Aegis: Automatic Enforcement of Security Policies in Workflow-driven Web Applications*

Luca Compagna
SAP Labs France
luca.compagna@sap.com

Daniel R. dos Santos
Fondazione Bruno Kessler
SAP Labs France
dossantos@fbk.eu

Serena Elisa Ponta
SAP Labs France
serena.ponta@sap.com

Silvio Ranise
Fondazione Bruno Kessler
ranise@fbk.eu

ABSTRACT
Organizations often expose business processes and services as web applications. Improper enforcement of security policies, e.g., authorization and control-flow integrity, in these applications leads to business logic vulnerabilities that are hard to find and may be devastating. **Aegis** is a technique and tool to automatically synthesize run-time monitors to enforce control-flow and data-flow integrity, as well as authorization policies and authorization constraints in web applications. The enforcement of these properties can mitigate attacks such as forceful browsing, authorization bypass, and workflow violations, while allowing regulatory compliance in the form of, e.g., Separation of Duty. **Aegis** also solves the satisfiability problem for constrained applications, i.e., determining if there is a possible execution of the application in which all tasks can be performed without violating the security policy. We evaluate **Aegis** on a set of real-world benchmark applications, assessing the enforcement of policies, mitigation of vulnerabilities, and performance overhead.

Categories and Subject Descriptors

Keywords
Web Application; Policy Enforcement; Workflow Satisfiability

1. INTRODUCTION
Web applications are nowadays one of the preferred ways of exposing business processes and services to users. Many web applications implement workflows, i.e. there is a pre-defined sequence of tasks that must be performed by users to reach a goal [8]. If an application does not correctly enforce its workflows, attackers can exploit this vulnerability to subvert it. In an e-commerce application, for instance, users must **Select products, Checkout, Enter shipping information, Enter payment information** and **Confirm**. If the application does not verify that user actions follow this sequence, a user can, e.g., skip the payment step and receive products without paying. Even errors generated while accessing pages in an unexpected order, can be exploited [27]. Workflow and business logic vulnerabilities are listed in the Common Weakness Enumeration (CWE) 1 in the OWASP Testing Guide [35] and (tangentially) in the OWASP Top 10 [38]. Control-flow integrity, i.e. the enforcement of an application’s workflow, has been used in web applications to prevent workflow attacks and others, e.g., forceful browsing and race conditions [8, 9].

Data-flow integrity is also crucial and incorrect enforcement can lead to vulnerabilities where, e.g., a user can change the price of a product being purchased to pay less for it [53]. This kind of vulnerability is even more prominent in multi-party scenarios, where a user receives data such as tokens from one party (e.g., an identity provider) and must relay them to another party (e.g., a service provider). Several vulnerabilities have been discovered in recent years due to improper enforcement of data-flow integrity [53, 59, 41].

Besides control- and data-flow integrity, access control is fundamental for web application security whenever users must access only data and functionalities that they are authorized to by a given policy. Access control vulnerabilities are common and hard to find [45]. Moreover, some web applications implement collaborative work, in which many users work together to complete a workflow. Examples are Enterprise Resource Planning (ERP) software, used by employees in an organization to, e.g., manage purchases; and e-health applications, used by doctors and technical staff to manage patient records. In these applications, not only it is important to enforce authorization policies, but it may also be necessary to support authorization constraints, which impose more restrictions on what users can do at run-time. Examples of such constraints are Separation or Binding of Duty (BoD or SoD), requiring two different users (same user, respectively) to execute a pair of tasks; and cardinality, limi-
ing the number of tasks that a single user can execute in a workflow. These constraints can be used to avoid errors and frauds in security-critical applications that must follow regulatory compliance rules. Nonetheless, none of the applications we experimented with provided support for an easy to use, declarative specification of constraints. Including Odoo\textsuperscript{3}, an open-source ERP platform with more than 5000 developers and 2 million users, among them big companies such as Toyota and Danone. Without declarative specifications and proper enforcement, authorization constraints have to be implemented as application code embedded into each task \textsuperscript{5} or translated to static assignments in the authorization policy. Both solutions are error-prone and can hardly scale.

Even with suitable specification and enforcement mechanisms, support for authorization policies and constraints may lead to situations where an application workflow cannot be completed because no user can execute an action without violating them. Determining if such a situation can be avoided, i.e. if a workflow can be completed in the presence of a policy and constraints, is known as the Workflow Satisfiability Problem (WSP) \textsuperscript{49}. The WSP has received much attention in the workflow security community \textsuperscript{29}, but, to the best of our knowledge, has never been considered in web applications. In fact, transferring WSP solutions to the web domain is not trivial. These solutions often rely on a workflow model specification and a workflow management system to handle the control-flow of tasks and to provide an interface for users to request task executions, elements which are frequently not available for web applications.

In this paper, we present Aegis\textsuperscript{1}, a technique to synthesize run-time monitors for web applications that are capable of automatically (i) enforcing security policies composed of combinations of control- and data-flow integrity, authorization policies, and authorization constraints; and (ii) solving the run-time version of the WSP by granting or denying, at run-time, requests of users to perform tasks based on the satisfaction of the policy and constraints and the possibility to terminate the current workflow instance. Aegis is based on \textsuperscript{6}, where a technique to synthesize run-time monitors that solve the WSP for security-sensitive workflow models was presented. We extend \textsuperscript{6} by supporting data integrity. To synthesize a monitor, Aegis first infers, using process mining \textsuperscript{17}, workflow models of the target application from a set of HTTP traces representing user actions. Traces must be manually edited to contain only actions that should be controlled by the monitor. Inferred models are Petri nets \textsuperscript{30} labeled with HTTP requests representing tasks and annotated with data-flow properties obtained by using a set of heuristics based on differential analysis (as in, e.g., \textsuperscript{53} \textsuperscript{14}). These Petri nets (or a user-friendly representation, e.g., BPMN \textsuperscript{52}) can be refined by a human user who, optionally, specifies authorization constraints and an authorization policy. A monitor is then generated from the model by pre-computing its possible executions \textsuperscript{6}. At run-time, a reverse proxy is used to (i) capture login actions to later establish the acting users, and (ii) capture incoming requests and query the monitor to either allow or deny the request. Aegis is completely black-box and can be used with new or legacy applications to support the enforcement of security-related properties or to mitigate logic vulnerabilities.

The main contribution of this paper is the description and implementation of Aegis, which paves the way for the support of complex security policies in workflow-driven web applications. Another contribution is an empirical evaluation on relevant applications, some taken from related work \textsuperscript{39}. The rest of this paper is organized as follows: Section 2 presents an overview using motivating examples; Section 3 details the three steps of the technique; Section 4 shows the implementation and evaluation of our work; Section 5 discusses the approach and limitations; and Section 6 concludes the paper.

2. OVERVIEW

Aegis synthesizes run-time monitors for workflow-driven web applications, i.e. applications implementing business processes and customer services as workflows. Hereafter, web application is used as an abbreviation for workflow-driven web application, unless stated otherwise.

A monitor synthesized by Aegis can enforce three security-related properties: authorization policies (\(P\)), defining which users are entitled to perform which tasks; authorization constraints (\(C\)), defining run-time restrictions on the execution of tasks, e.g., a SoD requiring two different users to perform a pair of tasks; and control- and data-flow integrity (\(I\)), specifying the authorized control-flow paths that the application must follow, as well as data invariants. Different web applications have different enforcement needs, which allows for the synthesis of different configurations of monitors, depending on which properties are switched on or off. We identify each configuration as a tuple containing the active properties, e.g., \((P, C, I)\), \((P, I)\), \((C, I)\), \((I)\). Control- and data-flow integrity are in the same category because it is not realistic that an application needs to enforce one and not the other.

Aegis takes as input sets of HTTP traces representing user actions executed while interacting with a target web application. It synthesizes an external monitor composed of a set of queries to be used by a proxy sitting between users and the application. Each set of input traces is produced by a user simulating real clients completing a workflow as foreseen by the application ("good traces"). The monitor only enforces those workflows given as input by the user, having no impact on the rest of the application besides the overhead of a reverse proxy (which is frequently used in any case to implement, e.g., load balancing). Traces can be collected using test automation tools such as Selenium\textsuperscript{6} or ZAP\textsuperscript{6} and must be manually edited to contain only critical tasks. After trace collection, the whole technique is automated.

Figure 1 shows an overview of Aegis. The top of the Figure shows the entire approach, where rectangles represent the three main steps (with sub-steps), yellow notes are inputs, and ovals are generated artifacts. The bottom of the Figure details the internals of the Run-time Monitoring component. The three main steps are the following.

1. Model Inference. The set of HTTP traces is automatically stripped of all information except request and response URLs, headers, and bodies; each request and response is annotated with data-flow properties inferred by a set of heuristics; traces are aggregated into a file called event log; and a process mining tool takes the log as input to generate

\textsuperscript{3}https://www.odoo.com/

\textsuperscript{1}Aegis was the mythological shield carried by Athena, and “under the aegis of” means “under the protection of.”

\textsuperscript{5}http://www.seleniumhq.org/

\textsuperscript{6}https://goo.gl/XvxKd1
a Petri net workflow model whose transitions are labeled by the annotated requests. The inferred model can be refined according to the user's understanding of the application.

2. Monitor Synthesis. Given a workflow model, the user specifies the Authorization Constraints to be enforced (if any) and whether an Authorization Policy will be provided at run-time. Control- and data-flow integrity are obtained automatically from the model, can be modified by the user, and are always enforced. The workflow model is presented in a convenient BPMN notation, and the specification of constraints is done graphically. A run-time monitor capable of enforcing the chosen properties is synthesized by translating the model to a symbolic transition system (the translation among BPMN, Petri nets, and transition systems is automatic [6]) and computing a reachability graph, whose nodes are labeled by first-order formulae describing states in the execution of the workflow. The graph represents all possible valid executions of the workflow by symbolic users, allowing us to support different authorization policies at run-time. The Monitor is a set of queries derived from the graph.

3. Run-time Monitoring. A reverse proxy is instantiated with the synthesized monitor and a concrete authorization policy (if required). It sits between users and the application, filters requests and translates them to the monitor. The monitor enforces the properties defined in step 2, granting a request if the control-flow is respected, the data-flow invariants hold, the user issuing the request is authorized by the policy, the authorization constraints are not violated and the current instance execution can still terminate. The proxy, based on the response from the monitor, may forward requests to the application or drop them to prevent the violation of some property.

A single application may implement several workflows. Steps 1 and 2 are performed for each workflow to be monitored, generating one monitor per workflow. Step 3 uses all the synthesized monitors and queries the correct one depending on the incoming request. Requests not related to any monitored workflow go directly to the application, without triggering a monitor query.

Below, we present two motivating examples that illustrate the configurations \((P,C,I)\), \((C,I)\) (first example), and \((I)\) (second example). The first example motivates the distinctive contributions of our technique: support for authorization policies, constraints, and workflow satisfiability. The second example shows that Aegis is capable of mitigating logic vulnerabilities related to control- and data-flow integrity.

2.1 Example 1 - Enforcing constraints

Dolibarr\footnote{http://www.dolibarr.org/} is an open-source ERP web application for small and medium enterprises. It implements, among others, a business process similar to the one shown at the top of Figure 2 (in BPMN) to manage customer invoices.

The process contains 6 tasks (depicted by rounded boxes). Tasks \(t_1\) to \(t_4\) must be performed in sequence (as indicated by the solid arrows), while either \(t_5\), \(t_6\) or neither are performed last (as indicated by the diamond-shaped exclusive gateway). The original application implements each of the tasks shown in Figure 2. An authorization policy, control-flow, and possible data-flow invariants are implemented in an ad-hoc way, whose correctness is hard to verify, which may lead to vulnerabilities. The authorization policy originally supported by the application has a granularity of permissions that does not match the user-task assignment we support (there is no specific permission to, e.g., re-open an invoice or send it by e-mail). Authorization constraints are not supported. As a result, it is not trivial to prevent that a malicious user creates and validates a customer invoice (SoD between \(t_1\) and \(t_2\)) or inserts and validates a payment (SoD between \(t_3\) and \(t_4\)), which would allow him to, e.g., close invoices with an incorrect payment.

A user who wants to securely deploy this application can use Aegis to generate a \((P,C,I)\) monitor to enforce control-flow integrity, ensuring that all the steps in the customer invoice process are performed in the correct order; an authorization policy, ensuring that only authorized users can execute each task; and the SoD constraints described above, to avoid frauds. If the user prefers to leave authorization enforcement to the application, a \((C,I)\) monitor could be generated to only add support for constraints and integrity. To generate a monitor for the invoicing process, without impacting other parts of the application, the user starts by collecting traces simulating users performing the process. Some HTTP traces representing these executions are:

\[
\tau_1 = \{/invoice?action=create&value=10&prod=abc, /invoice(validate)?id=1, / invoice/pay/create?id=1&value=10, / invoice/pay/validate?id=1\},
\]

\[
\tau_2 = \{/invoice?action=create&value=20&prod=def, \}
\]

![Figure 1: Overview of the technique](image1.png)

![Figure 2: Customer invoice process in BPMN (top) and as a Petri net (bottom)](image2.png)
Each trace \( \tau_i \) represents one possible execution of the invoicing process and each request represents one task. The first four requests in each trace are essentially the same, but with different parameter values (e.g., \( \text{id} \) is 1 in \( \tau_1 \), 2 in \( \tau_2 \), and 3 in \( \tau_3 \)). They represent tasks \( t_1, t_2, t_3, \) and \( t_4 \). \( \tau_1 \) is an example of the branch where only the first four tasks are executed, while \( \tau_5 \) is executed after \( t_4 \) in \( \tau_2 \), and \( \tau_6 \) is executed after \( t_4 \) in \( \tau_3 \). The traces are automatically analyzed to extract data-flow properties, annotated and aggregated into an event log, sent to a process mining tool and the resulting Petri net labeled with a task-to-URL map (Step 1).

Figure 2 shows, at the bottom, the Petri net obtained from the process mining tool (ignore for a moment the dashed lines). The tasks in the nets are labeled as \( t_i \), with the following task-to-URL map:

- \( t_1 \): /invoice/action=create?value=<<I>>&prod=<<DC>>
- \( t_2 \): /invoice/validation?id=<<IID>>
- \( t_3 \): /invoice/validation?id=<<IID>>&value=<<I>>
- \( t_4 \): /invoice/validation?id=<<IID>>
- \( t_5 \): POST /invoice/send BODY id=<<IID>>
- \( t_6 \): /invoice/reopen?id=<<IID>>

Data-flow properties are represented by annotations on the URLs. The \( \text{<<IID>>} \) (instance identifier) annotation is applied to the elements used to bind all the requests to the same instance of a workflow, in this case the \( \text{id} \) parameter.

The \( \text{<<I>>} \) (invariant) annotation is applied to values that should not change during the workflow, in this example the value of the invoice in \( t_1 \) should be the same as the value of the payment in \( t_2 \). The \( \text{<<DC>>} \) (“don’t care”) annotation is applied to parameters that should be present in the request to help identify it as a unique action, but whose values are irrelevant. The parameter \( \text{prod2} \), which is present in the request of \( t_1 \) only in \( \tau_3 \), is dropped in the task-to-URL map because it is considered optional, i.e., a trace may represent an invoice with one or more products, so only the first \( \text{prod} \) parameter needs to be present. These data-flow properties, as well as others not used in this example, are obtained by using heuristics detailed in Section 3.

The user then specifies the constraints that must be enforced, shown as dashed lines labeled by \( \neq \) in Figure 2. The model is used to synthesize a monitor (Step 2), which is composed of a set of SQL queries like:

1. SELECT U2.ID FROM USERS AS U1, USERS AS U2, HST WHERE HST.dt1 AND NOT HST.dt2 AND NOT HST.dt3 AND NOT HST.dt4 AND NOT HST.dt5 AND NOT HST.dt6 AND NOT HST.t1by = U1.ID AND U2.ID IN (SELECT * FROM T2) AND U1.ID IN (SELECT * FROM T3) AND U2.ID IN (SELECT * FROM T4) AND U1.ID IN (SELECT * FROM T5)
2. \( \tau_5 = \{ (u1, t1), (u2, t2), (u3, t3), (u4, t4), (u5, t5), (u6, t6) \} \), where \( (u, t) \in \tau \) means that \( u \) is authorized to execute \( t \). The assignment is stored in the database, and a reverse proxy is instantiated with the synthesized monitor. The proxy is capable of receiving a request such as:

GET /invoice/validation?id=5 Cookie: sid=abcd1234

and identifying that it refers to task \( t_2 \) of instance 5 of the invoicing process being performed by user \( u2 \) (whose cookie \( \text{sid} \), sent in the header, has been stored during login). It then queries the monitor and, assuming \( u1 \) has previously executed \( t_1 \) and \( t_2 \) has not yet been executed, the query presented above is satisfied and the request is granted. On the other hand, a request can be blocked in several cases, such as if \( u3 \) tries to execute \( t_2 \) \( ((u3, t2) \notin \tau_4 \) i.e., not authorized by the policy), if \( u1 \) tries to execute \( t_2 \) (there is a SoD between \( t_1 \) and \( t_2 \)), if any user tries to execute \( t_3 \) before \( t_2 \) (because of the control-flow), or if any user issues a request for \( t_3 \) with a value different from the one sent for \( t_1 \) (because of the invariant).

To solve the WSP, regardless of the execution history, any request of \( u4 \) to execute \( t_3 \) should also be blocked. Granting that request would mean that the only user authorized to execute \( t_4 \) has already executed \( t_3 \), while both tasks are in SoD. Therefore, any execution where \( u4 \) performs \( t_3 \) would either not terminate or terminate with the violation of some constraint or policy. This choice between business compliance (not violating policies) and business continuity (terminating the execution) should be avoided. The synthesized monitor presents a transparent way of avoiding it by blocking requests that lead to an undesired situation. Exceptional situations, where it is preferable to violate the policy with the knowledge of an administrator, can be accommodated by using a soft enforcement mode, as discussed in Section 3.

### 2.2 Example 2 - Mitigating vulnerabilities

TomatoCart is a popular e-commerce application that implements the checkout process depicted on the top of Figure 3. It is composed of 5 tasks executed in sequence, where \( t_4 \) is a sub-process that can be implemented in different ways, but must produce a data object representing a token issued by a trusted third party, that is read in \( t_5 \).

This is an example of a multi-party web application, which implements the payment step by using third-party solutions such as PayPal. An execution of this workflow, using PayPal Express Checkout, involves three actors: a client \( C \), a service provider \( SP \) implementing TomatoCart and a trusted third party \( TTP \) implementing the payment provider. The execution starts with the client browsing the

---

1. \( \text{http://www.tomatocart.com/} \)
2. \( \text{https://www.paypal.com/} \)
SP, selecting some product \((t_1)\), requesting checkout \((t_2)\), and entering shipping information \((t_3)\). The SP then contacts the TTP and receives a token identifying the transaction (not shown in the workflow). The user is redirected to the TTP with the token \((t_4)\), completes the payment (again not shown in the Figure), and is redirected back to the SP passing the token, which is verified to complete the transaction \((t_5)\).

In version 1.1.7, TomatoCart had a vulnerability that allowed users to replay a PayPal Express Checkout token in \(t_5\) of a new transaction and shop for free \([44]\). This vulnerability was manually fixed in a later release of the application, but AEGIS could have been used to mitigate it until a patch was available (or until the patch could be applied, which is not always trivial). To mitigate the replay vulnerability, we can generate a monitor in the configuration \((\mathcal{I})\), enforcing control-flow integrity and the data invariant that the token received in \(t_4\) is the same that is sent in \(t_5\). An authorization policy and authorization constraints are not specified since every user can execute the steps in the checkout process and all steps are executed by the same user. Details of the communication between SP and TTP and between C and TTP are not shown in the workflow because the monitor only needs to enforce that no user can replace the token that has been sent to him/her. Although AEGIS ignores some messages, many vulnerabilities in multi-party web applications can be mitigated this way, because the vulnerabilities are commonly in the SP side \([53]\).

To generate the monitor, we repeat the steps presented for Example 1. Below, there are some traces of the execution of the TomatoCart checkout process, again simplified for readability. Now the traces involve three parties, thus each request must be identified with its host.

\[
\begin{align*}
t_1 & = \{ \text{shop.com/select}, \text{shop.com/checkout}, \text{shop.com/shipping}, \\
& \quad \text{shop.com/payment} \rightarrow \text{paypal.com/webscr?token=abcd1234}, \\
& \quad \text{shop.com/confirm?token=abcd1234} \}, \\
t_2 & = \{ \text{shop.com/select}, \text{shop.com/checkout}, \text{shop.com/shipping}, \\
& \quad \text{shop.com/payment} \rightarrow \text{paypal.com/webscr?token=efgh5678}, \\
& \quad \text{shop.com/confirm?token=efgh5678} \}.
\end{align*}
\]

Figure 3 shows the Petri net obtained for the checkout process, labeled directly with the URL of each task (where \(\rightarrow\) represents a redirect). The invariant annotation \(\langle I\rangle\) is applied to the token received from PayPal, specifying that its value must be the same in \(\text{/payment}\) and \(\text{/confirm}\).

A monitor is synthesized as before, however with neither authorization policy nor constraints. Workflow instances can be identified by the user identifier, since each user has only one checkout process at any given time. At run-time, whenever a user tries to replay a token, the monitor blocks this request because the token sent in \(t_5\) is different from the one received in \(t_4\) (since PayPal generates unique tokens). If the user tries to bypass the monitor by skipping step \(t_4\) and sending the token directly in \(t_5\), the monitor blocks the request because of a control-flow violation.

3. DETAILS

An HTTP trace (or a web session) is a sequence \(S = \{(u_1 : r_1, s_1), (u_2 : r_2, s_2), \ldots, (u_n : r_n, s_n)\}\) of pairs of web requests \(r_i\) issued by users \(u_i\) (which may or may not be all distinct) and responses \(s_i\). Each web request or response is defined as \(r_i = (\text{method, url, P})\), where method \(\in\{\text{GET, POST}\}\), url is the requested URL, and \(P\) is a set of parameters of the form \((k, v)\), which can be in the URL (in GET requests), the body (in POST requests) or in the headers (e.g., cookies or Location in redirects). Data values passed as JSON can be flattened to the same representation. The parameters in \(P\) represent the data values later annotated with data-flow properties.

A workflow \(W(T, U)\) is a set of tasks \((t \in T)\) in a causal order executed by a set of users \((u \in U)\), and a web application is composed of a set of workflows \(\Psi = \{W_1(T_1, U_1), \ldots, W_n(T_n, U_n)\}\). We take as input sets of web sessions \(W_S = \{S_1, S_2, \ldots, S_n\}\) and infer from each \(W_S\) a workflow \(W_i(T_i, U_i)\), using a process mining function \(P_M\), and a set of data property labels \(L_i\), using heuristics. We also take as input, optionally, sets of authorization constraints \(C_i\).

We then use a monitor synthesis procedure \(M_S(W, L, C)\) that returns a monitor \(M_i\). \(M_i\) is capable of answering requests of the form “can user \(u\) perform task \(t^?\)” — encoded as \(\text{can\_do}(u, t)\) — with True iff the control-flow in \(W_i\) and the data-flow in \(L_i\) are respected, no authorization constraint in \(C_i\) is violated, the requesting user \(u\) is authorized by an authorization policy \(\mathcal{TA}\) (specified at run-time), and the workflow can be executed until the end. \(\mathcal{TA}\) is not taken as input by \(M_S\) because the procedure can accommodate different authorization policies given at run-time.

**Attacks and enforcement.** At run-time, a reverse proxy \(RP\) receives an incoming request \(u : r\) and based on the information taken from it, tries to translate it into a query of the form \(\text{can\_do}(u, t)\), for \(u \in U\) and \(t \in T_i\) of workflow \(W_i(T_i, U_i)\), which can be answered by \(M_i\). Attacks on the application at the level of web requests are reflected on the workflows \([33]\) as shown below. The monitor can mitigate these attacks because they do not comply with the expected workflow (naturally, they are only mitigated in the parts of the application covered by the inferred model).

A request forgery is an extra request not foreseen in a workflow \(({r_1, r_2, \ldots, r_{\text{forged}}, \ldots, r_n})\). A workflow bypass is a missing request \(({r_1, r_2, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n})\). A workflow violation is an attempt to either repeat a unique request \(({r_1, r_2, \ldots, r_i, \ldots, r_n})\) or execute a request out of order \(({r_1, r_2, \ldots, r_{i+1}, \ldots, r_n})\). Authorization violations happen when a request is issued by a user who is not entitled to do so by the policy or when, for two tasks \(t_1\) and \(t_2\) in SoD, a user who previously issued a request \(r_1\) to execute \(t_1\), issues a new request \(r_2\) to execute \(t_2\).

**Adversary model.** The monitor enforces security properties related to access control and control- and data-flow integrity, ignoring vulnerabilities such as code injection. The target application is trusted, as well as any third parties trusted by it. Application users are not trusted, since they can be partially or fully controlled by an adversary.

3.1 Step 1 - Model inference

The goal of AEGIS is not to produce an accurate model of the whole application, but only workflow models containing a sequence of critical actions. Critical actions are the requests related to workflow tasks, whose execution should be controlled by the monitor. The definition of what is critical varies from application to application, but besides the usual noise in HTTP traces (e.g., requests loading images and other resources), any request that leaves the application state unchanged (e.g., AJAX requests for auto-completion of input fields) should be filtered out. Such requests are called navigation events, as opposed to system-interaction events,
which change the state of the application \[42\]. Not every system-interaction event should be controlled by the monitor (this should be decided by the user). However, discarding navigation events is crucial to keep the inferred models to a reasonable size and to eliminate imprecision due to variations in the process when executed by different users.

We assume that this treatment of the input traces is done before Aegis is invoked. It can be done manually by deleting unimportant actions, but there are proposals of automated black-box techniques to detect state changing requests. Some techniques detect a state change by sending identical requests and comparing their outputs \[17\] [18\], others are based on an abstraction of the user interface \[42\]. The former have limitations such as the need to isolate the application (other users cannot interact with it while a trace is collected) and to be able to reset it to its initial state. The latter cannot detect system states that are not reflected in the UI. Such techniques are usually embedded in crawlers to obtain a model of the entire application. Applying just the state-change detection part to traces of a single workflow may have sub-optimal results. Evaluating similar approaches to automate trace collection is left to future work.

Since some URLs in an application can take different parameters and different values for these parameters, while still representing the same action, and since we apply differential analysis to identify data-flow properties, we need at least two different traces as input, each containing a possible value for each of the parameters (including their presence and absence). The input traces should also represent all the possible execution paths of the process (control-flow). The number of input traces required for a precise model depends on the number of control-flow branches in the workflow being analyzed, as well as the diversity of the traces. Related works use, e.g., four traces as input \[53\] or traces with specific requirements for each of the parties in the process \[44\]. At least two login traces with distinct users must also be present, so that cookies defining the user session identifier and parameters representing user names can be mined, to map requests to concrete users at run-time.

It is possible to obtain input traces by reusing functional tests, which are common in web development and usually implemented using a framework such as Selenium. From the set of HTTP traces, we extract three artifacts: a workflow model, a task-to-URL map, and a set of data properties.

**Workflow model and map.** A workflow model is automatically obtained from a process mining tool. There are many well-known process mining algorithms and a simple example is the \( \alpha \)-algorithm \[17\]. It mines workflow nets by recording all the events in a log and detecting relations between them, such as sequence (\( \rightarrow \)), exclusive or (\( \# \)), and parallel (\( || \)). In the traces used in Example 1, it is possible to see that \( t_1 \) always precedes \( t_2 \) and \( t_2 \) never precedes \( t_1 \), so the algorithm infers a causal dependency between them and adds a place connecting transitions \( t_1 \) and \( t_2 \) in the output net (place \( pl \) in Figure 2). It is also possible to see that \( t_4 \rightarrow t_5 \) (\( t_4 \) precedes \( t_5 \)), \( t_4 \rightarrow t_6 \) (\( t_4 \) precedes \( t_6 \), and \( t_5 \neq t_6 \) do not happen in the same trace), thus the algorithm creates a place after \( t_4 \) that branches the execution (\( pl \) in the same Figure). Since the input traces contain only relevant URLs and each unique URL becomes a transition after process mining, the task-to-URL map is trivial to obtain.

**Data-flow properties.** Identification and annotation of data properties has been used initially in \[50\] and later in other works \[53\] [39\] [44\]. We use five annotations, namely constant, don't care, invariant, instance identifier, and user identifier, which are used for three goals. Constants and don't cares are used to restrict and generalize, respectively, the input traces by fixing or hiding given values that are used to match incoming requests at run-time. A user identifier is used to detect the user issuing a request and an instance identifier to detect the workflow instance that the request is related to. This is because several instances of the same workflow may be running at the same time and they may have different execution histories (e.g., an instance 1 of the invoicing process where only \( t_1 \) was executed by \( u_1 \) and an instance 2 where both \( t_1 \) was executed by \( u_1 \) and \( t_2 \) was executed by \( u_2 \)). Invariants indicate values that should not be modified during a workflow instance execution.

Data-flow properties are obtained by using differential analysis, i.e., comparing the differences in the data values between traces, as is done in related work (e.g., \[53\] [44\]). For each trace, the analysis compares the values of all parameters in each request in relation to (i) the same parameter in other requests of the same trace, (ii) the same parameter in other traces, (iii) other parameters in the same trace, and (iv) other parameter in other traces. Aegis does not apply syntactic annotation (as, e.g., \[44\]) to identify the data type of each parameter, and does not try to discover possible values or intervals for data elements, because it does not enforce particular values that were seen in the traces (except for constants). Below, we describe the differential analysis used to identify each kind of data-flow property.

Let \( WS \) be the set of traces \( \tau \) used for analysis, each \( \tau \) be composed of requests \( r_j \) and responses \( s_j \), and each request or response have a set \( P \) of parameters \((k,v)\). Considering the same request \( r_j \) in every trace \( \tau \in WS \), if a parameter \((k,v)\) appears in only a strict subset \( r' \subset \tau \) of the traces, it is considered optional and ignored, i.e. dropped from the URL in the labeling function \( L \). Constants are parameters that are present in every trace \( \tau \in WS \) for the same URL of a request \( r_j \) and whose key \( k \) and value \( v \) never change. An example is the parameter \text{action}\text{=create}, which is in \( t_1 \) of traces \( t_1, t_2 \), and \( t_3 \) in Example 1. Don't cares are parameters that appear in every trace \( \tau \in WS \) for the same URL of a request \( r_j \) and whose key \( k \) remains constant, but whose value \( v \) is different in at least one of the requests. One example is \text{prod}\text{=abc}, \text{prod}\text{=def} and \text{prod}\text{=ghi} in \( t_1 \) of Example 1 annotated as \text{prod}\text{=abc}. An instance identifier is a key \( k \) whose value \( v \) is present in every request \( r \) of a trace \( \tau \), with different \( v \)'s in every trace. In Example 1, the parameter \text{id} is an instance identifier, since it has the value 1 in every request of \( t_1 \), the value 2 in every request of \( t_2 \), and the value 3 in every request of \( t_3 \). Notice that what must remain constant is the value and not the key, so it is possible to have an instance identifier called, e.g., \text{id} in one request and \text{uid} in another request. A user identifier is a parameter that comes from a response issued by the server, is stored in a cookie, sent in every request of a trace and whose value changes in every trace in \( WS \). In Example 1, only URLs are shown in the traces, but the cookie \text{sid} is sent with every request, as can be seen towards the end of the example. Invariants are values \( v \) that remain constant during a trace, change between traces in \( WS \) and are not present in every request of a trace (as opposed to instance identifiers). Two examples are the \text{value} parameter in \( t_1 \) and \text{3} in Example 1 and the \text{token} in \( t_4 \) and \text{5} of Example 2. Like instance identifiers, invariant
values should not change, but their keys might, so that an invariant can be called, e.g., \texttt{price} in one request and \texttt{amount} in another. There may be many invariants in a workflow, so they are annotated as \texttt{<<1,1>>}, \texttt{<<1,2>>}, for run-time enforcement (there may be several don’t cares too, but they are not enforced and do not need separate annotations).

The result of Step 1 is a tuple \((PN, L)\), where \(PN\) is a Petri net obtained from process mining and \(L\) is a labeling function that associates to each transition in the net a URL annotated with the identified data properties. Although the inferred model \((PN, L)\) is obtained automatically, it can be edited by a user before being sent for monitor synthesis. Control-flow constraints can be changed by graphically adding or removing places or transitions in the Petri net (or tasks and gateways in BPMN), while data properties can be modified by adding or removing annotations on the URLs.

### 3.2 Step 2 - Monitor synthesis

Monitor synthesis takes as input the tuple \((PN, L)\) obtained from Step 1 and, optionally, augments it with security properties given by the user. As an example, the user can specify a set of authorization constraints \(\text{SoD}(tx, ty)\) indicating that tasks \(tx\) and \(ty\) must be executed by different users (the same goes for other constraints, such as BoD). The user must also indicate whether the monitor should enforce an authorization policy, which will be specified at run-time. A symbolic transition system is obtained from the augmented tuple and sent to a model checker, which computes a graph containing all valid executions of the workflow. The graph represents these executions compactly by using symbolic states, symbolic users, and sharing common paths.

### Security properties specification

All behaviors of the web application that satisfy the specified security properties (namely those deriving from authorization policies and constraints) are represented by the executions of a symbolic transition system \(S = (V, Tr)\), where \(V\) is a set of state variables and \(Tr\) is a set of transitions. In general, each workflow task corresponds to one transition. The enforced properties, and as a consequence the variables in \(V\), fall into four categories. Therefore, \(V\) can be seen as the union of four disjoint sets \(V_{CF}, V_{DF}, V_{C},\) and \(V_{A}\), explained below. First, control-flow constraints (involving state variables from the set \(V_{CF}\)) are automatically derived from the Petri net \(PN\) by using Boolean variables \(p_i.s\), one for each place in \(PN\), indicating the presence or absence of tokens in these places; and Boolean variables \(d_{i.t.s}\), one for each task, representing the fact that a task has been executed or not. Second, data values of parameters annotated as invariants are represented by variables \(v_i\) and \(g_i\) (\(g\) stands for ghost) in \(V_{DF}\). Data types are abstracted and every \(v_i\) and \(g_i\) is represented by an integer, since any type can be encoded as an integer. Third, the set \(V_{C}\) contains Boolean functions \(h_{i.t.s}\), one for each task, keeping track of which user has executed task \(t\). The functions start with a value False for every transition and user, which is updated after each task execution. Authorization constraints of the form \(\text{SoD}(tx, ty)\) are represented by an enabling condition \(-h_{u.t}(u)\) in transition \(ty\). BoD constraints can be encoded in a similar way as \(\text{SoD}\) and cardinality constraints can be specified by using a function \(\text{count}(u)\) that keeps track of the number of tasks executed by each user. Fourth, an authorization policy is represented by constraints on Boolean functions \(a_i.s\), one for each task, that involve state variables from the set \(V_{A}\) and return True iff a user is authorized to perform task \(t\). The functions \(a_i.s\) are an interface to the authorization policy that is provided at run-time. Although we do not detail them here, other security policies can be encoded in this framework, such as data-based access control and Chinese Wall by using separate authorization and history functions for data objects.

Transitions in \(Tr\) have the shape

\[
t(u) : \text{enCF} \land \text{enC} \land \text{enA} \rightarrow \text{actCF} | \text{actC} | \text{actA} | \text{actDF}
\]

where \(t(u)\) is an identifier, the \(en\)'s are predicates on the state variables in \(V\) representing the enabling conditions of the transitions (in terms of control-flow, constraints, and authorization policy, respectively), and the \(act\)'s are parallel (||) assignments to the variables in \(V\) representing the effects of executing a transition (again, for each security property). Data variables in \(V_{DF}\) are not used in the conditions, only in the assignments of transitions that contain data invariants, as \(g_i := v_i\). The values of \(v_i\) are taken as input at run-time. As an example, the transition for task \(t_2\) in Example 1 is

\[
t2(u) : p1 \land \neg d_{12} \land \neg h_{11}(u) \land a_{12}(u) \rightarrow p1, p2, d_{12}, h_{12}(u) := F, T, T, T
\]

indicating that, for this transition to be executed, there must be a token in \(p_1\), \(t_2\) should not have been executed (\(\neg d_{12}\)), the user \(u\) should not have executed \(t_1\) (\(\neg h_{11}(u)\)) and the same user \(u\) should be authorized to execute \(t_2\) (\(a_{12}(u)\)); the result of its execution is that a token is removed from \(p_1\), placed in \(p_2\) and the functions \(d_{12}\) and \(h_{12}\) are updated to record that \(t_2\) has been executed and user \(u\) has executed \(t_2\), respectively. Since \(t_2\) does not contain invariants, there is no assignment to data values. However, \(t_1\) contains \(g_1 := v_1\) in the update, where the value of \(v_1\) will be taken as the value of parameter \(value\) of the incoming request at run-time.

### Monitor synthesis

The transition system \(S\) is fed to a symbolic model checker, which computes a reachability graph \(RG\) representing all possible executions of the workflow by a set of symbolic users. A procedure \(MS\) to compute this graph is described in [6]. \(MS\) is based on backward reachability, starting from a goal formula describing the termination of the workflow and applying transitions until a fix-point is reached. \(RG\) is a directed graph whose edges are labeled by task-user pairs in which users are symbolically represented by variables (called user variables) and whose nodes are labeled by a symbolic representation (namely, a formula of first-order logic) of the set of states from which it is possible to reach a state in which the workflow successfully terminates. A Datalog program \(M\) (with negation) is derived from \(RG\) by generating a clause of the form \(\text{can}_n\_\text{do}(u, t) \leftarrow \beta_n\) for each node \(n\) in the graph. An invariant \(d_{i.t} \Rightarrow v_i = g_i\), for every \(v_i\) in the assignments of transition \(t\), is conjuncted with each clause \(\beta_n\). This invariant specifies that after the execution of each transition the value of a variable remains the same as the value of its respective ghost variable. \(M\) is then translated to SQL (aggregation-free SQL and non-recursive Datalog with negation are equivalent and the translation is straightforward [46]) and the SQL program is capable of answering—after being instantiated with a concrete authorization policy—user requests to execute tasks in a workflow in such a way that the authorization and execution constraints are not violated, the authorization policy is respected and termination of the workflow is guaranteed, thus enforcing the specified security

\[\text{in Section 3.2 token refers to Petri net tokens instead of security tokens}^{15}\]
properties and solving the run-time WSP.

The result of the monitor synthesis step is a tuple \((M, L)\), where \(M\) is the monitor generated from \(RG\) and \(L\) is the labeling function, which now maps from transitions in the system to annotated HTTP requests.

### 3.3 Step 3 - Run-time monitoring

Step 3 takes as input \((M, L)\) and, if previously specified, an authorization policy \(TA\) specifying which users can execute which tasks. The authorization policy is used to populate a database queried by \(M\), resulting in a concrete monitor.

A reverse proxy intercepts all incoming requests to the application and decides, for each request, whether it is part of a workflow or not. To do so, it tries to match the URL and parameters in the request to annotated URLs and parameters stored in \(L\), taking into account the constant, ignored and don’t care values. If there is no match, the proxy forwards the request to the application, as it is not part of any workflow.

If there is a match, the proxy associates the request to a task \(t\) of a workflow \(W(T, U)\) and checks the annotated URL for \(<<IID>>\) and \(<<UID>>\) values, extracting the instance \(i\) and the user \(u\). The user identifier is a cookie value that must be mapped to a user name in the policy. This is done by capturing login actions, storing the cookies issued to each user, and later retrieving the user names based on the cookie.

To enforce data invariants, when the proxy receives a request for the first URL containing the annotation \(inv=<<I_{i}>>\), it stores the value of the parameter \(inv\) as \(v_i\). When any subsequent task containing \(<<I_{i}>>\) is accessed, the value of the incoming annotated parameter \(inc\) is compared to the stored value \((v_i = inc)\). When any of the above conditions is not met, the proxy forwards or drops the request.

Finally, the proxy issues a request \(can_do(u, t)\) to the monitor of instance \(i\) of \(W\) and acts based on its response by either forwarding the request or dropping it.

## 4. EVALUATION

**Aegis** was implemented in Python 2.7. We capture execution traces as ZAP\(^{11}\) scripts exported by ZAP, extract data properties from them, aggregate them into a XES\(^{12}\) file and use ProM\(^{17}\) to output a PNML\(^{28}\) file containing the mined Petri net. The implementation of the monitor synthesis algorithm is the one from \([6]\), using the MCMT model checker\(^{24}\), which takes as input a PNML file and outputs the synthesized SQL monitor. We use mitmproxy\(^{19}\) and instantiate it with the generated monitor as in the example below:

```
mitmdump -R http://localhost:80 -p8080 -s httpmonitor.py
```

where `mitmdump -R` starts the proxy in the reverse mode to accept requests on port 8080, process them using the `httpmonitor.py` script and, possibly, forward them to the application (http://localhost:80). The `httpmonitor.py` script intercepts requests and responses using a proxy API, performs the URL matching, queries a MySQL database (where the authorization policies are stored) by using the synthesized queries, and either forwards or drops the request. The proxy also supports HTTPS connections.

### 4.1 Experimental setup

We tested **Aegis** on popular open-source applications (shown in Table 1), synthesizing monitors in the configurations \((P, C, I)\) and \((I)\). Applications 1-4 are ERP platforms, 5-6 are e-health applications and 7-10 are e-commerce applications. Column Application contains the name of each application; Language shows the language in which it was developed; Params describes the predominant method used for parameter passing (although an application can use several methods) and Downloads reports the number of downloads (applications 1-6) or public installations (applications 7-10).

The different languages show the versatility of the black-box approach, which has to be tailored to support each parameter passing method (to annotate and match URLs). Supporting new applications that use the same method is straightforward, whereas supporting new methods (e.g., ODData\(^{40}\)) requires new functionality for model inference.

The number of downloads and installations is a measure of the popularity of the applications and it comes from official repositories (applications 2, 3, 5, and 6), data in the web page of the project (applications 1 and 4), or related work\(^{40}\) (applications 7-10). The number of actual deployments for applications 1-6 is not available as they are usually internal to an organization and not indexed by search engines. The numbers shown for applications 7-10 were obtained using Google dorks and are from 2014\(^{39}\).

We pre-configured the applications using demo data (e.g., financial accounts for ERP, products for e-commerce, patients for e-health) either available during installation or generated by us. We then captured four execution traces for each workflow (as in \([53]\)) and two login traces for each application.

To compare **Aegis** in different ERP applications, we used workflows offered by all of them: Purchase order (PO), Sales order (SO), Purchase invoice (PI), and Sales invoice (SI). They are slightly different in each application, varying from 4 to 6 tasks, usually with a gateway defining 2 to 3 alternative execution branches. Figure 4 shows at the top the patient visit workflow mined from OpenEMR (where the lines labeled by = represent BoD constraints added by us). The same Figure shows at the bottom the lab analysis workflow mined from BikaLIMS. In these 6 applications, we added the authorization constraints and specified policies with 10 users assigned to each task, generating \((P, C, I)\) monitors. The number of users was arbitrarily chosen because it influences the time taken to answer queries (discussed in Section 4.2).

<table>
<thead>
<tr>
<th>Application</th>
<th>Language</th>
<th>Params</th>
<th>Downloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odoo</td>
<td>Python</td>
<td>JSON</td>
<td>2M</td>
</tr>
<tr>
<td>Dolibarr</td>
<td>PHP</td>
<td>GET</td>
<td>850k</td>
</tr>
<tr>
<td>WebERP</td>
<td>PHP</td>
<td>GET</td>
<td>617k</td>
</tr>
<tr>
<td>ERPNext</td>
<td>Python</td>
<td>JSON</td>
<td>25k</td>
</tr>
<tr>
<td>OpenEMR</td>
<td>PHP</td>
<td>GET</td>
<td>382k</td>
</tr>
<tr>
<td>BikaLIMS</td>
<td>Python</td>
<td>REST</td>
<td>111k</td>
</tr>
<tr>
<td>OpenCart 1.5.3.1</td>
<td>PHP</td>
<td>GET</td>
<td>9M</td>
</tr>
<tr>
<td>TomatoCart 1.1.7</td>
<td>PHP</td>
<td>GET</td>
<td>119k</td>
</tr>
<tr>
<td>osCommerce 2.3.1</td>
<td>PHP</td>
<td>GET</td>
<td>80k</td>
</tr>
<tr>
<td>AbanteCart 1.0.4</td>
<td>PHP</td>
<td>GET</td>
<td>21k</td>
</tr>
</tbody>
</table>
The workflows for e-commerce applications are similar to the one shown in Figure 4. For these applications, we use the \((Z)\) configuration, thus neither constraints nor authorization policies were defined. Applications 7 and 8 have a vulnerability allowing attackers to shop for free due to improper validation of PayPal Express Checkout tokens, which can be replayed from previous transactions, as explained in Section 2.2 (CVE-2012-4934 for TomatoCart). Applications 9 and 10 allow an attacker to buy products and pay to himself, by tampering with a parameter that indicates who should receive the payment for a PayPal Payments Standard transaction (CVE-2012-2991 for osCommerce).

All applications were deployed as Docker containers and the tests as Selenium scripts, using the architecture described in [13], which allows us to achieve repeatable experiments by automatically testing the applications in five steps: (i) start a new container with the application; (ii) run the workflow in the Selenium script without monitoring; (iii) start the monitor; (iv) run the workflow with monitoring; (v) capture results and destroy the container. The experiments ran on a MacBook 2014 laptop with a 1.3GHz dual-core Intel Core i5 processor and 8GB of RAM.

### Results

The enforcement of security properties and mitigation of vulnerabilities was successful in all applications, which was confirmed by manual inspection. In applications 1-6, we tested the enforcement of policies and constraints by trying the attacks described in Section 2.2 (workflow bypass, workflow violations, and authorization violations). The monitor was able to block situations such as the same user executing an entire workflow (SoD violation), and users trying to access tasks that were not assigned to them. In applications 7-10, we tried to exploit the vulnerabilities described above. In applications 7-8, the attacks were unsuccessful because \(\text{token}\) was detected as an invariant and automatically enforced. In applications 9-10, the \(\text{PayeeId}\) parameter was detected as a constant, since every trace in the input was related to the same shop. Constant values are usually not enforced, only used to match URLs (details in Section 3). For applications 9-10, we edited the inferred model by annotating \(\text{PayeeId}\) with \(<\text{I}>>\), so that requests with any value of \(\text{PayeeId}\) are controlled by the monitor, and used invariant enforcement with a constant, instead of with the first received value, to check that in every request containing \(\text{PayeeId}\), its value is equal to \(\text{ShopId}\) (the constant obtained in the traces). Manual editing could be avoided by doing inference from a dataset containing execution traces of different shops.

We measured the overhead of the monitors in terms of model inference and monitor synthesis and by comparing the execution of each workflow with and without monitoring. Each execution was repeated 10 times and Table 2 shows the results. Column \(\text{Appl.}\) shows the application under test (and the specific workflow tested for ERP applications); \(\text{Synth.}\) shows the median time to infer a model from the captured traces and synthesize a monitor for each workflow. \(\text{Original}\) reports the median time between receiving a request and sending a response with no monitor (measured by mitmpbxy without the httpmonitor.py script); \(\text{Query}\) reports the median time for the monitor to answer to a query (ignoring the time taken by the proxy to invoke the script, translate an incoming request to a monitor query, forward the request, etc); \(\text{Aegis}\) reports the median time of a response with the monitor script (the time taken by the application, plus the translation time, plus the querying time); and \(\text{Overhead}\) shows the overhead incurred by the use of the monitor as seen by a user (the difference between \(\text{Aegis}\) and \(\text{Original}\)).

The time in column \(\text{Query}\) varies with the size of a workflow and the number of users and constraints, as reported in [6], which describes a linear growth due to the \(\log\) space complexity of the queries used. The time in column \(\text{Aegis}\) adds, to the time in \(\text{Query}\), the time to process and match URLs, which depends on the data structures used. As shown in Table 2, the overhead varied from 8ms to 84ms, with a median of 13.5ms, out of which less than 10ms in most cases is spent in querying the monitor. The overhead variability is due to the complexity of the workflows and the time taken to translate a request to a monitor. For instance, applications 1 and 4 have a large overhead because of the time to flatten JSON requests. Monitor synthesis is computationally much more expensive, but it is run only once for each workflow (unless there is a change in the application).

### Table 2: Monitoring overhead

<table>
<thead>
<tr>
<th>#</th>
<th>App.</th>
<th>Synth.</th>
<th>Original</th>
<th>Query</th>
<th>Aegis</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Odoo</td>
<td>21.3 s</td>
<td>112 ms</td>
<td>6 ms</td>
<td>132 ms</td>
<td>20 ms</td>
</tr>
<tr>
<td>2</td>
<td>Dolibarr</td>
<td>14.3 s</td>
<td>93 ms</td>
<td>5 ms</td>
<td>103 ms</td>
<td>10 ms</td>
</tr>
<tr>
<td>3</td>
<td>WebERP</td>
<td>21.1 s</td>
<td>50 ms</td>
<td>5 ms</td>
<td>104 ms</td>
<td>12 ms</td>
</tr>
<tr>
<td>4</td>
<td>ERPNext</td>
<td>15.9 s</td>
<td>263 ms</td>
<td>10 ms</td>
<td>327 ms</td>
<td>64 ms</td>
</tr>
<tr>
<td>5</td>
<td>BikaLIMS</td>
<td>19.5 s</td>
<td>32 ms</td>
<td>4 ms</td>
<td>105 ms</td>
<td>15 ms</td>
</tr>
</tbody>
</table>

The workflows for e-commerce applications are similar to the one shown in Figure 4. For these applications, we use the \((Z)\) configuration, thus neither constraints nor authorization policies were defined. Applications 7 and 8 have a vulnerability allowing attackers to shop for free due to improper validation of PayPal Express Checkout tokens, which can be replayed from previous transactions, as explained in Section 2.2 (CVE-2012-4934 for TomatoCart). Applications 9 and 10 allow an attacker to buy products and pay to himself, by tampering with a parameter that indicates who should receive the payment for a PayPal Payments Standard transaction (CVE-2012-2991 for osCommerce).

All applications were deployed as Docker containers and the tests as Selenium scripts, using the architecture described in [13], which allows us to achieve repeatable experiments by automatically testing the applications in five steps: (i) start a new container with the application; (ii) run the workflow in the Selenium script without monitoring; (iii) start the monitor; (iv) run the workflow with monitoring; (v) capture results and destroy the container. The experiments ran on a MacBook 2014 laptop with a 1.3GHz dual-core Intel Core i5 processor and 8GB of RAM.

### 4.2 Results

The enforcement of security properties and mitigation of vulnerabilities was successful in all applications, which was confirmed by manual inspection. In applications 1-6, we tested the enforcement of policies and constraints by trying the attacks described in Section 2.2 (workflow bypass, workflow violations, and authorization violations). The monitor was able to block situations such as the same user executing an entire workflow (SoD violation), and users trying to access tasks that were not assigned to them. In applications 7-10, we tried to exploit the vulnerabilities described above. In applications 7-8, the attacks were unsuccessful because \(\text{token}\) was detected as an invariant and automatically enforced. In applications 9-10, the \(\text{PayeeId}\) parameter was detected as a constant, since every trace in the input was related to the same shop. Constant values are usually not enforced, only used to match URLs (details in Section 3). For applications 9-10, we edited the inferred model by annotating \(\text{PayeeId}\) with \(<\text{I}>>\), so that requests with any value of \(\text{PayeeId}\) are controlled by the monitor, and used invariant enforcement with a constant, instead of with the first received value, to check that in every request containing \(\text{PayeeId}\), its value is equal to \(\text{ShopId}\) (the constant obtained in the traces). Manual editing could be avoided by doing inference from a dataset containing execution traces of different shops.

We measured the overhead of the monitors in terms of model inference and monitor synthesis and by comparing the execution of each workflow with and without monitoring. Each execution was repeated 10 times and Table 2 shows the results. Column \(\text{Appl.}\) shows the application under test (and the specific workflow tested for ERP applications); \(\text{Synth.}\) shows the median time to infer a model from the captured traces and synthesize a monitor for each workflow. \(\text{Original}\) reports the median time between receiving a request and sending a response with no monitor (measured by mitmpbxy without the httpmonitor.py script); \(\text{Query}\) reports the median time for the monitor to answer to a query (ignoring the time taken by the proxy to invoke the script, translate an incoming request to a monitor query, forward the request, etc); \(\text{Aegis}\) reports the median time of a response with the monitor script (the time taken by the application, plus the translation time, plus the querying time); and \(\text{Overhead}\) shows the overhead incurred by the use of the monitor as seen by a user (the difference between \(\text{Aegis}\) and \(\text{Original}\)).

The time in column \(\text{Query}\) varies with the size of a workflow and the number of users and constraints, as reported in [6], which describes a linear growth due to the \(\log\) space complexity of the queries used. The time in column \(\text{Aegis}\) adds, to the time in \(\text{Query}\), the time to process and match URLs, which depends on the data structures used. As shown in Table 2, the overhead varied from 8ms to 84ms, with a median of 13.5ms, out of which less than 10ms in most cases is spent in querying the monitor. The overhead variability is due to the complexity of the workflows and the time taken to translate a request to a monitor. For instance, applications 1 and 4 have a large overhead because of the time to flatten JSON requests. Monitor synthesis is computationally much more expensive, but it is run only once for each workflow (unless there is a change in the application).
A modular approach allows the scalability of the synthesis procedure, which was tested with workflows of up to 500 tasks and returned a monitor in less than 10 minutes. We did not test the performance of monitoring concurrent executions of workflows. Since there is no interaction between instances, we believe that any additional overhead would be related to request processing in the proxy and database access to answer monitor queries.

5. DISCUSSION AND LIMITATIONS

Model inference. There are solutions to design web applications as workflows, as well as frameworks that allow their declarative description. However, model-driven development of web applications is not common. This highlights the need for model inference, which can be static or dynamic.

Authorization. Aegis provides an easy way to enforce authorization policies and constraints on the actions a user can trigger when interacting with a service (via URL requests). Obtaining the same behavior within an application is not trivial. In fact, it must be done differently for each application and the granularity of the permissions therein offered may not be easily related to URL requests. As an example, the granularity of permissions in the applications we experimented with varied from actions in a module (e.g., creating an invoice in the finance module of Dolibarr) to binary module access (i.e., users that can access a module can perform all of its actions). Though the former provides permissions over actions, maybe not all actions in the process are covered, e.g., Send by email of Figure 2. In the latter, it would not be possible to create SoD constraints between actions within the same module. None of the applications we tested supports authorization based on individual URLs nor authorization constraints. Moreover, the policy enforced by Aegis can be applied on top of the existing one (if any) and can be easily specified by connecting tasks (obtained from the HTTP traces) to users (obtained from the application).

Other approaches. Some properties enforced by Aegis can be achieved with other tools, e.g., Web Application Firewalls (WAFs), which filter incoming requests based on user-defined rules, or Run-time Application Self-Protection. Although the use of WAFs is a best practice, they are typically incapable of handling logical flaws. Indeed, most tools protect against injection vulnerabilities rather than business logic flaws and a defense-in-depth approach, combining multiple tools, is recommended.

We are unaware of any single tool that encompasses all the properties enforced by Aegis (related run-time enforcement approaches are discussed in the next section). Security mechanisms can be coded directly in the application, but this is error-prone, unscaleable, and difficult for legacy applications.

Usability. We have not tested Aegis with end-users, but we believe the learning curve is not steep, since most steps are automated. The manual steps—trace capturing, (optional) model editing, and (optional) authorization policy definition—are facilitated by the use of well-known tools for test automation and business process editing.

Limitations. Aegis only sees the traffic between users and the target application, ignoring messages between a third party and a user, and between the application and a third party. Model inference ignores some message exchange formats, e.g., XML. The first limitation is architectural; the second is an implementation issue. It is possible to have a parser for each format that returns \((k, v)\) pairs and searches for them in incoming requests. The invariants that Aegis detects and enforces are only exact matches, which is the most common case in web applications.

We did not see false positives (i.e., blocking valid requests) during our evaluation, because we inferred models from a diverse dataset and tested them to ensure they captured all the executions we foresaw. However, Aegis is not immune to false positives caused by poorly inferred models that do not match all executions of an application. Therefore, the user may not want Aegis to block incoming requests, which could prevent legal executions. Instead, it can be used for soft enforcement, where denied requests represent deviations that must be logged so that a human agent can later examine them.

Aegis synthesizes monitors that work in isolation, disregarding any possible inter-workflow and inter-instance dependencies and constraints. Related works consider such constraints when executing applications in several tabs.

6. CONCLUSION

We have described, implemented, and evaluated Aegis, a technique and tool to enforce authorization policies and constraints, control- and data-flow integrity and ensure the satisfiability of web applications. We have tested our implementation with relevant open-source applications. The experiments clearly show the validity of our approach in enforcing the desired properties and mitigating related vulnerabilities. The performance results show an acceptable overhead incurred at run-time.

Related work. There are many works related to the enforcement of authorization, control- and data-flow integrity (separately) in web applications, and mitigation of related vulnerabilities, such as missing authorization checks, workflow and state violations, logic vulnerabilities, forceful browsing, and parameter tampering. These approaches are white-box, whereas Aegis is completely black-box. A black-box approach to block request forgery was described in [31], but it ignores data-flow and authorization. Black-box enforcement of control- and data-flow integrity has been done in Ghostrail by dynamically replicating on the server-side valid user clicks, form entries, links, and parameters. Ghostrail does not consider authorization, and needs a fresh replica for each user session, which is not scalable.

Process mining has been used to optimize user interaction, but we are unaware of any work using it to enforce security in web applications. Web application workflow models have been used to detect anomalous user behavior and to find logic vulnerabilities by capturing execution traces, identifying behavioral patterns, and generating test cases. Aegis is focused on enforcement, so these techniques are complementary. It is possible to, e.g., find a vulnerability using and mitigate it using Aegis (this is what was done with the e-commerce vulnerabilities). The enforcement of authorization constraints for collaborative web applications was studied in. The authors considered different application domains than us, there was no discussion about workflow satisfiability, and their evaluation was limited to
The closest related works are BLOCK [22] and Inte-Guard [53]. Both use a reverse proxy, construct control- and flow-policies using invariants detected from network traces, and rely on manual identification of critical requests. BLOCK also extracts invariants from session information in PHP applications. Inte-Guard is tailored for multi-party application integration, where most steps are automatic (not human tasks) and workflows must be executed from beginning to end in one shot. Neither tool enforces authorization policies nor constraints. FlowWatcher [37] uses a similar approach of external monitoring, but enforces only authorization policies specified in a domain specific language.

**Future work.** We intend to test Aegis in more real-world applications, e.g., those in [53, 44], and to extend it with code analysis to measure the coverage of inferred models by executing traces and checking followed paths. We would also like to explore monitor inlining [23], which requires source code changes to embed the monitor into the application.

7. REFERENCES


