964
Roundup	Issue tracking system	43,788
MoinMoin	Extensible Wiki Engine	92,438

Table 3. Summary of applications

MoinMoin is substantially bigger than the rest and has the largest number of extensions. We have conducted a comprehensive case study of this application and will present it in Section 6.4, after describing the general results on all these applications as a whole.

6.2 Security Errors

All these applications are known to have XSS errors. To catch XSS errors, we needed to modify only the standard Python library to implement a taint policy like the one in Figure 2. We did not need to change the software platform code at all.

Roundup and MoinMoin are known to also have access control errors. We implemented access control policies for Roundup and MoinMoin in a manner similar to that described in Figure 7. They require changing less than 100 and 150 lines of code in the software platform, respectively. No changes are made to any of the extensions.

Table 4 shows the number of previously reported XSS and access control violations for each of the four applications [23, 26, 29, 30, 34, 36]. Running these applications with inputs known to trigger these errors, we found that the InvisiType policies are successful in catching every one of these errors.

Application Name	# of Security Bugs	
	XSS	**ACL**
PyBlosxom	3	0
ViewVC	1	0
Roundup	2	3
MoinMoin	13	3

Table 4. Number of security bugs in the applications

6.3 InvisiType Policy Overhead

To determine the overhead of the InvisiType approach, we measure the time the applications take without InvisiType, then with the InvisiType extension, and finally with all the policies enforced. Table 5 reports the measurements of the overhead averaged over ten runs. We found that the overhead imposed by the InvisiType system in the Python interpreter is negligible, being less than 1.5% in all the four applications. The overhead due to the taint and access control policies is also minor, with the largest still under 4%.

Application	InvisiType Overhead	Policy Overhead
PyBlosxom	0.5%	4.0%
ViewVC	0.3%	0.5%
Roundup	1.3%	3.7%
MoinMoin	1.5%	0.7%

Table 5. Overhead of the InvisiType system and security policies.

As a stress test, we wrote a taint micro-benchmark that does nothing but manipulate various-sized strings. The overhead of the InvisiType implementation itself remains less than 1.5%. If none of the strings are considered tainted, then there is no additional overhead. If we taint all strings, then the performance degrades by as much as 14.1%. This illustrates the power of the InvisiType system where overhead is incurred only on those instances that need protection. If the overhead was uniform for all objects of the same type, the web applications we studied would have a very high overhead since they manipulate mostly strings.

6.4 MoinMoin Case Study

In this section, we describe our experience in applying InvisiType policies to the MoinMoin Wiki system [21]. We chose MoinMoin for our case study not only because it is the biggest and has the most extensions, but since there was a previously reported attempt to secure MoinMoin, we are able to provide a comparison with previous work.

6.4.1 MoinMoin Overview

MoinMoin is a popular open-source wiki system implemented in Python. Many well-known communities such as Apache, Debian, Ubuntu and Python use MoinMoin as a collaborative documentation tool.

MoinMoin started as a simple wiki engine; in its first release, MoinMoin had 11,000 lines of code and had no support for third-party plugins. As it became popular, MoinMoin grew quickly. As of version 1.8.0, MoinMoin is as large as 92,000 lines of code, with support for third-party macros/plugins and access control for wiki pages.

This rapid growth in size and complexity led to several security issues. The MoinMoin official web site shows that 24 security bugs have been found since 2007 [23]; this number includes bugs found in the MoinMoin distribution alone. It is likely that hundreds of third-party plugins have similar security issues. Sixteen out of the 24 reported bugs fall into two categories: cross-site scripting vulnerabilities and access control errors.

6.4.2 Taint Tracking in MoinMoin

To prevent cross-site scripting, we started by implementing the taint policy described in Figure 2. We soon found that tainting every string received from a socket is too strict. The reason is that the `escape` method in the `cgi` module is not the only way to sanitize a string. More specifically, MoinMoin uses Python's standard HTTP server library to process HTTP requests. The library checks if received requests conform to the HTTP request syntax; doing so automatically ensures that Javascript is not included in the request, making it an effective sanitizer. As a matter of fact, since all request URIs are echoed back with the content of a page, treating HTTP requests as tainted would raise a false alarm each time a page is displayed.

This shows that the InvisiType system works only as well as the policies. Fortunately, the false alarms make it relatively easy for the platform developer to tighten up the specification of the policy. We can fix this problem by modifying the `BaseHTTPRequestHandler` class in the HTTP server in the Python standard library, as shown in Figure 11. After the `parse_request` method finishes parsing and syntax checking, we can safely promote the parsed strings back to the normal string type.

```
class BaseHTTPRequestHandler:
  def parse_request(self):
    ...
    requestline = self.raw_requestline

    # Syntax check of HTTP request

    self.command=promote(command,
                         TaintPolicy)
    self.path=promote(path, TaintPolicy)
    self.request_version=promote(version,
                             TaintPolicy)
```

Figure 11. Untainting the strings that are checked by the HTTP Server library

6.4.3 Access Control Checking for Wiki Pages

Wiki pages in MoinMoin have access control lists (ACL) that manage which users or groups can access the page. For example, an ACL of a page can specify that Alice can read and write the page, but all other users have only read access. The access control in MoinMoin is implemented by requiring developers to call an access control checking function before reading from or writing to a wiki page. As a result, the access control checking code is scattered throughout the code; as of version 1.8.0, access control checks are found in 76 different places across 30 different files.

The implementation of access control checking has been a source of security problems since it is very easy for developers to forget to insert the checking code. According to the MoinMoin web site, there are 3 security bugs caused by disregarding ACL in wiki pages [23]. Even a standard plugin `INCLUDE` in the MoinMoin distribution had this problem. `INCLUDE` includes a wiki page inside another wiki page; however, the plugin failed to check the access control list associated with the included wiki page.

Before explaining how we enhanced the security of MoinMoin's access control implementation, we first describe the current access control implementation. Figure 12 shows the `Page` class representing a wiki page and the `AccessControlList` class representing access control list for a wiki page. The handler of each HTTP request is given its own `Page` object. It must first call `getACL` to get the access control list for the requested wiki page; it then invokes `acl.may (request, name, "read")` where `request` is the current HTTP request and `name` is the user's login name; if the access control check succeeds, then it can invoke `get_raw_body` to retrieve the content of the wiki page. The added complication is that the `getACL` method also invokes `get_raw_body` to retrieve the ACL associated with the page. Hence, to enforce access control check we should restrict the invocation of `get_raw_body`, except when called by the `getACL` method, until `acl.may (request, name, "read")` successfully passes access control.

We enforce access control checking by defining `AccessControlPolicy`, as shown in Figure 13, and subject all pages to the access control policy until their ACLs have been checked.

To handle the complication that `getACL` needs to invoke `get_raw_body`, we define an attribute `accessingACL` to record the context whether `getACL` is currently executing for that page. `get_raw_body` is redefined so that it will throw an exception if it is invoked, not by `getACL`, on a protected object.

Every page is guarded by an AccessControlPolicy object when it is created. In the constructor `__init__` of the `Page` class, the page instance `self` is demoted with its own instance of the `AccessControlPolicy` class. An object

```
class Page(object):
    def get_raw_body(self):
        # returns the raw body of the text page
        ...
    def getACL(self, request):
        ...
        acl = self.parseACL()
        ...
        return acl

class AccessControlList(object):
    def may(self, request, name, dowhat):
        # returns True if user with name has
        # access to read, write, delete, etc
        ...
```

Figure 12. MoinMoin's Page class and AccessControlList class

is promoted back to an ordinary page when the access control check is performed successfully by the `may` method.

We have so far described how we enforce access control on read accesses; similar policies can be defined to protect the write accesses. We were able to detect all of MoinMoin's known access control bugs. More importantly, our system can guard against all the access control errors in the hundreds of existing MoinMoin plugins, without requiring any code changes be made.

6.4.4 Comparison With Previous Work

Krohn et al. previously attempted to enhance MoinMoin's security using Flume [17]. Flume's approach is based on OS-level information flow control; they extracted the login module and made it a separate process. However, Flume does not address cross-site scripting vulnerabilities since it is not possible with OS-level information flow control. Flume cannot be applied to taint tracking which needs fine-grained information flow control. Moreover, their approach requires more effort than ours. We used less than 150 lines of Python code to define and apply the two policies, but Flume required modifying 1,000 lines of Python code and adding 1,000 extra lines of C++ code. We also modified the Python interpreter by adding about 2,000 lines of RPython code, but this requires significantly less effort than modifying the operating system as in Flume.

7 Related Work

There has been much interest recently in using information flow control for improving security on software sys-

```
class AccessControlPolicy(InvisiType):
  def __init__(self):
    self.accessingACL = False
  def get_raw_body(method, self):
    if not self.accessingACL:
      raise Exception("Illegal read access")
    # invoking get_raw_body method
    # in Page class
    return method(self)
  def getACL(method, self):
    self.accessingACL = True
    # invoking getACL method in Page class
    result = method(self)
    self.accessingACL = False
    return result

class Page(object):
  def __init__(self, request, pagename,
               **kws):
    ...
    demote(self, AccessControlPolicy)

class AccessControlList(object):
  def may(self, request, name, dowhat):
    # after authorized to read the page,
    # we promote the page object back.
    promote(self.page, AccessControlPolicy)
```

Figure 13. Enforcing access control checking for wiki pages

tems. Asbestos [6] and Histar [37] incorporated Decentralized Information Flow Control (DIFC) into new operating systems. Flume [17] implemented DIFC in the Linux operating system. These projects control information flow between OS entities such as processes and threads. InvisiType enables applying security policies at a finer-grained level. For instance, `TaintPolicy` introduced in this paper supports information flow control within a process allowing enforcement of the policy on each object in memory.

Myers and Liskov introduced JFlow and its successor JIF, which are Java-based programming languages with DIFC support [24, 25]. JIF allows labeling variables to specify access permissions, and enforces it with static analysis, thus supporting information flow control. When properly used, JIF guarantees that there is no information leakage in an application written in JIF while InvisiType cannot give a similar guarantee. However, to take advantage of it, legacy applications need to be re-written using JIF. Our scheme allows developers to enforce policies to legacy applications with minimal modifications; our experience with MoinMoin shows that the amount of effort is relatively small.

Guard interface and *GuardedObject* class are used to enforce access control policy in Java security architecture [9]. An instance of *GuardedObject* embeds an object to be protected and a *Guard* object. The *Guard* object represents an access control policy for the protected object. When the protected object is requested to be retrieved, the *Guard* object is called upon to check against the access control policy. This mechanism can enforce access control on individual objects like InvisiType. However, it cannot describe security policies like `TaintPolicy` described in Section 2.2, where code needs to be injected into specific methods or operators.

Sekar et al. introduced the Model-Carrying Code (MCC) approach, where a security model is extracted from an application and the user of the application determines a security policy that is compatible with the model [32]. At runtime, MCC guarantees that the selected security policy is not violated by the application. MCC uses the Behavior Monitoring Specification Language (BMSL) to capture important security events, such as entering and exiting system calls, and describes security policies with regular expression patterns. Compared to their work, InvisiType adopts a more object-oriented approach. InvisiType policies are defined as a class hence policies may inherit from other policies; policies are applied by overriding the default behaviors of objects to be protected. As a result, InvisiType provides better abstraction for security policies.

Bytecode re-writing is a technique that modifies bytecode at static time or at runtime. There has been much interest in this area and there are many generic bytecode re-writing frameworks [3, 4, 13]. Welch and Stroud used the binary re-writing technique to enforce security policies on mobile code [35]. However, our approach differs from those works in that we implement security policies at the type system layer. This makes the implementation relatively simple and does not incur runtime translation overhead.

Aspect Oriented Programming (AOP) is a programming paradigm which allows specifying cross-cutting concerns across multiple classes [15]. There has been much research in applying AOP to security [5, 33]. Although there are similarities in that both allow modularizing security concerns, our work differs from these approaches in significant ways. First, InvisiType is capable of enforcing policies on individual objects, whereas AOP implementations only allow describing aspects for classes. This is essential for security policies like the taint policy, which requires safety checks be added at object granularity. Also, InvisiType allows the removal of policies when they are no longer necessary. Thus, no overhead is incurred by unused security policies. AOP implementations use bytecode re-writing and even when policies are not used, there is the overhead of inserted bytecodes.

The Inlined Reference Monitor (IRM) approach is to modify an application to include reference monitors, which observe the execution of the application and take actions on operations that violate a policy [7, 8]. Transactional Memory Introspection (TMI) is a reference monitor architecture that builds on software transactional memory [2]. In the TMI architecture, reference monitors are invoked when security-related resources are accessed inside transactions. InvisiType is a more object-oriented technique than IRM or TMI since safety checks are encapsulated in policy classes and the policies are applied to objects by changing the type of the objects. However, our technique is only applicable to object-oriented languages while IRM and TMI do not have such a restriction.

Program Query Language (PQL) allows expressing application-specific design rules and enforces them via static and dynamic analysis [19]. Although PQL's pattern-matching based query is capable of expressing many application-specific policies, some policies may not be easily expressed. PQL does not provide any means to examine the value of an object. This makes it difficult to express for instance, MoinMoin access control checking policy. Also, PQL only allows expressing error patterns; the only way to enforce a good pattern is to query for every pattern that violates the good pattern. Finally, PQL does not support untrusted domain, which is necessary to enforce policies on third-party code.

8 Conclusion

Third-party software is used across many software platforms today. To help ensure that third-party software follows fine-grained security policies, we have proposed a general type extension concept to object-oriented programming languages called InvisiType.

InvisiType allows safety policies to be encapsulated in an object-oriented manner. The platform developers specify all the safety checks in a class, and can selectively and dynamically decide which object instances are to be subjected to safety checks. No changes need to be made to third-party software, at either the source or binary level.

The InvisiType runtime system uses the efficient virtual method dispatch mechanism to enforce the execution of these safety checks on selected objects. Unlike source or binary level instrumentation, the overhead of safety checks is applied only to those instances considered at risk. This implementation makes it efficient to provide low-level safety checks, even at the granularity of individual attribute accesses. Thus, fine-grained control like taint and access control can be provided efficiently with very little overhead.

Another contribution of this paper is in the security policies themselves. These policies are general and can be used across many applications. We have demonstrated the usefulness of this approach by showing that we can secure large, real-life applications with relative ease and low overhead.

Acknowledgment

This paper is supported in part by the NSF POMI (Programmable Open Mobile Internet) 2020 Expedition Grant No. 0832820, NSF Grant TRUST CCF-0424422, and a Samsung scholarship. The authors thank John C. Mitchell, Sudheendra Hangal, Michael Barrientos, Ben Dodson, and Debangsu Sengupta for their help with the paper.

References

[1] C. Anley. Advanced SQL injection in SQL server applications. http://www.nextgenss.com/papers/advanced_sql_injection.pdf, 2002.

[2] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 223–234, Oct. 2008.

[3] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 313–336, June 2000.

[4] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 167–178, June 1998.

[5] B. De Win, W. Joosen, and F. Piessens. Developing secure applications through aspect-oriented programming. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 633–650, March 2004.

[6] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30, Oct. 2005.

[7] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, New York, 2004.

[8] U. Erlingsson and F. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, May 2000.

[9] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley Professional, 2003.

[10] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: an overview of the new security architecture in the Java development kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Dec. 1997.

[11] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, May 2004.

[12] W. G. J. Halfond and A. Orso. Amnesia: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183, Nov. 2005.

[13] R. Keller and U. Hoelzle. Binary component adaptation. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 307–329, July 1998.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, June 2001.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. JEAN-MARC Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[16] S. Kost. An introduction to SQL injection attacks for Oracle developers. http://www.net-security.org/dl/articles/ IntegrigyIntrotoSQLInjectionAttacks.pdf, 2004.

[17] M. Krohn, M. Brodsky, M. Kaashoek, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 321–334, Oct. 2007.

[18] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, Aug. 2005.

[19] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, Oct. 2005.

[20] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *The 1st Asian Symposium on Programming Languages and Systems*, pages 105–121, Nov. 2003.

[21] MoinMoin. http://moinmo.in.

[22] MoinMoin macro market. http://moinmo.in/MacroMarket.

[23] MoinMoin security fix announcements. http://moinmo.in/ SecurityFixes.

[24] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 129–142, Oct. 1997.

[25] A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[26] PyBlosxom contributed packages comments plugin multiple cross-site scripting vulnerabilities. http://www. securityfocus.com/bid/18292.

[27] PyPy performance benchmark. http://morepypy.blogspot.com/2008/05/general-performance-improvements.html.

[28] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications Companion*, pages 944–953, Oct. 2006.

[29] Roundup cross-site scripting vulnerability. http://secunia.com/advisories/9371.

[30] Roundup multiple vulnerabilities. http://secunia.com/ advisories/29336.

[31] Paving the way to securing the Python interpreter. http://tav.espians.com/paving-the-way-to-securing-the-python-interpreter.html.

[32] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 15–28, Oct. 2003.

[33] J. Viega, J. Bloch, and P. Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, 2001.

[34] ViewCVS's cross-site scripting bug. http://www.derkeiler.com/Mailing-Lists/Securiteam/2002-05/0082.html.

[35] I. Welch. Using reflection as a mechanism for enforcing security policies on compiled code. *Journal of Computer Security*, 10(4):399–432, 2002.

[36] XML injection in PyBlosxom. http://seclists.org/ fulldisclosure/2009/Feb/0084.html.

[37] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 263–278, Nov 2006.