Towards Security Vulnerability Detection by Source Code Model Checking

Keqin Li
SAP Research
Sophia Antipolis, France
e-mail: Keqin.Li@sap.com

***Abstract*—Security in code level is an important aspect to achieve high quality software. Various security programming guidelines are defined to improve the quality of software code. At the same time, enforcing mechanisms of these guidelines are needed. In this paper, we use source code model checking technique to check whether some security programming guidelines are followed, and correspondingly to detect related security vulnerabilities. Two SAP security programming guidelines related to logging sensitive information and Cross-Site Scripting attack are used as examples. In the case studies, Bandera Tool Set is used as source code model checker, and minimizing programmers' additional effort is set as one of the goals.**

Keywords-model checking; source code analysis; security; programming guidelines

I. INTRODUCTION

With the increasing usage of software applications, security is more and more perceived as an important aspect of software. Achieving high quality software with respect to security asks for activities across the whole software development life cycle, rather than a late activity under tight time and resource constraints. Correspondingly, various research topics and techniques are proposed to improve software security. Among them, security in code level (source code or bytecode) is an important one. Programs often contain fatal errors despite the existence of careful designs. Many deadlocks and critical section violations, for example, are introduced at a level of detail which designs typically do not deal with, if formal designs are made at all. In the end, it is the running code in which security vulnerabilities exist or not.

In order to improve security in the code level, security programming guidelines are defined in many software development organizations. These guidelines could cover a wide range of known security vulnerabilities related to programming style, usage of certain interfaces, etc. At the same time, mechanisms to enforce these guidelines are needed. Without enforcing mechanisms, the application of these guidelines and correspondingly the software security related to these guidelines will not be guaranteed.

Source code model checking applies model checking techniques to source code to find the potential violation of expected properties. Although these are some difficulties such as complexity of programming language structure and state space explosion, several source code model checking techniques and tools are proposed and applied in different applications, some fatal errors are found [15].

In this paper, we use source code model checking to check whether some security programming guidelines are followed, and correspondingly to detect related security vulnerabilities. We use two SAP security programming guidelines as example, one of them is about logging sensitive information, while the other one is about output encoding to prevent Cross-Site Scripting attack. In these case studies, we use Bandera Tool Set [7] as source code model checker. One important goal we want to achieve is to minimize the additional effort of programmers. Generally speaking, we cannot expect the normal programmers are very skillful in formal methods, thus specifying expected properties formally is a task of security and formal method experts. At the same time, large amount and wide spread program annotations could not be welcomed.

The paper is organized as follows. Section II gives background information about Java source code model checker Bandera and the corresponding property specification language BSL. In Section III, the application of Bandera to check security guideline about secure logging is described. In Section IV, the same technique is applied for security guideline related to Cross-Site Scripting attack. Related work is discussed in Section V. Section VI concludes the paper.

II. BACKGROUND: BANDERA AND BSL

The Bandera Tool Set [7] is an integrated collection of program analysis, transformation, and visualization components designed to facilitate experimentation with model-checking Java source code. Bandera takes as input Java source code and a software requirement formalized in Bandera's temporal specification language, and it generates a program model and specification in the input language of one of several existing model checking tools (including Spin [8], dSpin [4], SMV [2], and JPF [15]). Both program slicing and user extensible abstract interpretation components are applied to customize the program model to the property being checked. When a model checker produces an error trail, Bandera renders the error trail at the source code level and allows the user to step through the code along the path of the trail while displaying values of variables and internal states of Java objects.

In Bandera, source code properties to be checked are written in the Bandera Specification Language (BSL) [3]. BSL is a source level, model checker independent language

for expressing temporal properties of Java program actions and data. The organization of BSL is depicted in Figure 1. BSL is composed of the following sublanguages:

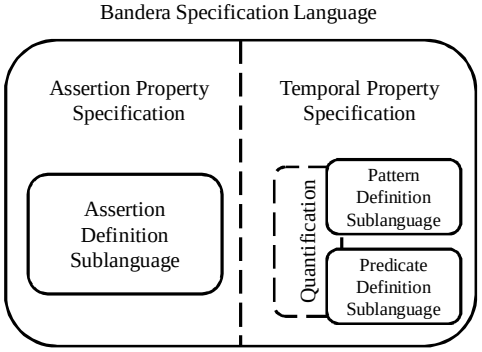

Figure 1: BSL Organization

- An *assertion* sublanguage allows users to define constraints on program contexts in familiar assertion style notation. Assertions can be selectively enabled or disabled so that one can easily identify only a subset of assertions for checking. Bandera exploits this capability by optimizing the generated models (using slicing and abstraction) specifically for the selected assertions.
- A temporal property sublanguage provides support for defining *predicates* on common Java control points (e.g., method invocation and return) and Java data (including dynamically created threads and objects). These predicates become the basic *propositions* in temporal specifications. The temporal specification language is based not on a particular temporal logic, but on a collection of field-tested temporal specification patterns [5]. This pattern language is extensible and allows for libraries of domain-specific patterns to be created.
- Interacting with both the predicate and pattern support in BSL is a powerful quantification facility that allows temporal specifications to be quantified over all objects/threads from particular classes. Quantification provides a mechanism for *naming* potentially anonymous data, and this type of support is crucial for expressive reasoning about dynamically created objects.

III. Application 1: Security Guideline about Secure Logging

One SAP security programming guideline is that before sensitive information is logged, it must be encrypted in order to prevent information leakage. In this section, we describe how to specify this guideline using BSL in order to check automatically whether this guideline is followed using Bandera.

A. Logging APIs and Encryption APIs

Logging is an important element for securing application server systems. Logs are essential for monitoring applications and tracking events if problems occur, as well as for auditing the correct usage of the system.

The SAP Logging API is provided with all functionality for events logging. The following methods of class `Category` are provided to write log messages with different severity levels. They have intuitive names that indicate the severity levels such as `FATAL`, `ERROR`, `WARNING`, `INFO`, `PATH`, and `DEBUG`.

```
fatalT(string the_message) ;
errorT(string the_message) ;
warningT(string the_message) ;
infoT(string the_message) ;
pathT(string the_message) ;
debugT(string the_message) ;
```

Using the Logging API, writing a password, which is sensitive information, to the log file could be implemented as follows.

```
Category myCat =
Category.getCategory
                ("/System/Database");
myCat.warningT("Sample message"
                 + password);
```

The last statement is where the sensitive information is written to the log file.

For more information about SAP Logging API, please refer to [12].

In SAP NetWeaver Platform, there are interfaces and classes derived from them available for implementing digital signature and encryption in the applications. We now proceed to describe them.

The interface `ISsfData` is the central interface used for the cryptographic functions. Its underlying classes specify the data format used, for example, `SsfDataPKCS7`, `SsfDataSMIME` and `SsfDataXML`. The available methods are `sign()`, `verify()`, `encrypt()`, `decrypt()`, and `writeTo()`.

The interface `ISsfProfile` provides access to the user's or server's profile, where the private key and corresponding public-key certificates are stored. If the public-key certificate has been signed by a CA, the interface also provides access to the CA chain associated with the certificate.

The interface `ISsfPab` provides access to a list of public-key certificates belonging to others. These public-key certificates are used to verify their owners' digital signatures or to encrypt documents.

Using the Encryption API, encryption could be implemented in the following way.

```
ISsfData data;
profile = new SsfProfileKeyStore
                (keyStore, alias, null);
result = data.encrypt(profile);
```

For more information about SAP interfaces and classes for using digital signatures and encryption, please also refer to [12].

B. Property Specification

In order to check whether the security guideline about secure logging is followed using Bandera, we need to develop auxiliary source file and specify the expected

property using BSL. In this section, we describe these two steps.

```
Class String {

  public boolean isConf ;
  public boolean isEncrypted ;

  public String() {
    isConf = false ;
    isEncrypted = false ;
  }

  public void encrypt( ISsfProfile profile ) {
    isEncrypted = true ;
  }
}

class Category {

  /**
   * @observable
   *   INVOKE call(this, String m) :
   *     ( ( m.isConf == true )
   *       && ( m.isEncrypted == false ) ) ;
   */
  public void errorT( String m ) {
  }


  /**
   * @observable
   *   INVOKE call(this, String m):
   *     ( ( m.isConf == true )
   *       && ( m.isEncrypted == false ) ) ;
   */
  public void warningT( String m ) {
  }
}
```

Figure 2: Auxiliary File for Secure Logging

The auxiliary source file is depicted in Figure 2. Two auxiliary classes are defined.

Two attributes are defined in the class `String`, `isConf` means that this string is confidential, while `isEncrypted` means that this string is encrypted. In the default constructor, both `isConf` and `isEncrypted` are initialized to `false`. In the method `encrypt()`, the attribute `isEncrypted` is set to true.

In the class `Category`, we give the definitions of two logging functions as examples. For each logging function, an *invocation predicate* `call` is defined, whose format is as `INVOKE <predicate-name> <params> [: <exp>]`. An invocation predicate is true when control is at the first executable statement in the corresponding method and `<exp>` is true given the parameters `<params>`. In the case of the predicate `call`, the expression means the string to be logged is confidential but not encrypted.

With these definitions, we can specify the expected property. With the help of Bandera GUI, the property is specified as Figure 3.

In the specification, the `<quantification>` element defines universal *class instance quantification*, which means the property is satisfied by all the instances of the specified class. In this case, two quantified variables `c` and `s` are defined for class `Category` and `String`, respectively.

```
<specificationOption>
  <temporal>
    <quantification>
      <quantifiedVariable name="c" type="Category"/>
      <quantifiedVariable name="s" type="String"/>
    </quantification>
    <pattern scope="Globally" name="Absence"/>
    <predicate name="P">
      Category.errorT.call(c, s)
      || Category.warningT.call(c, s)
    </predicate>
  </temporal>
</specificationOption>
```

Figure 3: Property Specification for Secure Logging

The `<pattern>` element defines the *temporal specification pattern* used. For more details about specification patterns, please refer to [5]. In this case, *absence* pattern is used, which means the argument is never true in the execution. In the `<pattern>` element, the attribute `scope` is used to specify *pattern scope*, which is variation of basic pattern in which checking of the pattern is enabled during specified regions of execution. In this case, the pattern is held *globally* throughout the system's execution.

Finally, the predicate expression is defined. In this case, it means methods `errorT()` or `warningT()` of class `Category` is called with parameters satisfying previously specified conditions.

```
void main () {
  String secret = new String() ;

  secret.isConf = true ;

  ISsfProfile profile = new ISsfProfile() ;

  /* secret.encrypt( profile ) ; */

  Category myCat = new Category() ;

  myCat.warningT( secret );
}
```

Figure 4: Sample Program for Secure Logging

In the Bandera GUI, the property is also presented as follows:

```
forall[c: Category].
forall[s: String]
{ Category.errorT.call(c, s)
||Category.warningT.call(c, s) }
is absent globally
```

With the auxiliary file and property specification, we can check whether the expected property is held in a program like the one depicted in Figure 4 using Bandera.

This program is different from the real programs developed by programmers. The main difference is that in real programs, the class `String` does not have an attribute named `isConf`. At the same time, we do not want to bother the programmers by asking them to define a subclass of `String` to introduce the `isConf` attribute. Our solution is as follows:

- We ask the programmers to write a comment `/* confidential */` after each confidential string;
- We develop a pre-processor to add a statement like "`secret.isConf = true`" for each confidential string.

In this way, the burden of programmers is minimized, and the pre-processed program is similar to the one in Figure 4 and could be processed by Bandera.

In a program, the confidential information could be propagated by string construction using another string as parameter, string assignment and string concatenation. In order to deal with these cases, more definitions depicted in Figure 5 need to be added into the auxiliary file.

```
public String( String str ) {
    isConf = str.isConf ;
    isEncrypted = str.isEncrypted ;
}

public operator=( String str ) {
    isConf = str.isConf ;
    isEncrypted = str.isEncrypted ;
}

Public String concat( String str ) {
    isConf = isConf || str.isConf ;

    return this;
}
```

Figure 5: More Definitions for Secure Logging

In summary, when a program is developed, it is pre-processed, and given to Bandera with the auxiliary file to check whether the security guideline for secure logging is followed.

IV. Application 2: Security Guideline about Cross-Site Scripting

Cross-Site Scripting (XSS) attacks may occur when a web application accepts data originating from a user and sends it to another user's browser without first validating or encoding it. For example, suppose an attacker embeds malicious JavaScript code into his or her profile on a social network web site. If the site fails to validate such input, the JavaScript may execute malicious code in the browser of any other user who visits the profile.

A. SAP Output Encoding Framework

In SAP NetWeaver Platform, the SAP Output Encoding Framework could be used to prevent XSS attacks. This applies when application developers generate HTML codes. By encoding user supplied input before rendering it, any inserted scripts are prevented from being transmitted to users in executable form. The encoding functions implement the corresponding sanitization routines.

In order to use SAP Output Encoding Framework to prevent XSS attacks, the following four different cases need to be distinguished.

Case 1: XSS attacks can occur when string from a user is output between tags. For example, for a piece of HTML code as follows,

```
<head>
  <title>[Case 1]</title>
</head>
```

a possible XSS attack could be in the following format,

```
<head>
  <title>
    <script>alert();</script>
  </title>
</head>
```

In this case, the following functions should be applied for output encoding.

```
static String escapeToHTML
       (String input);
static String escapeToHTML
       (StringBuffer sb, String input,
        int maxLength);
static String escapeToHTML
       (String input, int maxLength);
```

Case 2: XSS attacks can occur when string from a user is output inside tags, and the output is not a URL or style. In this case, the following functions should be applied for output encoding.

```
static String escapeToAttributeValue
       (String input);
static String escapeToAttributeValue
       (StringBuffer sb, String input,
        int maxLength);
static String escapeToAttributeValue
       (String input, int maxLength);
```

```
public class SampleServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {

    // Use "request" to read incoming HTTP headers
(e.g. cookies) and HTML form data (e.g. data the user
entered and submitted)
    String input = request.getParameter("Input");

    // Use "response" to specify the HTTP response
line and headers (e.g. specifying the content type,
setting cookies).
    PrintWriter out = response.getWriter();

    // Use "out" to send content to browser

    // case 1
    out.write("<td>");
    // input = API.escapeToHTML(input);
    out.write(input);
    out.write("</td>");

    // case 4
    out.write("<script>");
    // input = API.escapeToJS(input);
    out.write(input);
    out.write("</script>");
  }
}
```

Figure 6: Sample Program for XSS

Case 3: XSS attacks can occur when string from a user is output which is a URL or style. In this case, the following functions should be applied for output encoding.

```
class HttpServletRequest {
  public HttpServletRequest() {
  }

  /**
    * @observable
    *   RETURN from_input(this, String str): ( $ret == str ) ;
    */
  public String getParameter( String field ) {
    String temp_str = new String() ;
    return temp_str ;
  }
}

class PrintWriter {
  public PrintWriter() {
  }

  /**
    * @observable
    *   INVOKE tag_begin(this, String str): ( str == "<td>" ) ;
    *   INVOKE tag_end(this, String str): ( str == "</td>" ) ;
    *   INVOKE js_begin(this, String str): ( str == "<script>" ) ;
    *   INVOKE js_end(this, String str): ( str == "</script>" ) ;
    *   INVOKE call(this, String str):
    *     ( ( str != "<td>“ ) && ( str != "</td>" ) &&
    *       ( str != "<script>" ) && ( str != "</script>" ) ) ;
    */
  public void write( String str ) {
  }
}
```

```
class API {

  /**
    * @observable
    *   INVOKE call(this, String str);
    */
  static String escapeToHTML( String str ) {
    String temp_str = new String() ;
    return temp_str ;
  }

  /**
    * @observable
    *   INVOKE call(this, String str);
    */
  static String escapeToJS( String str ) {
    String temp_str = new String() ;
    return temp_str ;
  }
}
```

Figure 7: Auxiliary File for XSS

```
static String escapeToURL
       (String input);
static String escapeToURL
       (StringBuffer sb, String input,
        int maxLength);
static String escapeToURL
       (String input, int maxLength);
```

Case 4: XSS attacks can occur when string from a user is output inside a SCRIPT context. In this case, the following functions should be applied for output encoding.

```
static String escapeToJS
(String input);
static String escapeToJS
       (StringBuffer sb, String input,
        int maxLength);
static String escapeToJS
       (String input, int maxLength);
```

For more detailed information about the usage of SAP Output Encoding Framework, please refer to [12]

B. Property Specification

In this case study, we are targeting at programs like the one depicted in Figure 6. In this program, user input is obtained from a variable `request` of class `HttpServletRequest`, while output is written to a variable `out` of class `PrintWriter`. In this case study, we will focus on Case 1 and Case 4 mentioned previously. In the program, the corresponding output encoding functions are provided as comments. For the sake of simplicity, this program is a constrained comparing with programs in reality. For example, we don't deal with output statements like:

```
out.write(“<td>” + input + “</td>”) ;
```

In order to check the sample program against expected property, we need to develop an auxiliary file as depicted in Figure 7.

In this auxiliary file, a *return predicate* `from_input` is defined for function `getParameter()` of class `HttpServletRequest`. This predicate is evaluated when the control point is immediately after any of the corresponding method's return statements, and the Bandera reserved identifier `$ret` refers to the return value of the method.

For the method `write()` of class `PrintWriter`, several *invocation predicates* are defined. Among them, `tag_begin()` means the method is invocated to output the beginning part of a tag, which is `<td>` in this paper for the sake of simplicity, `tag_end()` means the method is invocated to output the ending part of a tag. The predicates `js_begin()` and `js_end()` are for SCRIPTS context. The predicate `call()` means the method is invocated to output some other string.

For the output encoding functions `escapeToHTML()` and `escapeToJS()`, *invocation predicates* are also defined to indicate that the functions are called to encode some string.

Now, we try to specify the expected properties. In this case study, we need several properties to specify the security guideline completely.

The first property is specified as follows:

```
forall [request: HttpServletRequest].
forall [s: String].
forall [out: PrintWriter]
{ API.escapeToHTML.call(s)
    || API.escapeToJS.call(s) }
```

```
precedes
{ PrintWriter.write.call(out, s) }
after
{HttpServletRequest.getParameter.from_
input(request, s) }
```

This property uses the *precedence* pattern, which informally means that a designated state/event always occurs before the first occurrence of another designated state/event. The scope of the property is *after*, which means the checking is enabled after the first occurrence of a state/event. Thus in summary, this property informally means that before a string is output, it should be encoded using SAP output encoding functions, and this check is only performed if this string is obtained from user input.

The second property is specified as follows:

```
forall[tag_begin: String].
forall[s: String].
forall[tag_end: String].
forall[out: PrintWriter].
forall[js_begin: String]
{PrintWriter.write.call(out, s)
 && PrintWriter.write.tag_end
                    (out, tag_end) }
responds to
{API.escapeToHTML.call(s)
 && PrintWriter.write.tag_begin
                   (out, tag_begin) }
before
{PrintWriter.write.js_begin
                   (out, js_begin) }
```

This property uses the *response* pattern, which informally means that the occurrence of a designated state/event is followed by another designated state/event in the execution. The scope of the property is *before*, which means the checking is enabled before the first occurrence of a state/event. Thus in summary, this property informally means that the output of the beginning part of a tag and the output encoding using `escapeToHTML()` are followed by the outputs of the encoded string and the ending part of a tag before outputting the beginning part of a SCRIPT.

The third property is specified for SCRIPT context similar to the second one as follows:

```
forall[js_end: String].
forall[s: String].
forall[js_begin: String].
forall[out: PrintWriter].
forall[tag_begin: String]
{PrintWriter.write.call(out, s)
 && PrintWriter.write.js_end
                    (out, js_end) }
responds to
{API.escapeToJS.call(s)
 && PrintWriter.write.js_begin
                   (out, js_begin) }
before
{PrintWriter.write.tag_begin
                   (out, tag_begin) }
```

When all the properties are held, the application of the security programming guideline is guaranteed. In summary, in this case study, with the auxiliary file and expected property specifications, we are able to check whether the security guideline for XSS are followed or not without additional effort from programmers.

V. Related Work

Java PathFinder (JPF) [15] is another source code model checking tool, which has been applied in several different applications [1] [9] [10]. By default, it checks the following "properties": no deadlock, no assertion violation, and no uncaught exceptions.

In order to specify more complex properties, such as what we need in order to enforce security programming guidelines, there are three ways in JPF:

- Use Java assertion inside the application under analysis. The drawback is that the assertions will widely spread all around the program. Thus, writing the assertions can only be programmers' responsibility. This is not realistic.
- The second way to specify properties is by using `gov.nasa.jpf.Property` or `gov.nasa.jpf.GenericProperty` instances to encapsulate property checks. The user typically creates an instance of such a class and provides an implementation for its `check()` method which does the main work for checking the property. The `check()` method is evaluated after each transition. If it returns false and termination has been requested, the search process ends, and all violated properties are printed (which potentially includes error traces). The advantage is that the security property checking part is just one class, which could be provided by security experts. The disadvantage is that the security experts need to work in the programming level to deal with all the programming details, rather than to work with temporal logic only.
- The third way of specifying properties is through the use of two listener classes: `gov.nasa.jpf.SearchListener` and `gov.nasa.jpf.VMListener`. The listeners can subscribe to events during the search, making JPF easily extensible. They can be used to implement more complex checks that require more information than what is available after a transition is executed. The advantage and disadvantage are the same as the previous option.

Although JPF previously supported LTL (Linear Temporal Logic) checking, this feature is no longer supported. Based on these observations, we use Bandera rather than JPF.

There are also some other source code model checkers which could be mentioned. Scott Stoller [13] has developed a stateless checker for multi-threaded distributed Java programs. The basic technology used is an extension of Godefroid's Verisoft approach [6]. In [10], a tool is

developed that translates Java into SAL (Symbolic Analysis Laboratory), an intermediate language designed to interface with several model-checking and theorem-proving tools. Eran Yahav has developed a tool for checking safety properties of Java programs [17] built on top of three-valued logic analysis tool TVLA.

There are much works on performing source code analysis to identify vulnerabilities related to XSS attack. Among these works, [14] and [16] are relatively recent ones.

VI. CONCLUSION

In this paper we present how to use Bandera Specification Language to describe security programming guidelines as temporal logic properties by two case studies. This effort is the first step of using Java source code model checker Bandera to check whether security programming guidelines are followed and consequently whether the corresponding security vulnerabilities exist.

In addition to the benefits brought by model checking, the additional effort of programmer is little in our approach. In both case studies, all the auxiliary files and property specifications are specified by security and formalism experts and could be applied across projects in the development organization. For each specific secure programming guideline, the security and formalism experts need to specify specific temporal logic properties accordingly. The scalability of this approach consists in that once the expected properties are specified, they could be used to check programs across the whole development organization without any additional effort of developers.

Currently, the properties in the two case studies are specified in Bandera version 0.3. Since Bandera 0.3 is a prototype, and our properties are more complex than the ones used to present the idea of Bandera, these properties have not been used to check real programs. Performing model checking itself and collecting related performance data are the next steps.

As a future work, we want to remove some constrains used in this paper to improve the applicability of the techniques developed in this paper, and to extend the techniques to check other security programming guidelines.

The temporal property checking is supported in Bandera version 0.3. In Bandera 1.0, there is an architectural change, and temporal logic property checking is removed. Another direction of our future work is to see whether we can transport what we have done in Bandera 0.3 to some other Java source code model checkers.

REFERENCES

[1] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, R. Washington and W. Visser, "Experimental Evaluation of Verification and Validation Tools on Martian Rover Software," Formal Methods in Systems Design Journal, Volume 25, Number 2-3, September 2004

[2] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new model checker," International Journals on Software Tools for Technology Transfer, Springer, Vol. 2, No. 4, March 2000, pp. 410-425.

[3] J. Corbett, M. Dwyer, J. Hatcliff, and Robby, "Expressing checkable properties of dynamic systems: the Bandera specification language," International Journals on Software Tools for Technology Transfer, Springer, Vol. 4, No. 1, October 2002, pp. 1433-2779.

[4] C. Demartini, R. Iosif, and R. Sisto, "dSPIN: a dynamic extension of SPIN," Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking, LNCS, Vol. 1680, Springer-Verlag, 1999, pp. 261–276.

[5] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," Proceedings of the 21st International Conference on Software Engineering, May 1999.

[6] P. Godefroid, "Model-checking for Programming Languages using Verisoft," POPL'97, January 1997, pp. 174-186.

[7] J. Hatcliff and M. Dwyer, "Using the Bandera tool set to model-check properties of concurrent Java software," Proceedings of the 12th International Conference on Concurrency Theory, LNCS, Vol. 2154, Springer-Verlag, 2001, pp. 39–58.

[8] G. Holzmann, "The model checker SPIN," IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-294.

[9] G. Lindstrom, P. Mehlitz and W. Visser, "Model Checking Real Time Java Using JavaPathfinder," Proceedings of the Third International Symposium on Automated Technology for Verification and Analysis (ATVA), October 2005

[10] D. Park, U. Stern, J. Skakkebaek, and D. Dill, "Java Model Checking," Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, June 2000.

[11] J. Penix, W. Visser. S. Park, C. Pasareanu, E. Engstrom, A. Larson and N. Weininger, "Verifying Time Partitioning in the DEOS Scheduling Kernel," Formal Methods in Systems Design Journal, Volume 26, Number 2, March 2005.

[12] Sap netweaver 7.0 knowledge center, URL: http://help.sap.com/content/documentation/netweaver/docu_nw_70_design.htm.

[13] S. Stoller, "Model-checking multi-threaded distributed Java programs," Proceedings of Seventh International SPIN Workshop, LNCS 1885, Springer-Verlag, 2000, pp. 224-244.

[14] O. Tripp, M. Pistoia, S. Fink, S. Sridharan, and O. Weisman, "TAJ, Effective Taint Analysis for Web Applications," PLDI'09, ACM SIGPLAN Notes, Vol. 44, No. 6, June 2009, pp. 87-97.

[15] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," Automated Software Engineering, Vol. 10, No. 2, Kluwer Academic Publishers, April 2003, pp. 203-232.

[16] G. Wassermann and Z. Su, "Static Detection of Cross-Site Scripting Vulnerabilities," ICSE 2008, May 2008, pp. 171-180.

[17] E. Yahav, "Verifying Safety Properties of Concurrent Java Programs using 3-valued Logic," POPL'01, January 2001, pp. 27-40.