Finding Error Handling Bugs in OpenSSL using Coccinelle

Julia Lawall*, Ben Laurie†, René Rydhof Hansen‡, Nicolas Palix* and Gilles Muller§
*University of Copenhagen, Email: {julia,npalix}@diku.dk
†Google, Email: benl@google.com
‡Aalborg University, Email: rrh@cs.aau.dk
§INRIA-Regal, Email: Gilles.Muller@lip6.fr

***Abstract*—OpenSSL is a library providing various functionalities relating to secure network communication. Detecting and fixing bugs in OpenSSL code is thus essential, particularly when such bugs can lead to malicious attacks. In previous work, we have proposed a methodology for finding API usage protocols in Linux kernel code using the program matching and transformation engine Coccinelle. In this work, we report on our experience in applying this methodology to OpenSSL, focusing on API usage protocols related to error handling. We have detected over 30 bugs in a recent OpenSSL snapshot, and in many cases it was possible to correct the bugs automatically. Our patches correcting these bugs have been accepted by the OpenSSL developers. This work furthermore confirms the applicability of our methodology to user-level code.**

***Keywords*-bug finding, OpenSSL, Coccinelle**

I. Introduction

OpenSSL is an open source implementation of the ubiquitous Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols essential for providing secure communication over the Internet. OpenSSL is available for a large number of operating systems, and is widely distributed and deployed on many platforms. It has also been used as an essential building block in a number of projects related to secure communication infrastructure, e.g., OpenVPN[1] and `mod_ssl`.[2] This makes OpenSSL a tempting target for attackers. It is thus vital that bugs are found and fixed as early in the development cycle as possible, hopefully before they can be used for malicious attacks.

In our previous work [1], we have developed a multi-step strategy for finding API function usage protocols in C code, and then finding uses of these API functions that do not follow these protocols. This strategy is derived from our study of Linux kernel code, in which we have observed that there are stereotypical code fragments controlling API function usage that characterize a wide range of API functions, such as the mechanism for reacting to detected errors or the means of managing allocated memory. Based on these observations, we use the Coccinelle program matching and transformation tool [2] first to develop and apply patterns that characterize these stereotypical code fragments, to find functions that have properties that are considered to be of interest, and then to develop patterns that describe typical incorrect usages of these functions, to find bugs in the code.

User-level code, like OpenSSL, and kernel-level code, like Linux, however, have different constraints, and each software project has its own coding conventions. Thus, it is not clear that the same patterns and bug finding strategies are applicable to both. Indeed, initial experiments with OpenSSL carried out by a student in our group [3] showed that patterns searching for problems such as duplicate testing of a value for NULL that frequently indicate bugs in Linux code either turned up no matches in OpenSSL or turned up matches that were not considered to be real bugs by the OpenSSL community. In this work, on the other hand, we have focused on a bug type that was previously highlighted in OpenSSL code by the security vulnerability report CVE-2008-5077,[3] at the suggestion of the second author, who is an OpenSSL expert. This vulnerability is based on the observation that many functions in OpenSSL return non-positive integers to indicate various kinds of errors. Nevertheless, callsite error-checking code often considers only 0 to be an error and any other value to be success. This pattern is not found in Linux, and thus OpenSSL-specific knowledge is needed.

We have detected around 30 bugs in a recent (September 11, 2009) snapshot of OpenSSL (openssl-1.0.0-stable-SNAP-20090911). In many cases, we were able to use the program transformation capabilities of the Coccinelle engine to automatically correct these bugs. Most of the corresponding bug fixing patches have been been submitted to the OpenSSL developers, and all of these submissions have been accepted. In the rest of this paper, we present Coccinelle (Section II) and then present our approach and its instantiation for finding error handling bugs in OpenSSL (Section III). We then present our results in detail (Section IV), and then briefly consider related work and conclude (Sections V and VI).

II. Coccinelle

Coccinelle is a program matching and transformation engine, targeting C code. Coccinelle provides the Semantic Patch Language (SmPL) for specifying program matches and transformations as *semantic patches*. A feature of semantic patches is that they are very close to C code, or more precisely

[1] http://www.openvpn.net/
[2] http://www.modssl.org/
[3] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5077

to patches on C code, and thus are easy for C programmers to develop and then refine, as new needs appear. In this section, we present some features of SmPL, as are needed in this paper. A complete grammar of SmPL is provided at the Coccinelle website (http://coccinelle.lip6.fr).

A simple semantic patch: Our goal is first to identify functions that may return both 0 and a negative result to indicate an error, and then to find calls to such functions that only test whether the result is 0. To present SmPL, we assume that we have already identified such a function, `EVP_VerifyFinal`, *i.e.*, the function mentioned in CVE-2008-5077, and we would like to find and fix cases where the result of calling this function is compared to 0 directly. In this case, the fix is to convert a test that the result is equal to 0 to a test that the result is less than or equal to 0, and to convert a test that the result is not zero to a test that the result is greater than 0. A semantic patch rule that expresses the first of these transformations is shown in Figure 1. A semantic patch rule that expresses the second is similar. The complete semantic patch consists of these two rules.

```
1  @@ expression list args; @@
2  -   EVP_VerifyFinal(args) == 0
3  +   EVP_VerifyFinal(args) <= 0
```

Figure 1. A simple semantic patch rule

A semantic patch rule consists of two parts: a set of metavariable declarations and a code matching and transformation specification. The set of metavariable declarations is delimited by `@@` (line 1). Metavariables can represent arbitrary code fragments, and are declared according to the kind of code they represent, *e.g.*, identifier, expression, statement, expression of a specific type, etc. In this example, there is only one metavariable, `args`, which represents a function argument list. The matching and transformation specification then consists of a fragment of C-like code, where subterms annotated with − should be removed and subterms annotated with + should be added (lines 2-3). In this case, we remove any call to `EVP_VerifyFinal` where the result is tested to be equal to 0, and then replace it by constructing the same call, using the binding of the metavariable `args`, and a test whether the result is less than or equal to 0.

In addition to matching and transforming atomic fragments of code, as shown here, Coccinelle can also match and transform scattered fragments of code connected by an execution path. This feature will be illustrated later.

Isomorphisms: The semantic patch rule shown above explicitly compares the result of calling the function `EVP_VerifyFinal` to 0, with the function call being on the left of == and 0 being on the right. A test for 0 can, however, have other forms, *e.g.*, the 0 can appear on the left, or it can be omitted and the ! operator can be used instead. It would be tedious for the rule developer to explicitly specify all of these variants. Thus, Coccinelle provides the notion of *isomorphisms*, which describe patterns of terms that should be considered to have the same semantics. Isomorphisms are defined in an external file, and new isomorphisms can be specified by the user. The isomorphisms that are relevant to the semantic patch rule shown in Figure 1 are defined below:

```
1  Expression
2  @commeq@
3  expression E; constant C;
4  @@
5  E == C <=> C == E
```

```
1  Expression
2  @ is_zero @
3  expression X;
4  @@
5  X == 0 => !X
```

The first rule uses <=> to indicate that any comparison expression in a semantic patch that has a constant on the right should be considered to match a term with the constant on the left, and vice versa. The second rule indicates that a comparison to 0 in a semantic patch can be considered to match a boolean negation, but, because of the use of => rather than <=>, the inverse as not allowed, as it would only be valid when the negated expression is an integer.

Refining a semantic patch: Our experience with versions of the above semantic patch rule considering other functions shows that sometimes the rule does not match, because the developer has enclosed the function call in parentheses. We can thus extend the above semantic patch rule to take this possibility into account, as shown in Figure 2.

```
1  @@ expression list args; @@
2  -   (EVP_VerifyFinal(args)) == 0
3  +   EVP_VerifyFinal(args) <= 0
```

Figure 2. A refined version of the simple semantic patch

Another isomorphism ensures that this rule also matches cases where the parentheses are not present.

Position variables and Python: The semantic patch rules we have presented both find and fix bugs. Such a rule is only possible when there is both a general pattern for finding a bug and a general pattern for fixing it. In some cases, however, there is only a general pattern for finding the bug, and the fix has to be performed manually, on a case by case basis. In this case, it is useful to report information on where the bug was found in the source code. For this, we use *position variables* to record the positions of relevant terms and then embed Python code in the semantic patch to print this position information in the desired format.

The extension shown in Figure 3 of the above semantic patch rule first records the position of the call to `EVP_VerifyFinal`, and then uses a library function defined by Coccinelle to print a hyperlink to this position in the source code according to the Emacs Org mode notation.[4] This hyperlink makes it easy to access the affected code from the bug report and make any fixes that are required.

In line 1 of Figure 3, the `@@` preceding the metavariable declaration now specifies the name of the rule, `r`. This name allows the rule's metavariables to be referenced by other rules.

[4]http://orgmode.org/

```
1  @r@ expression list args; position p; @@
2   (EVP_VerifyFinal@p(args)) == 0
3
4  @script:python@ p << r.p; @@
5  cocci.print_main("zero",p)
```

Figure 3. Using Python

Rule r now additionally declares the metavariable p, which represents a code position. Such a position variable can be used to record information about the position of any matched token in the C code. The token of interest is indicated using the operator @. In our example, we record the position of each matched occurrence of EVP_VerifyFinal. The Python code then inherits this position variable (line 4), and uses its value in the subsequent call to cocci.print_main. This function prints information about the position of each call to EVP_VerifyFinal in Org mode format.

III. The Bug-Finding Process

We find bugs using a two-step process [1]: 1) identify functions that may return negative values and 2) identify call sites where the error detection code appears to be incorrect. There is a separate semantic patch for each step. Essentially, the first semantic patch creates a list of names of functions that may return a negative result and this list is then used to instantiate the second semantic patch to find bugs in the error detection code at the callsites of each function. We describe these semantic patches in detail below.

A. Step 1: Finding functions that may return a negative result

This step finds functions that somewhere return a negative value, of any sort. It proceeds in two substeps: first we perform some normalization, and then we detect functions that may return a negative value.

Normalization: Some functions return various constant values directly. Others store the result in a variable, and then return that value, or return the result of some other computation. In the latter cases, the code must be analyzed to infer that the return value of a function may be negative. Because Coccinelle does not incorporate any dataflow analysis (*i.e.*, propagation of variable values to variable references), this analysis must be encoded in a semantic patch directly. We thus start with a normalization phase that makes the presence of negative values explicit in the source code.

A typical example of the need for this normalization is the following code extracted from the function AES_set_decrypt_key in acrypto/aes/aes_x86core.c:

```
1  status = AES_set_encrypt_key(userKey, bits, key);
2  if (status < 0)
3    return status;
```

The semantic patch rules performing the normalization transform this code into the following:

```
1  @rn exists@
2  identifier f; position ret_neg; expression E,E1; constant C;
3  @@
4
5  f(...) {
6  <+...
7  (
8    return@ret_neg (-C);
9  |
10   E = -C
11   ... when != E=E1
12   return@ret_neg (E);
13 )
14 ...+>
15 }
16
17 @ script:python @ f << rn.f; @@
18 print "category1: FN:%s" % f
```

Figure 4. Finding functions that return a negative result

```
1  status = AES_set_encrypt_key(userKey, bits, key);
2  if (status < 0) {
3    status = -1;
4    return status;
5  }
```

The normalization rules first identify conditionals that contain comparisons to 0, then ensure that the branches of these conditionals are delimited by braces, and finally insert assignments to -1 in these branches as needed. These rules amount to around 70 lines of semantic patch code.

Detecting functions that may return a negative value: Once negative values are explicit, we can write rules to find and print the names of functions that return a negative value along some path in their control-flow graph.

The rules are shown in Figure 4. The first rule, rn, searches throughout the body of each function f, as indicated by the *nest* notation <+... ...+> (lines 6 and 14), to find at least one occurrence of either a return of a negative constant (line 8) or an assignment of an expression to a negative constant, such that that expression is eventually returned as the result of the function (lines 10–12). The relationship between the assignment and the return is indicated using the notation "..." (line 11) meaning that there should be a path in the function's control-flow graph connecting the assignment to the return. The annotation when != E=E1 specifies that the expression matching E should not be reassigned within this path. The second rule, starting on line 17, is a Python rule that inherits the name f of the function from rule rn and prints out this name in the format expected by the second step. In particular, the printed string indicates that the semantic patch rules used to find bugs should be instantiated by replacing FN by the name of the identified function.

Refinements: In studying the initial results of finding bugs in the uses of the collected functions, we noticed some frequently occurring false positives, *i.e.*, bug reports for calls to functions that do not actually use both 0 and negative values to indicate an error condition. One case is when the called function is a comparison function, returning -1

```
1  @@ expression list args; @@
2  -   (FN(args)) == 0
3  +   FN(args) <= 0
4
5  @expression@ expression list args; @@
6  -   (FN(args)) != 0
7  +   FN(args) > 0
```

Figure 5. Finding bugs when the result of a function call is tested directly

when the first argument is less than the second, 0 when the arguments are identical, and 1 otherwise. In OpenSSL such functions seem to have names ending with `cmp`, and thus we use Python to discard matches where the name of the matched function has this form (not shown). Another case is when the called function uses negative values on an error and 0 on success. While it would be possible to perform bug detection for such functions, doing so is out of the scope of this project, and thus we add some SmPL rules (not shown) to detect the case where a positive value is never returned.

The ability to make these refinements illustrates the flexibility of the Coccinelle approach. When a frequent category of false positives is identified, the semantic patch can be rewritten to filter out this case. Furthermore, it is possible to write another semantic patch that only matches the false-positive category, to allow studying these cases in more detail, in case one of them turns out to be a real bug.

B. Step 2: Finding incorrect error detection at call sites

Given the list of functions that may return 0 or a negative number to indicate an error, we now detect callsites of these functions where the error detection code only checks whether the result is 0. In that case a negative value, which is nonzero, will cause the error to be overlooked.

The simplest instance of this bug is when the result of the call is compared to 0 directly. In this case, we can both detect the bug and correct it automatically,[5] as shown in Figure 5. These rules generalize the one presented in Figure 2. Here, `FN` is instantiated to each of the functions identified in the previous step, by a tool that is external to Coccinelle [1].

A more complex instance of the bug is when the result of the call is saved in some location. In this case, it may be tested multiple times: first for being 0, then for being negative, etc. We are interested in cases where somewhere between the initial assignment to the location and the next assignment to the location (or the end of the function, if there is no subsequent assignment) there are no tests for negative numbers and there are tests for 0. The rules that implement this are shown in Figure 6.

The first rule, tested (lines 1-13), identifies some cases where the result of calling the function is both stored in a variable and tested to be either positive or non-positive at the same time. We consider these cases to be correct, as they

[5]The user should nevertheless always check the result.

```
1  @tested@
2  expression x; constant C; position p1;
3  @@
4
5  (
6   (x@p1 = FN(...)) <= ( 0 | -C ) // comparison with 0 or with -C
7  |
8   (x@p1 = FN(...)) < ( 0 | -C )
9  |
10  (x@p1 = FN(...)) > 0
11 |
12  (x@p1 = FN(...)) == -C
13 )
14
15 @match@
16 expression x, E; position p1!=tested.p1,p2; constant C;
17 @@
18
19 x@p1 = FN(...)
20 <... when != x <= ( 0 | -C )
21       when != x < ( 0 | -C )
22       when != ( x > 0 | x == -C )
23  ( x@p2 != 0 | x@p2 == 0 )
24 ...>
25 ( return ...; | x++ | x-- | x += E | x -= E | x = E )
26
27 @script:python@ p1 << match.p1; p2 << match.p2; @@
28 cocci.print_main("FN",p1)
29 cocci.print_secs("test",p2)
```

Figure 6. Finding bugs when the result of a function call is stored in a variable

acknowledge that positive and non-positive are the cases of interest. The position of such an assignment is recorded in the position variable `p1`. The second rule, which finds bugs, then only considers calls that are at other positions than the ones found here.

The second rule, match (lines 15-26) matches the case that is considered to be a bug: a variable is initialized to the result of calling a function that may return a negative value to indicate an error, and this variable is subsequently only tested for being equal to or not equal to 0. This rule saves in position variables the position of the initial assignment (line 19) and the position of any zero or non-zero tests (line 23). The `when` declarations inside the nest (`<... ...>`) ensure that there is no test for being less than or greater than 0, or equal to a negative constant.

The semantic patch ends with Python code that prints the locations of the initial assignment and the zero tests. A Python rule is only executed if all of its variables, *i.e.*, both `p1` and `p2`, are bound, and thus we only get output when there is at least one comparison with 0.

IV. Results

We now present the results of applying our approach to the September 11, 2009 snapshot of OpenSSL (openssl-1.0.0-stable-SNAP-20090911). This snapshot contains around 250 000 lines of C code, as calculated using SLOCCount.[6] Our experiments were carried out on a HP ProLiant server with two 3 GHz quad-core Xeon processors and 16 GB memory.

[6]http://www.dwheeler.com/sloccount/

	Bugs	Acc.	FP	Unk.	Files
ASN1_item_ex_d2i	0	0	1	0	1
BIO_ctrl	1	0	0	0	1
BIO_write	6	6	0	0	3
BN_exp	1	1	0	0	1
CMS_get1_ReceiptRequest	2	2	0	0	1
ENGINE_ctrl	4	4	1	0	2
EVP_PKEY_cmp_parameters	0	0	1	0	1
EVP_PKEY_sign	1	1	0	0	1
OPENSSL_isservice	1	1	2	0	3
RAND_bytes	4	3	0	0	4
RAND_pseudo_bytes	2	0	0	0	1
UI_ctrl	0	0	2	0	2
X509_STORE_get_by_subject	0	0	0	2	1
X509_check_purpose	0	0	0	1	1
asn1_cb	0	0	10	0	3
asn1_check_tlen	0	0	2	0	1
i2a_ASN1_INTEGER	1	1	0	0	1
i2a_ASN1_OBJECT	1	1	0	0	1
sk_num	0	0	3	0	1
TOTAL	26	20	20	3	30

Table I
RESULTS USING THE SEMANTIC PATCH SHOWN IN FIGURE 5.

	Bugs	Acc.	FP	Unk.	Files
ASN1_INTEGER_get	0	0	2	0	2
BIO_ctrl	1	1	0	0	1
EVP_DigestVerifyFinal	1	1	0	0	1
EVP_SealInit	1	1	0	0	1
RAND_bytes	1	1	0	0	1
SSLStateMachine_read_extract	0	0	1	0	1
UI_UTIL_read_pw	0	0	1	0	1
X509_check_purpose	0	0	1	1	2
asn1_cb	0	0	4	0	2
asn1_check_tlen	0	0	2	0	1
asn1_template_noexp_d2i	0	0	1	0	1
dtls1_retrieve_buffered_fragment	0	0	0	1	1
get_cert_chain	0	0	1	0	1
i2b_PVK_bio	2	2	0	0	2
ocsp_check_issuer	0	0	1	0	1
TOTAL	6	6	14	2	19

Table II
RESULTS USING THE SEMANTIC PATCH SHOWN IN FIGURE 6.

The entire experiment, including both steps, requires around 12 minutes.

Table I presents the reports given by the semantic patch rules that detect direct tests (Figure 5), and Table II presents the reports given by the semantic patch rules that detect tests on stored values (Figure 6). Acc. (accepted) indicates the number of bugs for which patches have been submitted to and accepted by the OpenSSL developers. FP indicates the number of false positives (reports that we believe do not represent actual bugs). Unk. (unknown) indicates reports that we were not able to classify as bugs or false positives. Overall, the rate of false positives is around 50%. This rate is high, but the overall number of reports is low. It took us a few hours to separate the real bugs from the false positives. We expect that this could be done more rapidly by someone who is more familiar with OpenSSL.

In practice, it turned out to be necessary to perform the experiments twice. The first time, we applied the approach described in Section III to the OpenSSL source code and obtained a number of reports. For these, we identified the real bugs and submitted patches, including those generated automatically by the rules shown in Figure 5 to the OpenSSL developers. All of these patches have been accepted. Subsequently, however, we realized that the OpenSSL code contains some macros that are not covered by the rules used by Coccinelle when parsing C code. These rules were developed based on our experience with Linux code, and were thus not sufficient for OpenSSL. We then added information about four OpenSSL macros, `STACK_OF`, `MS_STATIC`, `MS_CALLBACK`, and `RSA`, to the Coccinelle rule set, which appeared to cover most of the parsing problems. The second run then found 6 more real bugs, for which we have not yet had time to submit patches to the OpenSSL developers. The need to configure Coccinelle with information of the definitions of macros that do not correspond to C syntax is an inconvenience, but allows Coccinelle to retain the original structure of the source program, as opposed to a solution that relies on first calling the C preprocessor. Furthermore, our results for OpenSSL show that only a few macro definitions may have to be provided to obtain a substantial improvement in the amount of parsed code.

Most of the false positives in our results come from cases where argument values imply that the code returning negative values can never be executed. One example is the function `asn1_check_tlen`, which only returns a negative value when it receives an argument indicating that an optional value should be considered. At the reported call sites, this argument is a constant that indicates that optional values should not be considered. `asn1_check_tlen` is furthermore called by two other functions for which there are are reports, `ASN1_item_ex_d2i` and `asn1_template_noexp_d2i`. The calls to these functions also indicate that the optional values should not be considered and thus the reports for these functions are also false positives.

The function `OPENSSL_isservice` illustrates another case. This function always return one of three possible constants: -1, 0, and 1. It is essentially a predicate, where -1 indicates an error, 0 indicates false, and 1 indicates true. In this case a `!= 0` test is a bug, but an `== 0` test is legitimate. We found one of the former and two of the latter, *i.e.*, one real bug and two false positives.

Finally, for `BIO_write` we found one case in which the test was checking for a non-zero value in a context where it should have been testing for an error value. In this case, the semantic patch of Figure 5 converts the test to `> 0`. This was the wrong fix, as the context indicates that the test should be `<= 0`. The semantic patch caused us to focus on the incorrect code, allowing us to manually solve the problem.

A. History

In other work, we have developed a tool, Herodotos [4], that tracks the history of code fragments that match a given semantic patch over multiple versions of a software project.

We have applied Herodotos to OpenSSL versions 0.9.8a through 0.9.8j, released between 2005 and 2008, and to the semantic patches defined in Figures 5 and 6, to obtain a history of error handling bugs. We have found that almost all of the bugs found here have been present since at least version 0.9.8a, with a handful introduced at a later date, either due to some change in the code or the introduction of a new file. The only bugs that were corrected were in three calls to `EVP_VerifyFinal` and four calls to `X509_verify_cert`, which were related to CVE-2008-5077, and in one call to `CMS_SignerInfo_verify_content`, which was related to CVE-2009-0591.[7]

The long lifetime of the bugs found here shows the value of bug finding tools. Furthermore, as compared to other bug finding tools, Coccinelle can easily be configured via semantic patches to find new potential bug types, once they becomes apparent. Indeed all of the bugs found here could have been found and fixed at the time of the release of the first CVE, CVE-2008-5077, in January 2009.

V. Related Work

Many tools have been developed to aid in the automatic bug detection. We focus on approaches that infer API usage protocols and that have considered related bugs in OpenSSL.

Engler et al. [5] pioneered the approach of searching for usage protocols in systems code, represented by common code patterns, and then searching for deviations in these patterns, which are considered to be bugs. This approach and subsequent protocol finding approaches, such as PR-Miner [6] and the work of Ramanathan et al. [7], use statistics to identify common patterns. In contrast, our approach incorporates the user's knowledge about common patterns in the given software project. Our approach was presented in previous work in the context of Linux kernel code [1]. This paper extends it to user-level code, specifically OpenSSL.

Dillig et al. [8] propose an approach to finding "source-sink" inconsistencies, where the means of creating a value and the means of using it are inconsistent. They also consider OpenSSL, but their experiments focus on the use of values that might be NULL, rather than integer-typed error codes.

VI. Conclusion

In this paper, we have shown the applicability of a bug-finding strategy developed for Linux code to OpenSSL code. This bug-finding strategy distinguishes itself from other approaches in that it is highly automated, but relies on user expertise to create bug-finding patterns. Thus, it is sensitive to the differences between software projects. We have used our approach to find over 30 bugs in OpenSSL code, and the overall number of bug reports is manageable for manual validation. We observe that the bug finding rules developed here would not be relevant to Linux code, which has been the main focus of our previous work, because Linux functions do not use both 0 and negative values to signal an error. Other software may use other error reporting protocols. Given knowledge about these protocols, appropriate rules for such software could be developed as well.

OpenSSL is a library, and thus many of its functions are also called directly by external code. It could thus be helpful to apply our rules to such code. None of the bugs that we found in the current snapshot of OpenSSL seem to represent a serious security threat. Nevertheless, as open source code, it is useful for it to be correct and consistent, so that it can reliably serve as a model for others who want to use these functions, perhaps in a more security-sensitive context.

[7] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0591

References

[1] J. L. Lawall, J. Brunel, R. R. Hansen, H. Stuart, G. Muller, and N. Palix, "WYSIWIB: A declarative approach to finding protocols and bugs in Linux code," in *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, (DSN 2009)*, Estoril, Portugal, Jun. 2009, pp. 43–52.

[2] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *EuroSys 2008*, Glasgow, Scotland, Mar. 2008, pp. 247–260.

[3] S. Rievers, "Finding bugs in open source software using coccinelle," Aug. 2009, bachelor's project, DIKU, University of Copenhagen.

[4] N. Palix, J. Lawall, and G. Muller, "Tracking code patterns over multiple software versions with Herodotos," in *Proc. of the ACM International Conference on Aspect-Oriented Software Development, AOSD'10*, Rennes and Saint Malo, France, Mar. 2010, to appear.

[5] D. R. Engler, D. Y. Chen, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of the 18th ACM Symposium on Operating System Principles*, Banff, Canada, Oct. 2001, pp. 57–72. [Online]. Available: http://citeseer.nj.nec.com/458580.html

[6] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal, Sep. 2005, pp. 306–315.

[7] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, USA, Jun. 2007, pp. 123–134.

[8] I. Dillig, T. Dillig, and A. Aiken, "Static error detection using semantic inconsistency inference," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, San Diego, CA, Jun. 2007, pp. 435–445.