

Provable Protection against Web Application Vulnerabilities Related to Session Data Dependencies

Lieven Desmet, Pierre Verbaeten, Wouter Joosen, and Frank Piessens

Abstract—Web applications are widely adopted and their correct functioning is mission-critical for many businesses. At the same time, web applications tend to be error-prone and implementation vulnerabilities are readily and commonly exploited by attackers. The design of countermeasures that detect or prevent such vulnerabilities, or protect against their exploitation is an important research challenge for the fields of software engineering and security engineering.

In this paper, we focus on one specific type of implementation vulnerability, namely broken dependencies on session data. This vulnerability can lead to a variety of erroneous behavior at run time and can easily be triggered by a malicious user by applying attack techniques such as forceful browsing. This paper shows how to guarantee the absence of run-time errors due to broken dependencies on session data in web applications. The proposed solution combines development-time program annotation, static verification and run-time checking to provably protect against broken data dependencies.

We have developed a prototype implementation of our approach building on the JML annotation language and the existing static verification tool ESC/Java2, and we successfully applied our approach to a representative J2EE based e-commerce application. We show that the annotation overhead is very small, that the performance of the fully automatic static verification is acceptable, and that the performance overhead of the run-time checking is limited.

Index Terms—Software/Program Verification, Security, Reliability, Data sharing, Web-based services, Web technologies.

I. INTRODUCTION

WEB applications are widely adopted in nowadays life. More and more individuals and organizations strongly depend on their correct functioning, resulting in an increasing demand for reliability and security [1]. For instance, many companies already incorporate e-commerce in their business model to increase their revenues. At the same time, web applications tend to be error-prone, and are a welcome target for attackers due to their high accessibility and possible profit gain.

NIST's national vulnerability database clearly shows an increasing number of vulnerabilities located in the application layer. A similar trend stands out in the Web Hacking Incidents Database (WHID). Design flaws and implementation bugs are two important root causes for many vulnerabilities in web applications. They potentially lead to erroneous behavior at run time and undermine the overall reliability and security of a web application. This is

especially the case in web applications, since attackers can more easily trigger specific implementation bugs because of the open and reactive nature of web applications.

This paper focuses on one particular type of implementation bugs, namely run-time errors due to broken data dependencies in data-centered web applications. In a data-centered application, the different components of the application can indirectly share data through a shared data repository without actually interacting with each other. This loose coupling adds extra flexibility to the software development and composition process, and is often used in software engineering as a viable trade-off between data encapsulation and efficient data sharing. For instance, web applications typically use indirect data sharing to store and retrieve the non-persistent, server-side session state for each user.

On the downside, data dependencies in data-centered software compositions can easily break. A data-centered application is correctly composed if, at run time, each component is able to retrieve the data from the repository that it expects to find. Infringements typically manifest themselves in unexpected exceptions. Thus, the correct functioning of a component depends on the run-time state of the shared repository during its execution. Since these dependencies typically are hidden within a software system, it is hard for a software composer to build web applications without breaking any of the hidden dependencies between the components and the shared data repository. This is a relevant composition problem and typically leads to run-time errors. These run-time errors can for instance be exploited in web applications by applying forceful browsing. The impact of the run-time errors depends on the particular application, but possible consequences include the execution of unexpected application logic, information leakage due to bad error handling, broken data integrity (e.g., storing null strings to the database back-end), skipping of clean-up code (such as the code that closes database connections) which in turn may lead to a Denial-of-Service, and many more. Moreover, the loose coupling between components in indirect data sharing circumvents several consistency checks of nowadays compilers, resulting in late detection of composition problems (i.e. at run time instead of at compile time).

The main contribution of this paper is that it shows how to formally guarantee that no data dependencies are broken in a given web application. To do so, the presented approach only requires a limited annotation of the given application, and it applies a combination of static and dynamic verification. The guaranteed absence of run-time errors due to broken data dependencies results in a more reliable and secure web application. In addition, a prototype implementation of the presented approach is built based on an existing verifier for Java and the prototype implementation

Manuscript received February 23, 2007; revised July 10, 2007.

L. Desmet, P. Verbaeten, W. Joosen, and F. Piessens are with the DistriNet Research Group of the Department of Computer Science at the Katholieke Universiteit Leuven (Belgium)

e-commerce application.

The rest of this paper is structured as follows. Section II provides some background information on indirect data sharing in web applications and some common composition problems due to broken data dependencies. This concept is illustrated in more detail in the *Duke's BookStore* case study, a representative e-commerce web application (section III). Next, section IV defines the requirements for a solution to counter these composition problems in a transparent and developer-friendly way. In section V, we present our solution to detect and prevent composition problems in data-centered, reactive web applications. In section VI, the design and implementation of our prototype is demonstrated and illustrated with the *Duke's BookStore* e-commerce application. In addition, the results are presented in section VII. Next, section VIII discusses the chosen trade-offs between the usability and the accuracy of the verification. In section IX, the presented work is related to existing research in program verification and web security and, finally, section X summarizes the contributions of this paper.

II. PROBLEM STATEMENT

This section provides more detailed background information on indirect data sharing in web applications and some common composition problems. After a short, general introduction to indirect data sharing in subsection II-A, the typical use of a shared repository in web applications is described in subsection II-B. In addition, the erroneous behavior due to broken dependencies is shortly discussed, as well as the adequacy of existing security countermeasures to prevent malicious users to exploit these kind of bugs.

A. Indirect Data Sharing

In the repository architectural style [2], a system consists of a central data structure (representing the state of the system) and a set of separate components interacting with the central data store. This architectural style is quite common in software engineering as a viable trade-off between data encapsulation and efficient data sharing. For instance, indirect data sharing is adopted in several component models and APIs such as JavaServlet containers [3], Pluggable Authentication Modules framework (PAM) [4] and JavaSpaces in Sun's Jini [5].

The application is correctly composed with respect to the indirect data sharing if, at run time, each component is able to retrieve the data from the repository that it expects to find. Thus, the correct functioning of a component depends on the run-time state of the shared repository during its execution. Or rephrased, in applications with a shared data repository, implicit semantical dependencies exist between components that share a common data item on the shared repository. In Fig. 1, the implicit semantical dependencies within a data-centered application are explicitly shown, while the actual component interactions (i.e. the control flow) are abstracted.

B. Session Sharing in Web Applications

Web applications are server-side applications that are invoked by thin web clients (browsers), typically using the HyperText Transport Protocol (HTTP). A user can navigate through a web application by clicking links or URLs in his browser, and he is also able to supply input parameters by completing web forms.

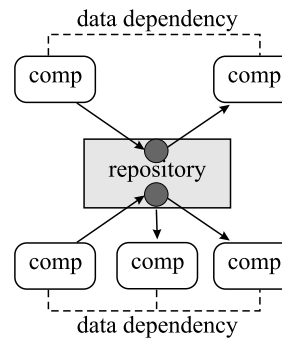


Fig. 1. Data dependencies in data-centered applications

HTTP is a stateless, application-level request/response protocol and has been in use on the World Wide Web since 1990 [6]. Since the protocol is stateless, each request is processed independently, without any knowledge of previous requests. To enable the typical user's session concept in web applications, the web application needs to add session management on top of the stateless HTTP layer. Different techniques exist to embed web requests within a user session such as the use of cookies, URL rewriting or hidden form fields [7].

Nowadays, most web applications use an underlying framework or web technology to facilitate the development and the deployment of the web application. Widespread technologies such as PHP, ASP.NET and JSP/Servlets support among others the management of user sessions. Next to tracking to which user session a web request belongs, these technologies also provide server-side state for each user session. While processing a web request, server-side web components can store non-persistent, user-specific data (e.g. a shopping cart in an e-commerce-site) in a shared data repository bound to the user session. Other web components can then retrieve this data while processing future requests in the same user session.

Breaking data dependencies is a common risk in composing data-centered applications, and likewise data-centered web compositions are vulnerable to broken data dependencies without additional support. Based on extensive experience in several data-centered applications, we identified two common types of composition mismatch: cases in which a data item is not available on the shared repository although a reading component expects it on the repository during execution, and cases in which the type of an available data item does not correspond with the type expected by the reading component. Since the loose coupling in indirect data sharing circumvents several consistency checks of nowadays compilers, these composition mismatches typically lead to run-time errors (e.g. *NullPointerException* and *ClassCastException* in Java-like languages). This is a relevant composition problem, and although the impact of such a run-time error depends on the particular web application, broken data dependencies undermine the reliability and security of the web application. Possible consequences of such run-time errors include the execution of unexpected application logic, information leakage and Denial-of-Service.

Existing security solutions do not provide adequate support to protect web applications against such implementation-specific bugs. Network security fails to effectively protect web applications against attackers [8]. Network firewalls such as stateful packet filters typically operate on the network or transport layer

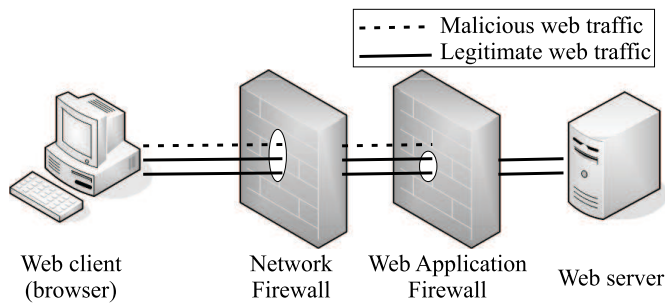


Fig. 2. Web Application Firewall infrastructure

(e.g. granting access to a complete web application by allowing TCP port 80 traffic), whereas web applications are typically attacked on the application layer. To counter web application vulnerabilities, Web Application Firewalls (WAFs) operate on the application layer and analyze web requests between a browser and the web server [9]. Often, WAFs are placed inline between the browser and server (as displayed in Fig. 2), and enforce real-time access control, based on application-level information such as the requested URL, the supplied credentials and input parameters and the user session's history.

WAFs are applied to mitigate a range of vulnerabilities, including vulnerabilities to forceful browsing. Forceful browsing is the act of directly accessing web pages (URLs) without consideration for their context within an application session [10]. Bypassing intended application flow in a web application can generally lead to unauthorized access to resources or unexpected application behavior [11]. Moreover, a malicious user will typically apply forceful browsing to exploit implementation-specific broken data dependencies in data-centered web applications in a more or less controlled way. Therefore, WAFs that counter forceful browsing attacks are an important countermeasure to prevent the exploitation of broken data dependencies. According to the "Web Application Firewall Evaluation Criteria" [12], such a WAF implements the *strict request flow enforcement* criterion. This criterion refers to the technique where a WAF monitors individual user sessions and keeps track of the links already followed and of the links that can be followed at any given time [12].

An important drawback of WAFs is their limited coverage of protecting implementation-specific vulnerabilities. A WAF uses either a positive or negative security model as basis for access decisions, and is configured manually by the administrator or automatically by observing legitimate or malicious network traffic. Thus, WAFs are typically configured without a strong binding to the implementation of the web application they protect, and because of this, there is no strong guarantee that a configured WAF actually mitigates all implementation-specific bugs of a given web application. Therefore, additional support is needed to achieve stronger security guarantees in data-centered web application.

III. CASE STUDY: THE DUKE'S BOOKSTORE APPLICATION

In this section, the *Duke's BookStore* application illustrates in more detail the typical use of shared data repositories for sharing session state in a servlet-based web application. In addition, the case study will be used in sections V to VII to illustrate and validate our solution.

The Java Servlet technology is part of the J2EE specification and provides mechanisms for extending the functionality of a Web server and for accessing existing business system [3]. Java Servlets are functional units of the web tier. A J2EE web application is typically a collection of Java Servlets and is deployed in a servlet-based web container such as Tomcat, JBoss or WebSphere. Servlets can indirectly share data by means of a shared data repository. In fact, five instances of shared repositories are provided to the servlet, each with a different access scope: a data repository associated with the dynamic web page (1), with the web request (2), with the user session (3), the web context (4) and the application (5). We limit the illustration of data sharing by only discussing sessions. Verification of broken data dependencies on the request level has been investigated in [13].

The *Duke's BookStore* web application is an exemplary Java Servlet application that is bundled together with the J2EE 1.4 Tutorial [14]. This relatively small e-commerce application consists of about 3500 lines of code, and implements the basic functionality of a web shop by using Java Servlets. The core application logic is supplied by 6 servlets and 1 filter:

a) *BookStoreServlet*: The *BookStore* servlet returns the main web page for the Duke's Bookstore. From this start page, links are provided to browse the book catalog, or jump to the book details of a particular book (e.g. a book in promotion).

b) *BookDetailsServlet*: The *BookDetail* servlet returns information about any book that is available from the bookstore. Links are provided to the user to either add the book to the shopping cart, or to look further into the book catalog.

c) *CashierServlet*: The *Cashier* servlet asks for the user's name and credit card number so that the user can buy the books in his shopping cart. Payment information is sent to the *Receipt* servlet.

d) *CatalogServlet*: The *Catalog* servlet displays the book catalog, and provides the possibility to add books to the user's shopping cart or to buy the books in the shopping cart by redirecting to the cashier servlet.

e) *ReceiptServlet*: The *Receipt* servlet processes the order by updating the book database inventory. Next, the servlet invalidates the user session.

f) *ShowCartServlet*: The *ShowCart* servlet returns information about the books in the user's shopping cart.

g) *OrderFilter*: The *Order* filter provides server-side logging of shopping orders, whenever the *ReceiptServlet* is called.

The components share three data items on the shared session repository: messages, cart and currency (table I). The data item *messages* is a *ResourceBundle* and contains locale-specific objects. This data item is important for the internationalization of the application and can be used by all components to display messages in the user's preferred locale. Similarly, the data item *currency* of the type *Currency* contains the user's preferred currency. In addition, the data item *cart* of type *ShoppingCart* represents the shopping cart of the user in the e-commerce application.

The components interact with the shared session repository as listed in table I. The interactions are specified by a type (e.g. *ResourceBundle*), a string identifier (e.g. messages) and the type of interaction. The interaction types used are displayed in table I. We have chosen to use more fine-grained interactions than just simple read and write operations to increase the accuracy. The type *def. read/write* stands for a defensive read/write operation as

TABLE I
INTERACTIONS WITH THE SHARED SESSION REPOSITORY IN THE BOOKSTORE APPLICATION

	Shared data items		
	ResourceBundle messages	Currency currency	ShoppingCart cart
BookDetailsServlet	read	cond. def. read/write	
BookStoreServlet	def. read/write		
ReceiptServlet	read		def. read/write
CashierServlet	read	def. read/write	def. read/write
CatalogServlet	read	def. read/write	def. read/write
ShowCartServlet	read	cond. def. read/write	def. read/write
OrderFilter		read	read

shown in Fig. 3, i.e. the application can handle a null pointer as result of the read operation, and in that case the servlet stores a not null object of the expected type to the shared session repository. The label *cond.* means that the operation possibly occurs, depending on an unspecified condition (e.g. the run-time state of the book database inventory).

```

Currency c = (Currency) session . getAttribute ("currency");
if (c == null) {
    c = new Currency();
    session . setAttribute ("currency", c);
}

```

Fig. 3. Example of a defensive read/write operation in BookDetailsServlet

Session repository interactions are typically not specified in a Servlet-based application and neither they are in this J2EE application. Thus, the implicit assumptions of the developer on how a servlet or filter should be used with respect to its interactions with the shared session repository are neither articulated or available in the source code. This makes correct deployment or software evolution very hard without reanalyzing the complete source code.

Even in small e-commerce applications such as Duke's BookStore, the interactions with the shared session repository impose restrictions on the allowed client-server interaction protocol. If for example a user session starts with any URL path other than the /bookstore starting point of the application (which is a typical forceful browsing attack), the execution of any servlet ends up with a *NullPointerException*: every servlet retrieves the messages data item from the shared repository and assumes in its execution that the retrieved *ResourceBundle* is not null. Another *NullPointerException* occurs in the Duke's BookStore application if the *OrderFilter* (according to the deployment information of this application applied to the *ReceiptServlet*) is called in a user's session before the cart and currency data items are stored to the shared repository. Also notice the impact of software evolution on indirect sharing. For example, the latter problem does only occur if the *OrderFilter* is added to the application.

The impact of a *NullPointerException* during execution depends on the particular application. Possible consequences include the execution of unexpected application logic, information leakage due to bad error handling, broken data integrity by storing null strings to the database back-end, skipping of clean-up code (such as the code that closes database connections) which in turn may lead to a Denial-of-Service, and many more. In the remainder of this paper we assume that the occurrence of a

NullPointerException due to data repository interactions in a web application negatively affects the security of the application and thus should be prevented from happening.

Similarly, incompatibilities between the type of the data item on the repository and the type expected by a retrieving component result in *ClassCastExceptions* at run time. In Java-like data-centered web application, existing static type checking at compile time is merely circumvented, since retrievals from the repository are done under the *Object* type, and the retrieved object is then downcasted to the expected type at run time.

Finally, the complexity of shared data dependencies in real-life applications may not be underestimated. We have documented the complexity of the GatorMail webmail application in [15]. In the in-depth dependency analysis of this medium-sized software system, already more than 1350 interactions with the shared data repository were identified without any form of documentation.

IV. REQUIREMENTS

The high-level goal of this research is to increase the reliability and security of data-centered web applications by reducing run-time errors caused by broken data dependencies. We define the following desired composition property for indirect data sharing in data-centered web applications.

No broken data dependencies:

No client request causes a data item to be read from the server-side, shared session repository before it actually has been written. For each shared data read interaction, the shared data item that already has been written to the shared session repository is of the type that is expected by the read operation.

In particular, this paper eliminates certain types of run-time errors (such as a *NullPointerException* or a *ClassCastException*) by giving a formal guarantee that the *no broken data dependencies* property is not violated in a given composition.

The composition property defines the correctness of the indirect data sharing in term of what is expected by a component whenever it tries to read a data item from the shared repository. In addition, it is also important to investigate the control flow of an application, since the control flow determines which components are executed in which order, and hence also what shared data interactions occur in what order.

Reducing certain types of run-time errors by formally verifying that a given composition does not violate the desired composition property certainly improves the reliability of the software composition, but in order to be really useful, the following interoperability and usability criteria are important as well:

1) *Interoperability*: It is important that the proposed solution is interoperable with the existing web infrastructure and does not interfere with other web security solutions. Optimally, the solution can be added to the infrastructure in a transparent way and cooperates with other security countermeasures, if needed.

2) *Usability*: In order to encourage wide adoption by developers, we also identified two important usability characteristics for the solution: limited overhead and applicability to real-life applications.

a) *Limited overhead*: In order to be generally applicable, the introduced overhead for the software developer and software composer must be minimal, both in terms of additional workload and verification time. In addition, the solution must be easy to understand for mainstream developers and software composers. The less overhead and complexity the solution introduces, the more likely that the proposed solution will actually be adopted.

b) *Applicability to real-life applications*: The applicability of the proposed solution may not be limited to toy examples, but the proposed solution must also be more generally applicable to larger, real-life applications. This includes among others that the proposed solution is scalable to larger software projects and that the solution is not limited to a specific choice of program language or software framework. Instead a rather technology neutral solution is preferred, which then can be easily adopted in different software platforms.

V. OVERVIEW OF THE SOLUTION

In this section we propose our solution: we specify a component's interactions with the shared session repository and use static and dynamic verification to guarantee that no client-server interaction leads to violation of the *no broken data dependencies* property. Fig. 4 depicts an overview of our solution. At the left side of the figure the different artifacts of our application are listed: next to the implementation, the deployment information and the run-time web traffic are also used as input for our verification process.

The verification process consists of three steps. Firstly, the interactions with the shared session repository are explicitly specified in component contracts, and static verification is used to verify that each component implementation obeys its contract specification. Secondly, the *no broken data dependencies* property is verified in each possible execution path within a user's session. To verify this property statically, an upper bound is defined for the client-server interactions, namely the *intended client-server protocol*. Next, the property is verified under the assumption that

the client-server interactions are prefixes of the *intended client-server protocol*. Finally, run-time policy enforcement is used to guarantee that only web requests that are prefixes of the *intended client-server protocol* are processed by the web application. By combining these three verification steps, our solution ensures the *no broken data dependencies* property in a given application.

We will now discuss each of the three steps in more detail in the following subsections.

A. Server-side Specification and Verification

In order to specify a component's interactions with the shared session repository, each web component is extended with an appropriate component contract. The contract is expressed in a problem-specific contract language, which is easy to understand for application developers. The grammar of the proposed problem-specific contract language is shown in Fig. 5.

The problem-specific contract language expresses the interactions with the repository in terms of *read*, *write* and *possibly write* statements:

a) *The read statement*: lists the component's expectations about the repository state. To do so, the contract specifies the expected type for each relevant data item, and uses the label *Nullable* to indicate that the component can handle a null reference for that particular data item. The latter one reflects the defensive reads as shown in Fig. 3.

b) *The write statement*: expresses the data items on the shared session repository that will be altered into a non-null instance of the specified type by executing the component.

c) *The possibly write statement*: lists the data items on the shared session repository that may be altered by executing the component. This statement defines that for each of the data items, the write interactions results in a non-null instance of the specified type, or that the particular data item is not altered at all, depending on unspecified conditions.

Notice the explicit distinction between write and possibly write statements. Write statements are subsumed in possibly write interactions, but are semantically richer in describing the state of the shared data repository. A write statement clearly describes the state of the data item after executing the component (i.e. that the data item will be altered to a non-null instance of the given type), whereas a possible write statement gives less information about the state of the data item. Moreover, every update of the shared session repository during the execution of the component is either covered by the write statement or the possibly write statement.

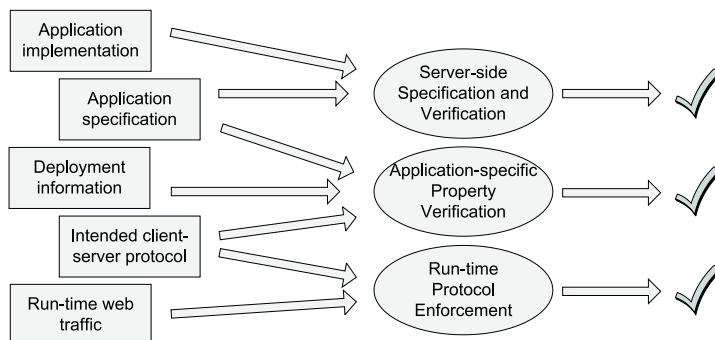


Fig. 4. Solution overview

```

CONTRACT := SPECLINE*
SPECLINE := SPECTAG ( READS | WRITES | CONDITIONALWRITES)
SPECTAG := '//spec: '
READS := 'reads ' READOBJECTSET ' from session;'
READOBJECTSET := '{' (READOBJECT,)* READOBJECT '}'
READOBJECT := ( 'Nullable<' TYPE '>' NAME | TYPE NAME )
WRITES := 'writes ' WRITEOBJECTSET ' on session;'
WRITEOBJECTSET := '{' (WRITEOBJECT,)* WRITEOBJECT '}'
WRITEOBJECT := TYPE NAME
CONDITIONALWRITES := 'possibly' WRITES
TYPE := IDENTIFIER
NAME := IDENTIFIER

```

Fig. 5. EBNF notation of the problem-specific contract language

```

1 //spec: reads {ResourceBundle messages, Nullable<ShoppingCart> cart, Nullable<Currency> currency} from session;
2 //spec: writes {ShoppingCart cart} on session;
3 //spec: possibly writes {Currency currency} on session;

```

Fig. 6. Problem-specific specification of ShowCartServlet

For instance, Fig. 6, shows such a problem-specific contract of the *ShowCartServlet*, which is a straightforward mapping of the following shared data interactions.

ShowCartServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)
Currency currency (cond. def. read/write)

The *read* interaction for the messages data item is translated in a read statement in the problem-specific contract (data item *messages* in line 1 of Fig. 6). The *def. read/write* interaction is translated in a combination of a *Nullable*-labeled read statement and a write statement (data item *cart* in line 1 and 2 of Fig. 6). Similarly, the *cond. def. read/write* interaction is translated in a combination of a *Nullable*-labeled read statement and a possibly write statement (data item *currency* in line 1 and 3 of Fig. 6).

The formal semantics of these contracts are given by a translation into the Java Modeling Language (JML) [16]. We briefly discuss this translation in subsection VI-A.

Given the component contracts for each component, static verification is used to verify that a component's implementation obeys its contract, i.e. that only the read and write interactions happen that are specified in the contract.

Notice that in case of a mismatch between the provided contract and the component implementation, this first static verification step will fail to verify the compliance and as a result the overall verification process will fail.

B. Application-specific Property Verification

The *no broken data dependencies* property is verified by checking all possible execution paths in a user's session. To verify the property statically, an upper bound is defined for the client-server interactions, namely the *intended client-server protocol*. This is an upper bound for the non-deterministic interactions between client and server, and includes all valid client-server interactions that may occur in the application under normal circumstances.

The intended client-server protocol can be expressed in various ways such as a regular expression, an EBNF notation or a labeled state transition system. For example, Figs. 7 and 8 are two

different representations of the intended protocol for the Duke's BookStore application. Note that in web applications the protocol can be interrupted at any time, e.g. if a web user stops surfing to the given web application or closes the browser. This is indicated by *nil* in the EBNF notation, and with dashed lines in the labeled state machine.

In order to statically verify that any prefix of the *intended client-server protocol* does not violate the desired application property, the intended protocol is verified in combination with the component contracts and the given deployment information. In a J2EE web application for example, the web deployment descriptor contains among others the mapping between URLs and servlets, as well the servlets on which filters are applied.

To verify the desired application property, it is important that component contracts precisely describe the interactions with the shared repository. In case of too generic contracts (e.g. if a programmer does not take time to adequately identify the interactions or tries to shortcut the annotation process by annotating every interaction as *possibly write*), the second verification step will not succeed for certain legitimate client-server protocols, and as a result the overall verification process will fail.

C. Run-time Protocol Enforcement

Finally, since the *no broken data dependencies* property is verified under the assumption that all web requests obey the *intended client-server protocol*, this assumption needs to be enforced at run-time. This can be done by loading the protocol specification into a supporting WAF or extending the application with an appropriate filter. As a result, only prefixes of the *intended client-server protocol* are allowed to be processed by the web application.

VI. DESIGN AND PROTOTYPE

In this section, we describe the design and implementation of our prototype and discuss how it can be used to secure the Duke's BookStore web application.

```

PROTOCOL := /bookstore + SERVLET_A * + RECEIPT
RECEIPT := ( SERVLET_B + SERVLET * + orderfilter + /bookreceipt ) | nil
SERVLET := SERVLET_A | SERVLET_B
SERVLET_A := /bookstore | /bookdetails | /bookshowcart | /banner | nil
SERVLET_B := /bookcatalog | /bookcashier

```

Fig. 7. EBNF notation of the client-server protocol

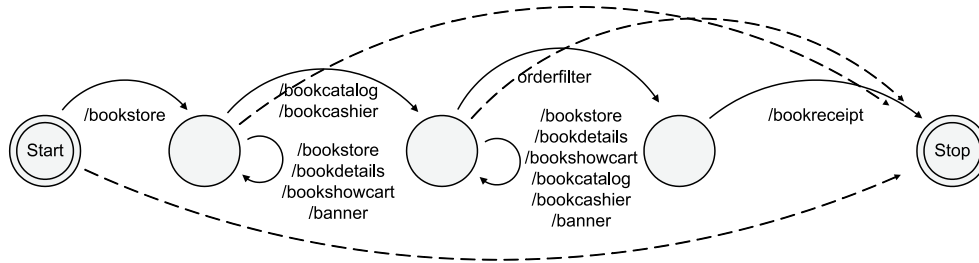


Fig. 8. Client/server interaction protocol

```

1 package servlets ;
2
3 public class ShowCartServlet extends HttpServlet {
4     //JML contract:
5     // @ also
6     // @ requires request != null;
7     // @ requires response != null;
8     // @ requires request.session != null;
9     // @ requires request.session.currency instanceof Currency || request.session.currency == null;
10    // @ requires request.session.messages instanceof ResourceBundle;
11    // @ requires request.session.cart instanceof ShoppingCart || request.session.cart == null;
12    // @ ensures request.session.cart instanceof ShoppingCart;
13    // @ ensures request.session.currency instanceof Currency || \old(request.session.currency) == request.session.currency;
14    // @ modifies request.session.cart;
15    // @ modifies request.session.currency;
16    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException;
17 }

```

Fig. 9. Contract for shared session repository interactions (ShowCartServlet.spec)

A. Server-side Specification and Verification

In order to use existing verifiers to check if the implementation of a component adheres to its contract, the problem-specific contracts are translated into the Java Modeling Language (JML) [16]. For instance, the JML contract in Fig. 9 expresses the interactions between the ShowCartServlet and the shared session repository in terms of pre- and post-state of the repository.

In our prototype, the problem-specific component contracts are translated automatically in JML contracts, and the translation tool can be downloaded from the paper’s accompanying website [17]. The rationale for the translation is as follows:

a) Read statements: Read statements in the problem-specific component contracts are translated into the precondition that the given data item is of the expected type. For instance, line 10 of Fig. 9 specifies that the method requires that the data item *messages* on the shared session repository is a non-null instance of *ResourceBundle*.

In case the read statement is labeled with *Nullable*, the requirement on the data item is relaxed to either be a non-null instance of the expected type or to be the null reference (e.g. line 9 of Fig. 9).

b) Write statements: Write statements in the problem-specific component contracts are translated in an *ensures* clause specifying that after execution the given data item is of the expected type. Line 12 of Fig. 9 specifies for example that the method ensures that after execution the data item *cart* on the shared session repository will be a non-null *ShoppingCart*. In addition, the JML contract also explicitly expresses the frame condition of the method, i.e. what part of the session state a method is allowed to modify. This is done by adding a *modifies* clause for each written data item (e.g. data item *cart* in line 14 of Fig. 9).

c) Possibly write statements: Possibly write statements in the problem-specific component contracts are translated in an *ensures* clause specifying that after execution the given data item is of the expected type or that the given data item remains unchanged while executing the method (line 13 of Fig. 9). In addition, a frame condition is added for each data item that is possibly written by executing the method. For instance, line 15 of Fig. 9 expresses that the method may alter the shared data item *currency*.

Finally, notice the use of the *also* keyword in the JML contracts. The *ShowCartServlet* extends the *HttpServlet*, and by doing so it

inherits the public method specification of the *doGet* method. To refine the specification of an overridden method (e.g. by weakening preconditions or by strengthening postconditions), the specification in JML starts with the *also* keyword, which combines the specifications of the supertype and the subtype. Similarly, the *also* keyword can also be used in regular specification to combine different specification blocks into a nested specification. More information about the desugaring of *also combinations* in JML can be found in [18].

Since the *doGet* method of the *HttpServlet* does not provide common behavior for all the inheriting servlets, the supertype method is annotated with the strongest possible precondition, i.e. the *requires false* pragma. In this way, all inheriting servlets are able to weaken this precondition conform the Liskov substitution principle. Note that this design decision prohibits polymorphic use. This was not an issue in the prototype implementation, since we did not encounter polymorphic use of the *HttpServlet* objects in the different servlet implementations or the application-specific check method.

In the remainder of the paper we have chosen to show the translated JML contracts since JML is a fairly well-known contract language. One of the main advantages of JML is the large amount of tool support that is available [19]. Tools are available for run-time contract checking, test generation, static verification and inference of specifications. A variety of static verification tools is available that make different trade-offs in verification power and need for user interaction.

In our prototype, we have chosen to use the ESC/Java2 verifier [20]. The main advantage of this verifier is that it requires no user interaction. On the downside, the verifier is far from complete and has some known sources of unsoundness [21]. In section VIII, we will discuss how this impacts on our prototype.

To check the compliance of the component implementation with ESC/Java2, the specification of the shared repository is generated (Fig. 10). Hereby, *explicit JML pragmas* provide a mapping between a *ghost field* and the state of a specific data item in the hashtable since the current version of the ESC/Java2 tool does not support reasoning about hashtable indirections. This mapping allows us to express the state of the data repository in a component's contract in terms of the object fields rather than hashtable indirections, and allows us to still reason about the repository state without losing the verification power of ESC/Java2.

In order to reduce the verification complexity and the overhead of instrumenting all library calls, we use a pragmatic framing approach to verify if a component's implementation obeys its contract. Instead of letting ESC/Java2 verify the *modifies* clauses, we use a component-specific specification of the session repository, in which we constrain the allowed write operations to the actual write interactions that the component claims to have in its *modifies* clauses.

Fig. 11 is an example of such a component-specific annotation to use with the *ShowCartServlet*: the precondition of the *setAttribute* method states that only write operations are allowed for the *cart* and *currency* data item. In contrast to the complete specification of the session repository (Fig. 10), the *messages* data item may not be modified by the *ShowCartServlet*.

In case the component's implementation triggers an unspecified state change in the shared data repository, the verification of the component with ESC/Java2 will detect this contract viola-

```

package javax.servlet.http;

public interface HttpSession {
    // @ public ghost Object cart;
    // @ public ghost Object currency;
    // @ public ghost Object messages;

    // @ requires false;
    // @ also
    // @ requires name == "cart";
    // @ ensures this.cart == value;
    // @ modifies this.cart;
    // @ also
    // @ requires name == "currency";
    // @ ensures this.currency == value;
    // @ modifies this.currency;
    // @ also
    // @ requires name == "messages";
    // @ ensures this.messages == value;
    // @ modifies this.messages;
    public void setAttribute (String name, Object value);

    // @ requires false;
    // @ also
    // @ requires name == "cart";
    // @ ensures \result == this.cart;
    // @ modifies \nothing;
    // @ also
    // @ requires name == "currency";
    // @ ensures \result == this.currency;
    // @ modifies \nothing;
    // @ also
    // @ requires name == "messages";
    // @ ensures \result == this.messages;
    // @ modifies \nothing;
    public /* @ pure */ Object getAttribute (String name);

    // @ requires false;
    public void removeAttribute (String name);
}

```

Fig. 10. JML contract of the session repository (HttpSession.spec)

tion (even without checking the component's *modifies* clauses), since the state change will also violate the precondition of the component-specific *setAttribute* annotation of the shared repository.

The full component contracts of the Duke's BookStore validation experiment (both in the problem-specific contract language and in JML), the contract translation tool and generator tool for the component-specific repository specification can be found on the paper's accompanying website [17].

B. Application-specific Property Verification

A server-side check method is automatically generated from the intended client-server protocol, to statically verify that no client-server interaction violates the *no broken data dependencies* property. This check method simulates the intended protocol in a server-side method body, in which every web interaction is translated into a method call to the appropriate request processing component (if needed preceded by one or more filters). In addition, reactive or non-deterministic behavior is translated by applying the *java.util.Random* class, if-then-else branches, switch-cases and while-loops. The protocol-simulating check method for

```

//@ requires request != null;
//@ requires request.session.messages == null && request.session.cart == null && request.session.currency == null;
public void protocolCheck(HttpServletRequest request, HttpServletResponse response){
    try {
        Random random = new Random();
        bookstore.doGet(request, response);
        while(random.nextBoolean()){
            int randomInt = random.nextInt ();
            switch(randomInt){
                case 0: showcart.doGet(request, response); break;
                case 1: banner.doGet(request, response); break;
                case 2: bookstore.doGet(request, response); break;
                case 3: bookdetail.doGet(request, response); break;
                default: break;
            }
        }
        if (random.nextBoolean()){
            switch(random.nextInt ()) {
                case 0: cashier.doGet(request, response); break;
                default: catalog.doGet(request, response); break;
            }
        }
        while(random.nextBoolean()){
            switch(random.nextInt ()) {
                case 0: showcart.doGet(request, response); break;
                case 1: catalog.doGet(request, response); break;
                case 2: cashier.doGet(request, response); break;
                case 3: bookstore.doGet(request, response); break;
                case 4: bookdetail.doGet(request, response); break;
                case 5: banner.doGet(request, response); break;
                default: break;
            }
        }
        orderFilter.doFilter ( request, response, null );
        receipt.doPost( request, response );
    }
    catch(Exception e) { e.printStackTrace (); }
}

```

Fig. 12. Protocol-simulating check method to be verified by ESC/Java2

the Duke's BookStore application is listed in Fig. 12.

The application-specific property verification is then reduced to statically verifying the implementation of the check method with ESC/Java2. Compliance to a component's assumption on the shared session state is verified implicitly because static verifiers such as ESC/Java2 check that the preconditions are fulfilled for each method that is called. Since the static verifier verifies all possible execution paths in the check method, the *no broken data dependencies* property is verified for all transitions of the *intended client-server protocol* and for all its prefixes.

For the application-specific protocol verification, ESC/Java2 relies on the explicit framing conditions of the different component contracts, in combination with the full specification of the shared session repository from Fig. 10.

C. Run-time Protocol Enforcement

Since the static verification step requires that the protocol at run time adheres to the intended client-server protocol, run-time enforcement is needed to ensure that only requests conform the intended protocol are processed by the application.

As a proof of concept, we embedded a lightweight run-time enforcement engine in our web application container by installing a custom J2EE Filter. Before a servlet is invoked by means

of the *service(ServletRequest request, ServletResponse response)* method in a J2EE web application, a chain of deployed filters is always applied to the request.

At deployment time, our enforcement engine is loaded with an object-oriented instantiation of the labeled state transition system (Fig. 13). For each user session the current state is stored, and for each incoming web request, the enforcement engine verifies that the transition is allowed and the current state is updated before the request is dispatched to the servlet. In case of a protocol violation, a pluggable strategy is consulted, defining the action that should be taken ranging from blocking access to the originator's IP or invalidating the user's session to just logging the access violation.

The proposed property verification assumes sequential processing of requests within a user's session. In order to guarantee the absence of broken data dependencies in a multi-threaded server environment, the run-time protocol enforcement of our prototype uses coarse-grained locking on the user's *HttpSession* object. By doing so, all requests belonging to one session are processed sequentially, while the server still can process multiple sessions in parallel.

```

package javax. servlet . http ;

public interface HttpSession {
    // @ public ghost Object cart ;
    // @ public ghost Object currency ;
    // @ public ghost Object messages ;

    // @ requires false ;
    // @ also
    // @ requires name == "cart" ;
    // @ ensures this . cart == value ;
    // @ modifies this . cart ;
    // @ also
    // @ requires name == "currency" ;
    // @ ensures this . currency == value ;
    // @ modifies this . currency ;
    public void setAttribute ( String name , Object value ) ;

    // @ requires false ;
    // @ also
    // @ requires name == "cart" ;
    // @ ensures \ result == this . cart ;
    // @ modifies \ nothing ;
    // @ also
    // @ requires name == "currency" ;
    // @ ensures \ result == this . currency ;
    // @ modifies \ nothing ;
    // @ also
    // @ requires name == "messages" ;
    // @ ensures \ result == this . messages ;
    // @ modifies \ nothing ;
    public /* @ pure @ */ Object getAttribute ( String name ) ;

    // @ requires false ;
    public void removeAttribute ( String name ) ;
}

```

Fig. 11. Component-specific specification of the repository (HttpSession.spec for ShowCartServlet)

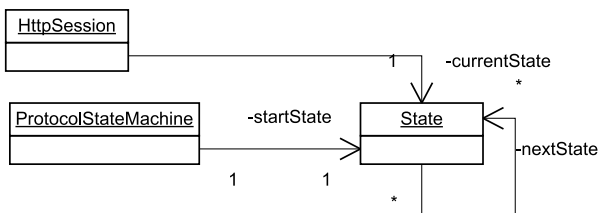


Fig. 13. Class diagram of the run-time enforcement engine

VII. EXPERIMENTAL RESULTS

We successfully applied the proposed solution to verify the *no broken data dependencies* property in the Duke's BookStore web application. We used this validation experiment to measure the annotation cost, the verification performance and the run-time overhead of the proposed solution.

A. Annotation Overhead

As a quantification of annotation overhead, a specification line count is performed on the annotated components. At most 4 lines of specification per component are used to express the interactions with the shared session repository. In addition, thanks to the pragmatic framing no library calls need to be instrumented to verify the different components.

B. Static Verification Performance

To evaluate the performance of the static verification process, the verification time is measured. The performance tests were run on a Pentium Mobile (1.4GHz) with 512MB RAM, running Debian Linux, while using Java 1.4.2.09, ESC/Java2 2.0a9 and Simplify 1.5.4. Table II shows the performance results of verifying the implementation compliance. Notice that the control flow complexity of the *ShowCartServlet* and *CatalogServlet* components exceeded the verification power of ESC/Java2 and the underlying theorem prover Simplify, so that we had to split up the *doGet* method for both components in order to get them verified.

The verification of the protocol-simulating check method succeeded smoothly in about 13.5 seconds.

C. Run-time Enforcement Overhead

To estimate the overhead of the run-time flow enforcement, we ran the following experiment on the BookStore application with and without our enforcement filter. We sequentially simulated 1000 different visitors, in which each user's protocol consisted of 6 web requests and 2 % of the visitors applied forceful browsing. For the experiment, the BookStore application has been deployed on the Sun Java System Application Server Platform Edition 8.2.

We ran 40 simulations, alternating the bookstore with and without run-time protocol enforcement. The processing time of these 40 simulations are shown in Fig. 14. We noticed an increasing processing time because of the increasing number of open sessions on the server. In this experiment, we measured a worst-case run-time overhead of 5.4 %, although the average run-time overhead is much lower.

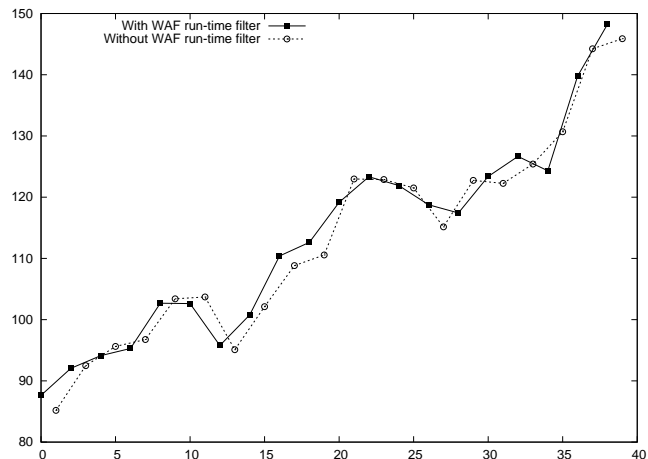


Fig. 14. Results of the overhead measurement experiment

The validation experiment in the Duke's BookStore application showed that the proposed solution is applicable to real-life web applications. The problem-specific contracts led to a small annotation overhead, and an acceptable verification performance was achieved thanks to the modular verification. By combining static and dynamic verification, the run-time overhead was also limited.

Moreover, similar experiments to specify and verify shared data interactions on the request scope show that this type of annotation and verification easily scales to larger web application [13].

In addition, the proposed solution is interoperable with the existing web infrastructure and does not interfere with other web

TABLE II
VERIFICATION PERFORMANCE

Component	Verif. time	Code lines
BookDetailsServlet	55.489 s	74
BookStoreServlet	19.085 s	61
ReceiptServlet	6.443 s	65
ShowCartServlet	6.258 + 240.634 s	157

Component	Verif. time	Code lines
OrderFilter	21.510 s	61
CashierServlet	68.699 s	60
CatalogServlet	6.380 + 222.301 s	123

security solutions. Moreover, the proposed solution is able to leverage the power of existing Web Application Firewalls by providing formal techniques to prove the absence of broken data dependencies in a given WAF protocol enforcement configuration. This partially counters an important drawback of nowadays WAFs, that have a limited coverage in protecting implementation-specific vulnerabilities.

VIII. DISCUSSION

In this paper, the required usability characteristic has been an important driver for the solution, and therefore, we deliberately chose a developer-centric point of view. This developer-centric point of view significantly influenced the proposed solution. In every step of the solution, we aimed to achieve the right trade-off between the accuracy and power of formal verification on the one hand, and limited overhead for the developer and composer and applicability to real-life software applications on the other hand. In this section, we first discuss the most important trade-off decisions of our prototype in more detail. Next, we highlight opportunities to further improve the usability of the proposed solution.

1) *Problem-specific annotations and pragmatic framing:* We used a problem-specific annotation language to significantly reduce the annotation overhead for the developer. However, this partial specification also implied that, as a consequence, the framing conditions of the different components were partially specified as well, and that the traditional framing verification was not feasible in such scenarios.

In this paper, we used a pragmatic framing approach to only verify state changes on the shared data repository. Since other state changes were neglected on purpose, this framing approach only guaranteed correct framing regarding to state changes on the shared session repository. Although such a pragmatic framing is not applicable for general verification purposes, this framing approach was sufficient for our verification process since we were only interested in the component's interactions with the shared repository.

In fact, the pragmatic framing approach was an interesting enabler for our approach. Thanks to the pragmatic framing, we were able to verify partially specified components, i.e. we only specified the parts of the contract that we were actually interested in (namely the interactions with the shared repository). In case we would have applied traditional framing, we would also had to specify every state change that occurred in the application by executing a component's method, as well as to annotate every called library method with its appropriate frame condition.

2) *Use of ESC/Java2:* In our prototype, we have chosen to use the ESC/Java2 verifier. The main advantage of this verifier is that it requires no user interaction. On the downside, the verifier is far from complete and has some known sources of unsoundness [21].

We will shortly discuss how this did impact our verification process.

a) *Limited reasoning about hashtable indirections:* One of the problems that we were confronted with in our prototype was the poor support of the ESC/Java2 tool to reason about hashtable indirections. We were forced to circumvent this lack of support by introducing ghost variables and verbose specifications of the shared repository. This is a temporary problem and future versions of the tool are expected to incorporate better hashtable support.

b) *Default framing:* When reasoning about a call to a routine, ESC/Java2 assumes that the routine only modifies its specified modification targets (as given in modifies pragmas). As defined in JML, ESC/Java2 has a default for missing modifies clauses (i.e. *modifies everything*) to unhide unexpected changes to variables caused by calling a routine. However, logic to reason about routine bodies that contain these modifies clauses has not yet been implemented in ESC/Java2. As a result, methods without explicit modifies clauses can be verified since the default frame condition includes everything. But, the current implementation of ESC/Java2 does not take this default frame condition into account when such methods are called, resulting in an unsound verification.

In our prototype, we annotated every method that updates the shared repository with an appropriate framing condition. By doing so, we prevented that calls to methods without a framing condition could break data dependencies without being detected.

c) *Loops:* To avoid the need for loop invariants, ESC/Java2 implements an approximative and possibly unsound verification of loops. Since the influence of loop bodies on the state of the repository is typically limited, this did not impact our verification of Duke's BookStore. In fact, inferring loop invariants about the state of the repository seems feasible and is an interesting item for future work. With inferred loop invariants, this potential source of unsoundness could be removed completely.

d) *Unsound pragmas:* ESC/Java2 allows users to introduce assumptions into the verification process by using unsound pragmas such as *assume*. When these assumptions are invalid, the verification is unsound. In the Duke's BookStore application, we needed to instrument some of the components with *assume* pragmas to assist the verifier in the verification process, e.g. by supplying type information while iterating over *Collections*. Notice that these annotations are needed because the lack of generics in Java 1.4, and are not necessary in later versions of Java.

All these issues are limitations of the current tool, and can be expected to disappear with improved technology for Java program verification. Our experience with this experiment shows that state-of-the-art program verification technology is already useful today and further improvements to the technology will further increase that usefulness.

To valorize this research in a concrete developer's tool, the balance between usability and accuracy would probably even shift further in favor of the developer and composer in several steps of the solution:

1) *Problem-specific annotation and contract verification*: To reduce the annotation overhead for the software developer, the problem-specific annotation of a component can be inferred from its implementation. This specification inference significantly lowers the impact on the development process, but also implies that some implementation bugs will only be identified during the property verification phase (typically close to deployment), rather than during the contract verification phase in which mismatches can already be detected between the problem-specific annotation and the component's implementation. Moreover, the specification inference eliminates the need for checking the compliance between problem-specific annotation and the component's implementation.

2) *Intended client-server protocol*: To verify the *no broken data dependencies* property statically, an upper bound is defined for the client-server interactions, namely the intended client-server protocol. At this moment, the intended client-server protocol is constructed manually, based on the expected use of the web application and the URLs generated by the different web pages. In order to reduce the composer's involvement in the verification process, this application specific protocol can also be constructed automatically based on a representative client implementation (e.g. in case of a rich web client), by analyzing the hyperlinks generated by the different web pages or by observing legitimate web traffic as is often done in Web Application Firewalls.

3) *Integrated tool*: To be practically useful for software developers, providing an integrated tool is essential. The different subtasks of the solution can be integrated in a single framework-specific tool, and the tool is preferably embedded in the developer's Integrated Development Environment (IDE). In addition, extra support is needed to give useful feedback to the developer in case the *no broken data dependencies* property verification fails. By doing so, the software developer is able to build more secure and reliable web applications, while he is completely shielded from the underlying specification and verification process.

IX. RELATED WORK

The work presented in this paper is related to a broad spectrum of ongoing research. We only present some key pointers for each of the domains, and present in more detail for the domain most related to the proposed solution, namely static and dynamic verification in web application security.

Several implementation-centric security countermeasures for web applications have already been proposed [22]–[28], but most of them focus on injection attacks (SQL injection, command injection, XSS, ...) and use tainting, pointer or data flow analysis. Our solution targets another set of implementation bugs, namely bugs due to broken data dependencies on the server-side session state and to do so we rely on the static and dynamic verification of component contracts.

Gould *et al.* also aim to reduce the number of run-time errors in web applications by applying static verification [29]. Their solution focusses on the reduction of SQL run-time exceptions and uses a static analysis tool to verify the correctness of all dynamically generated query strings within an application. Our solution is based on program annotations and we verify

interactions between components and the non-persistent, server-side state.

We combine in our solution static and dynamic verification to reduce the run-time enforcement overhead. The idea of combining static and dynamic verification is not new, and is for instance already adopted by Yao-Wen Huang *et al.* in securing web application against web vulnerabilities caused by insecure information flow, such as SQL injection, XSS and command injection [30]. Their approach uses a lattice-based static analysis algorithm for verifying information flow based on type systems and type state. Sections of code considered vulnerable are automatically instrumented with run-time guards. In contrast, our approach aims to reduce run-time errors due to composition problems. In addition, our approach is based on program annotations and the verification of component preconditions.

In [31], Jeff Offutt *et al.* generate bypass tests which check if an online web application is vulnerable to forceful browsing or parameter tampering attacks. The bypass tests are blackbox tests using data that circumvents client-side checks. They define three levels of fault injection: bypass tests at the value level, at the parameter level and at the control flow level. Since the fault injections are based on violations of the client-side validations, they operate independent of the server implementation and do not give formal guarantees about the absence of bugs. In contrast our verification approach is able to guarantee the absence of errors at the control flow level, and in future work we like to investigate how well our approach is suited to counter errors at the other two levels as well.

Firewall configuration analysis is proposed to manage complex network infrastructures (such as networks with multiple network firewalls and network intrusion detection systems) [32], [33]. These approaches aim to achieve efficiency and consistency between the different network-layer security devices, whereas our approach focuses on the application-layer consistency between the WAF and the web application.

Since more than a decade, software architectures are used to abstract reasoning about software systems from source code level towards coarse-grained architectural elements and their interconnections [2], [34]–[36]. Architectural styles abstract reoccurring patterns of components, connectors and behavioral interactions within different software architectures and try to capture the advantages or main characteristics of a particular architectural style, as well as the constraints introduced by the style. In [2], Shaw and Garlan proposed a taxonomy of different architectural styles, including the data-centered style.

To support architecture-based reasoning, (semi-)formal modeling notations and analysis techniques are required. Several Architecture Description Languages (ADLs) are proposed for architectural specification and analysis. Although these ADLs strongly vary in the abstractions and analysis capabilities they provide, most ADLs explicitly provide abstractions for components, connectors and their behavioral interactions, as well as tool support for analysis and architecture-based development [36], [37]. However, in most cases a discontinuity exists between the architectural model and the actual implementation model, making outcomes of architectural analysis meaningless. To counter this, ArchJava [38] offers a unique binding between architectural description and actual implementation, but ArchJava does not yet provide indirect sharing verification.

Component contracts have already often been proposed before

for various purposes [39]–[42]. For components written in Java, the Java Modeling Language (JML) [16] is a popular formal contract specification language. The use of JML or related languages such as Spec# [43] for verifying component properties is a very active research domain. For example, Jacobs *et al.* [44] verify absence of data races and Pavlova *et al.* [45] focus on security properties of applets. Other applications of JML are surveyed in [19].

The research presented in this paper proposes a pragmatic solution to broken session dependencies in web applications. The main advantage of such pragmatism is the potential of short term applicability. But of course, research on more fundamental approaches is also needed and can have more substantial impact in the long term. There is a large body of research on how to improve programming languages for programming distributed applications. In his keynote speech at ICSE 2005, Cardelli discussed three important areas where improvements are important: asynchronous concurrency, dealing with semi-structured data and additional security abstractions [46]. The programming language E [47] is an example of a language that has emphasized security in its design. Other languages focus on specific classes of Internet applications, such as distributed consensus applications [48].

X. CONCLUSION

In this paper, we have presented an approach to improve the security and reliability of web applications by guaranteeing the absence of run-time errors. In particular, we have proposed a solution to prevent run-time errors due to broken data dependencies on session data.

Our solution combines development-time program annotation, static verification and run-time checking to provably protect against broken data dependencies in web applications. We designed and developed a prototype implementation building on the Java Modeling Language (JML) and the static verifier ESC/Java2. In addition, we successfully applied our approach to Duke's BookStore, a representative J2EE-based e-commerce application.

Our solution also provides a good trade-off between usability and verification power. Because of some well-considered developer-centric design decisions in our prototype, the validation experiment showed a limited overhead and demonstrated the applicability of the presented approach to real-life applications. In addition, the proposed solution is interoperable with the existing web infrastructure and does not interfere with other web security solutions. Moreover, the proposed solution is able to leverage the power of existing Web Application Firewalls by providing formal techniques to prove the absence of broken data dependencies in a given WAF protocol enforcement configuration.

To the best of our knowledge, the research presented in this paper is the first to improve web application security by providing an appropriate solution to the specific problem of broken data dependencies on the session data.

ACKNOWLEDGMENT

The authors would like to thank Wolfram Schulte (from Microsoft Research), Bart Jacobs, Adriaan Moors and Jan Smans (from the Katholieke Universiteit Leuven) for their useful comments and insight in some interesting discussions on this research.

REFERENCES

- [1] P. G. Neumann, "Keynote speech: System and Network Trustworthiness in Perspective," in *13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*.
- [2] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.
- [3] Sun Microsystems, Inc., "Java Servlet Technology," <http://java.sun.com/products/servlet/>.
- [4] V. Samar, "Unified login with pluggable authentication modules (PAM)," in *Proceedings of the 3rd ACM conference on Computer and Communications security, CCS 1996*. ACM Press, 1996, pp. 1–10.
- [5] E. Freeman, K. Arnold, and S. Hupfer, *JavaSpaces Principles, Patterns, and Practice*. Essex, UK, UK: Addison-Wesley Longman Ltd., 1999.
- [6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," <http://www.ietf.org/rfc/rfc2616.txt>, June 1999, Request For Comments: 2616 (Category: Standards Track).
- [7] V. Raghvendra, "Session tracking on the web," *Internetworking*, vol. 3, no. 1, March 2000.
- [8] Karl Forster, Lockstep Systems, Inc., "Why Firewalls Fail to Protect Web Sites," <http://www.lockstep.com/products/webagain/why-firewalls-fail.pdf>.
- [9] I. Ristic, "Web application firewalls primer," *(IN)SECURE*, vol. 1, no. 5, pp. 6–10, January 2006.
- [10] S. Pettit, "Anatomy of a web application: Security considerations," Sanctum, Inc., Tech. Rep., Jul. 2001.
- [11] webSecurity, Inc., "The Weakest Link: Mitigating Web Application Vulnerabilities," http://www.websecurity.com/pdfs/webapp_vuln_wp.pdf.
- [12] Web Application Security Consortium, "Web Application Firewall Evaluation Criteria, version 1.0," <http://www.webappsec.org/projects/wafec/>, January 2006.
- [13] L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten, "Static Verification of Indirect Data Sharing in Loosely-coupled Component Systems," in *Software Composition*, ser. Lecture Notes in Computer Science, vol. 4089. Springer Berlin / Heidelberg, 2006, pp. 34–49.
- [14] E. Armstrong, J. Ball, S. Bodoff, D. B. Carson, I. Evans, D. Green, K. Haase, and E. Jendrock, *The J2EE 1.4 Tutorial*. Sun Microsystems, Inc., December 2005.
- [15] L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten, "Dependency analysis of the Gatormail webmail application," Department of Computer Science, K.U.Leuven, Leuven, Belgium, Report CW 427, Sep. 2005.
- [16] G. T. Leavens, "The Java Modeling Language (JML)," <http://www.jmlspecs.org/>.
- [17] L. Desmet, P. Verbaeten, W. Joosen, and F. Piessens, "Provable protection against web application vulnerabilities related to session data dependencies," <http://www.cs.kuleuven.be/~lieven/research/TSE2007/>.
- [18] A. D. Raghavan and G. T. Leavens, "Desugaring JML method specifications," Iowa State University, Department of Computer Science, Tech. Rep. 00-03e, May 2005.
- [19] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 3, pp. 212–232, Jun. 2005.
- [20] KindSoftware, "The Extended Static Checker for Java version 2 (ESC/Java2)," <http://secure.ucd.ie/products/opensource/ESCJava2/>.
- [21] J. R. Kiniry, A. E. Morkan, and B. Denby, "Soundness and completeness warnings in esc/java2," in *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*. New York, NY, USA: ACM Press, 2006, pp. 19–24.
- [22] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, 2005, pp. 124–145.
- [23] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 303–311.
- [24] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *SEC, R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, Eds. Springer, 2005, pp. 295–308*.
- [25] W. G. J. Halfond and A. Orso, "Amnesia: analysis and monitoring for neutralizing sql-injection attacks," in *ASE '05: Proceedings of*

the 20th IEEE/ACM international Conference on Automated software engineering. New York, NY, USA: ACM Press, 2005, pp. 174–183.

- [26] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks," in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006, pp. 9–9.
- [27] V. B. Livshits and M. S. Lam, "Finding security errors in Java programs with static analysis," in *Proceedings of the 14th Usenix Security Symposium*, Aug. 2005, pp. 271–286.
- [28] N. Jovanovic, C. Kruegel, and E. 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*. New York, NY, USA: ACM Press, 2006, pp. 27–36.
- [29] C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 645–654.
- [30] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *WWW '04: Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA: ACM Press, 2004, pp. 40–52.
- [31] J. Offutt, Y. Wu, X. Du, and H. Huang, "Bypass testing of web applications," in *ISSRE*. IEEE Computer Society, 2004, pp. 187–197.
- [32] T. E. Uribe and S. Cheung, "Automatic analysis of firewall and network intrusion detection system configurations," in *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*. New York, NY, USA: ACM Press, 2004, pp. 66–74.
- [33] K. Golnabi, R. K. Min, L. Khan, and E. Al-Shaer, "Analysis of Firewall Policy Rules Using Data Mining Techniques," in *10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006)*, April 2006, pp. 305–315.
- [34] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.
- [35] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
- [36] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, 2000.
- [37] P. C. Clements, "A survey of architecture description languages," in *Proceedings of the 8th International Workshop on Software Specification and Design*. IEEE Computer Society, 1996, p. 16.
- [38] J. Aldrich, "Using types to enforce architectural structure," Ph.D. dissertation, University of Washington, August 2003.
- [39] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.
- [40] B. Liskov, *Abstraction and specification in program development*. Cambridge, MA, USA: MIT Press, 1986.
- [41] Y. L. Traon, B. Baudry, and J.-M. Jezequel, "Design by Contract to Improve Software Vigilance," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 571–586, 2006.
- [42] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
- [43] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview," *Lecture Notes in Computer Science*, vol. 3362/2005, pp. 49–69, January 2005.
- [44] B. Jacobs, K. R. M. Leino, F. Piessens, and W. Schulte, "Safe concurrency for aggregate objects with invariants," in *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, 2005, pp. 137–146.
- [45] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet, "Enforcing high-level security properties for applets," in *CARDIS*, J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A. A. E. Kalam, Eds. Kluwer, 2004, pp. 1–16.
- [46] L. Cardelli, "Transitions in programming models: 2," in *ICSE '05: Proceedings of the 27th International conference on Software Engineering*. New York, NY, USA: ACM Press, 2005, p. 2.
- [47] M. S. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. dissertation, Johns Hopkins University, May 2006.
- [48] J. C. M. Baeten, H. M. A. van Beek, and S. Mauw, "Specifying internet applications with dicons," in *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*. New York, NY, USA: ACM Press, 2001, pp. 576–584.



Dept. of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium; Lieven.Desmet@cs.kuleuven.be.



Pierre Verbaeten is a full professor at Katholieke Universiteit Leuven's Department of Computer Science; he is currently head of the department. His research interests include open system software, dynamic configuration and integration, and ad hoc networks. He received his PhD in computer science from K.U. Leuven. He's a member of the ACM and IEEE. Contact him at the DistriNet Research Group, Dept. of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium; Pierre.Verbaeten@cs.kuleuven.be.



Celestijnenlaan 200A, B-3001 Leuven, Belgium; Wouter.Joosen@cs.kuleuven.be.



Frank Piessens is a professor at the Department of Computer Science of the Katholieke Universiteit Leuven, Belgium. His research interests lie in software security, including security in operating systems and middleware, architectures, applications, Java and .NET, and software interfaces to security technologies. He is an active participant in both fundamental research and industrial application-driven projects, provides consultancy to industry on distributed system security and serves on programme committees for various security-related international scientific conferences. Contact him at the DistriNet Research Group, Dept. of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium; Frank.Piessens@cs.kuleuven.be.