Security Threat Identification and Testing

Roberto Carbone  
Security & Trust, FBK-Irst  
Trento, Italy  
carbone@fbk.eu

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

Annibale PanicHELLa  
Delft University of Technology  
Delft, The Netherlands  
a.panicHELLa@tudelft.nl

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

Abstract—Business applications are more and more collaborative (cross-domains, cross-devices, service composition). Security shall focus on the overall application scenario including the interplay between its entities/devices/services, not only on the isolated systems within it. In this paper we propose the Security Threat Identification And TEsting (STIATE) toolkit to support development teams toward security assessment of their under-development applications focusing on subtle security logic flaws that may go undetected by using current industrial technology. At design-time, STIATE supports the development teams toward threat modeling and analysis by identifying automatically potential threats (via model checking and mutation techniques) on top of sequence diagrams enriched with security annotations (including WHAT-IF conditions). At run-time, STIATE supports the development teams toward testing by exploiting the identified threats to automatically generate and execute test cases on the up and running application. We demonstrate the usage of the STIATE toolkit on an application scenario employing the SAML Single Sign-On multi-party protocol, a well-known industrial security standard largely studied in previous literature.

I. INTRODUCTION

Nowadays business applications are more and more complex and collaborative. Consumers may use multiple devices to access remote business applications whose delivered resources are often the results of a complex business logic executed among multiple organizations and service providers. These business applications are clearly crossing trusted domains, making use of multiple heterogeneous devices and services. Security shall focus on the overall application scenario including the interplay between its entities/devices/services, not only on the isolated systems within it. For instance, having an online shopping store interacting with a highly secure online payment system via a flawed protocol makes the entire application scenario trivially vulnerable.

Literature does not lack examples of serious logic flaws arising from the protocols underlying the interplay between application entities (e.g., [1], [2], [3]). Mistakes embrace both the design and development phases, during which the development team (DevT, in short) strives with the resources at its disposal (people, tools, etc) to meet the requirements posed by the application scenario and may overlook the security implications associated with the choice of some message elements, transport layer security, etc. Needless to say, without tool support, such an activity remains a time consuming and error-prone one.

At design-time, DevTs should be supported toward the security evaluation of the choices at their disposal so to identify also threats emerging from the interplay of the entities involved in the business application. For instance, DevTs may be interested in ensuring that certain security properties are guaranteed, e.g., confidentiality of the delivered resources, and assessing WHAT-IF conditions, e.g., WHAT-IF a HTTP message contains or not a specific field (e.g., a token). At run-time it is then important to support the DevTs with, possibly automated, techniques to test whether the identified threats are indeed exploitable.

Industrial approaches for identifying threats are often based on Threat Modeling methodologies, e.g., STRIDE [4]. Drawing an architecture diagram is common practice in running threat modelling as it allows for brainstorming in a structured manner on potential threats. However the architecture diagram captures a static view of the business application so that subtle security logic flaws—whose root causes lie in the interaction between the various entities and underlying protocols—can go undetected. In addition, no automation is offered to test the identified threats.

Approaches based on formal methods have been proposed to detect logic-flaws at design time (e.g., [2], [5]). However, despite their successful application to real and complex protocols in academic and research world, they all struggle in finding their way into industrial practice. One of the main obstacles lies in the provisioning of the formal specifications needed to trigger these techniques. Indeed, identifying threats via formal analysis starts with the specification of a formal model capturing the protocol underlying the application scenario. Variants of this formal model can then be derived so to capture any relevant combination of the available design and development choices e.g., the WHAT-IF conditions. Two main problem statements emerge here: (i) DevTs are not used to write formal models and this activity would be too costly; and (ii) writing and managing all these inter-related formal models can be complex and tedious.

Even though formal analysis is used for identifying threats, the formal specifications used to this aim can be significantly different from real protocol implementations written in programming languages because some security assumptions made at design level may not hold in the implementation [6]. For example, when writing the formal specification the expert may assume that the messages exchanged between two entities travel on the same TLS channel, while in the real environment this may be not the case. While some approaches based on formal methods have been extended toward automated testcase generation and execution (e.g., [7], [8]), bridging the gap between the abstract and the real world is still requiring a lot of manual effort from the DevT. All the problems reported above negatively impact the cost/benefit ratio of a potential usage of formal methods in an industrial setting.
In this paper we present STIATE, a toolkit\(^1\) that mitigates these problems so to support security experts within DevTs toward the security assessment of the protocols underlying their business applications with the ultimate goal of (i) determining safe and unsafe choices as well as (ii) identifying logic-flaws that, otherwise, can go undetected. Specifically, STIATE provides:

- **(usability)** a front-end for specifying security-relevant aspects of the protocols (including WHAT-IF conditions) on top of sequence diagrams, and guiding the security expert during the design process\(^2\);
- **(threat identification)** a translator for the automatic generation of formal specifications, written in the ASLan++ [14] formal language, starting from options and choices made by experts during the design process;
- **(threat identification)** the formal analysis of the generated formal specifications by using SATMC, a SAT-based bounded Model Checker that has been successfully used to discover serious flaws in security protocols and services [7], [9];
- **(threat identification)** a mutation engine allowing the analysis of the real implementation of security protocols that might deviate from its formal (abstract) model defined at design level; and
- **(testing)** a semi-automated approach for the concretisation of abstract attack traces (threats) obtained by formal analysis in real test cases.

In doing so, STIATE borrows software components developed for the SPaCiOS Tool [10], [11] that are further improved with new usability features (e.g., the STIATE front-end), richer content for the mutation engine, and reduced manual effort for testing execution. We demonstrate STIATE against an application scenario employing the SAML Single Sign-On multi-party protocol, a well-known industrial security standard largely studied in previous literature. STIATE can be used against any multi-party application built on top of HTTP. For instance, it has been also applied on few internal use cases at SAP and on a proprietary solution for mobile e-Payment in Italy.

The paper is structured as follows. Section II describes the motivating example, used in the remainder of the paper to highlighting the main functionality and the practical usage of the STIATE toolkit. Section III describes the main architecture of the toolkit, while Section IV and Section V provide a deeper overview of its two main components: (i) the STIATE front-end and (ii) the STIATE back-end. Section VI concludes the paper.

### II. Motivating Example

To demonstrate the usage of the STIATE toolkit, we consider the situation in which a DevT wants to enrich its application with a Single Sign-On feature, by integrating the SAML 2.0 Web Browser Single Sign-On Profile [12] (SAML-SSO, for short). The corresponding high level sequence diagram is depicted in Figure 1. Three entities take part in the protocol: a client, an identity provider, and a service provider.

![Sequence Diagram of SAML-SSO](http://st.fbk.eu/stiate)

In [13], the authors reported a severe vulnerability of the SAML-based Single Sign-On for Google Apps, probably due to a wrong design choice, and/or too strong assumptions on the trustworthiness of SP. Indeed, that solution just omitted the SP and ID elements from the assertion issued by IdP. This led to a replay attack, allowing a compromised SP to impersonate a user at another SP.

What if while implementing the IdP entity the DevT opts for omitting a few elements from the assertion? Are the assumptions mentioned among the key points above all valid, both at design and at implementation level?

Answers to these questions come from previous literature [13], [2], [3], where thorough analysis have been done for SAML-SSO and serious vulnerabilities detected, due to both wrong design choices and assumptions, and implementations issues.

In [13], the authors reported a severe vulnerability of the SAML-based Single Sign-On for Google Apps, probably due to a wrong design choice, and/or too strong assumptions on the trustworthiness of SP. Indeed, that solution just omitted the SP and ID elements from the assertion issued by IdP. This led to a replay attack, allowing a compromised SP to impersonate a user at another SP.
In [2], the authors discussed the impossibility to guarantee an at-first-sight reasonable assumption on the communication channels, namely that it cannot be guaranteed that all the messages exchanged between SP and C travel over the same channels. A consequent authentication flaw was reported, which could be used as a launching pad for cross-Site Request Forgery and cross-Site Scripting attacks against e.g., the SAML-based SSO for Google Apps and the SSO available in Novell Access Manager v.3.1.

In the rest of the paper, we will use the SAML-SSO as a running example to illustrate how the STIATE toolkit, with its WHAT-IF conditions and mutation engine, could have helped to spot and prevent the issues reported above.

III. ARCHITECTURE AND IMPLEMENTATION

Figure 2 outlines the high level architecture of the STIATE toolkit. The security expert uses off-the-shelf commercial products such as SAP PowerDesigner (industry leading modeling and enterprise architecture tool) and Eclipse to interact with the two main components (dark color) of the STIATE toolkit: (i) the STIATE front-end, supporting the security expert during the security annotation process; and (ii) the STIATE back-end, for the automatic identification of threats (via model checking and mutation techniques) and the corresponding generation and execution of test cases against the System Under Testing (SUT).

The STIATE front-end provides a UI wizard to guide the expert in enriching UML Sequence Diagrams (SD) with security annotations. It is realized as an add-in for Enterprise Modelling Tools. Sequence diagrams enriched with security annotations can be exported as XMI files (XML Metadata Interchange) that is the interchange format for UML models. For our prototype, we implemented the front-end in C# and integrated it as DLL add-in for the SAP PowerDesigner product.

The STIATE back-end provides several functionalities. Firstly, the XMI file obtained as output by the front-end is translated by the XMI translator into formal specifications which are in turn analysed by means of the SATMC model checker. SATMC systematically explores all possible behaviours of the system and identifies potential attack traces. The Testing generation and execution module recasts the potential attack traces into test cases which, upon provision of testing data, are automatically executed against the SUT. Last, but not least, the Mutation Engine mutates the formal specification so to capture, e.g., the most common security errors in web applications. If the mutated models reveal potential flaws (according to model checker), the corresponding attack traces can be turned into test cases to prove the presence of a real flaw at the implementation level. If an attack trace can be reproduced on the implementation, then the STIATE back-end found a real security flaw. We implemented the STIATE back-end as a collection of extension points (plugins) to the standard Eclipse platform, one of the most used Implementation Development Environment overall.

A detailed description of the main components of the STIATE toolkit is reported in Sections IV and V.

IV. STIATE FRONT-END

The STIATE front-end is composed of a UI wizard for enriching UML sequence diagrams with security annotations, minimizing—whenever possible—the input required from security experts. Three categories of security annotations are supported:

- **Message annotation**: defining the detailed content for the messages exchanged between agents;
- **Entity annotation**: defining the knowledge of the agents defined in the sequence diagram;
- **Environment annotation**: defining the context of the protocol including its security goals and scenarios of execution.

The security expert provides message and environment annotations by means of UIs, while entity annotations are automatically derived. It is important to notice that annotations are typed. In the following we describe the different security annotations.

**Message Security Annotation.** Message security annotations define the message content. A message is represented as a tree where the root is always of type message, each node can be either a constant, a variable or a function. Node leaves can only be variables or constants. An example of message security annotation is shown in Figure 3 (more details are provided in Example 1). To support the security expert, STIATE provides an extensible catalog of functions (represented as a class diagram) that can be used for the definition of the message content. The current version supports—among others—the HTTP and SAML-SSO protocols. Figure 4 shows an excerpt of the catalog for function `httpResponse(Code, Location, Body)` that is used as root in Figure 3. The type of the function
above is given by the parent class, i.e., message, while its attributes define the types of the function arguments, i.e., Code of type code, Location of type res_header, and Body of type hbody.

Example 1. Figure 3 shows the message security annotations for message (4) of SAML-SSO (see Figure 1). The message is an HTTP Response with code 200, empty location (nil value), and an HTTP body containing a POST redirect form that, in turn, includes the assertion `authResponse(SP,IdP,C,ID)` (cf. KP1 of Section II).

Each function used in a message can be specified by means of dedicated UIs, e.g., Figure 5.

Example 2. The authentication assertion KP1 used in message (4), `authResponse(SP,IdP,C,ID)`, can be specified with the UI of Figure 5. Such UI allows one to select the desired value for each argument of the authentication assertion. The structure of the `authResponse` function is known as it is defined within the catalog. As a result, also the type of each argument is provided. Thereby, the security expert is guided in the definition, as only values of the required type can be selected in the drop-down box.

The security expert can define each function argument (corresponding to a leaf of the message tree) as follows:

- optional: to express a WHAT-IF condition;
- freshness: to state that it has to be freshly generated before sending it;
- To be learnt: to state that it has to be learnt by the receiver of the message.

It is worth noticing that the UI also allows one to sign and/or encrypt parts of the message.

Example 3. As introduced in Section II, a severe vulnerability found in SAML-SSO was due to the fact that the service provider information was not included in the authentication assertion KP1. By using the STIATE front-end the security expert can use the WHAT-IF condition to assess the security implications of such choice. This can be done as shown in Figure 5 where SP is marked as Optional. As a result, the analysis will consider both the case where such value is contained in the message and it is not.

To complete the definition of a message the security expert has to define the properties of the channel over which the message is sent. Channel properties can be defined in the bottom part of the UI (see Figure 3) as: Insecure, i.e., to model a channel without any protection, Unilateral, to model a run of TLS in which only one entity has a valid certificate (e.g. the server), or Bilateral to model a run of TLS in which the two entities have a valid certificate with each other. More than one option can be selected to consider the WHAT-IF analysis for different channel alternatives.

Example 4. In SAML-SSO, according to KP2, the channel between C and SP can be specified as unilateral, whereas the one between C and IdP can be defined as bilateral (as shown in Figure 3).

Entity Security Annotation. Entity security annotations define the knowledge of the entity, e.g., what is known by the entity since the beginning of the protocol execution and what is freshly created or learnt at a certain protocol step. More in detail, each variable part of the entity knowledge can be:

- Known. A variable is known if it is part of the initial knowledge of an entity.
- Fresh. A variable is fresh if it is created by the entity during the protocol execution.
Learnt. A variable is learnt if it is received and learnt by an entity as result of a received message.

The fact that a variable is known, learnt or fresh is derived from the way it is used in the messages. As a result, the security expert does not have to explicitly provide any entity security annotation, but she can use this information to have a precise overview of the data relevant for each entity.

Environment Security Annotation. Environment security annotations define the security goals the protocols should enjoy and the scenarios of execution, i.e., the number of sessions and initial knowledge of the agents.

The current version of the STIATE front-end supports authentication and secrecy goals. As a result, the threats that can be discovered are those endangering at least one of those properties. In the case of authentication goals, the modeler has to define the agent to be authenticated, the one performing the authentication, and the object on which the authentication is performed.

Example 5. Figure 6 shows the STIATE front-end for the authentication goal KP4 described in Section II where C has to be authenticated by SP over the object URI.

Scenarios define the execution context of the protocol in terms of number of sessions (two by default) and entity properties: is trusted denotes that the agent cannot be played by the intruder whereas unique is used to indicate that all sessions have to be played by the same agent (see Figure 7). The STIATE front-end automatically computes all scenarios encompassing such inputs and the modeler can select the desired ones or manually create others. For each selected scenario, the initial values of all variables of type Known have to be provided. The UI in Figure 8 supports such activity. The modeler can provide such values manually or use either the Invariant or Unique options to set, respectively, the same or a unique value for each session. In the figure the same, invariant value, data, is set.

Example 6. The trustworthiness assumption KP3 can be expressed as depicted in Figure 7 and leads to the alternatives of Figure 9. The security expert can simply select the first scenario in Figure 9 that involves two sessions. Both of them involve a client (c), and an honest IdP (idp). In the first session the SP is honest (sp), while in the second one the SP is malicious (i).

Though not visible in the motivating example, the STIATE front-end supports more advanced objects/features such as tables (e.g., SP may have a table of already consumed assertions) and entity actions (e.g., before granting the resource SP may check in its table that the assertion has not been already spent). All security annotations are encoded in XML and are included in the XMI representation of the UML Sequence Diagram that can be exported by SAP PowerDesigner.

V. STIATE BACK-END

The STIATE back-end is a collection of Eclipse plugins that provides a plethora of functionalities to support developers during the validation and testing processes: (i) allowing the formal analysis; (ii) supporting the generation and the execution of test cases; (iii) providing a deeper validation analysis via the mutation engine. Note that the concretisation and the execution of the test cases have been implemented by re-using components of the testing execution engine extracted from the SPaCiOS project [10].
**XMI Translation.** The output of the STIATE front-end is an XMI representation of the security protocol in terms of UML sequence diagram enriched with security annotations. Though it clearly defines the semantics of the protocol, it is not suitable for model checking analysis. The STIATE back-end is based on the SATMC model checker, which takes as input an ASLan (AVANTSSAR Specification Language) [14] specification. The translation is performed in two steps: the XMI files are first translated in formal ASLan++ models and then to lower-level ASLan models.

The key part of the translation is the first step from XMI to ASLan++. A single XMI representation contains all the annotations and the details specified by the security expert using the front-end, including: (i) optional arguments in a message (WHAT-IF conditions); (ii) multiple channel properties (e.g., a channel with both the insecure and unilateral properties set); (iii) multiple scenarios (each one including a set of sessions). All these elements represent possible alternatives for the same protocols, e.g., each WHAT-IF condition mirrors two different situations according to whether the optional field is contained in the message or not. In order to probe these alternatives one by one, the translator captures them in separate ASLan++ models. Figure 10 shows the pseudo-code of the algorithm that the translator uses to compute the number of ASLan++ models to be created. Note that, the optionality of an argument of a message does not only impact the message itself but the subsequent messages as well, i.e., when the optional message is not sent, then the receiver cannot use it in later messages (if not received via another message or already known). The generation of the ASLan++ and ASLan models is realized by relying on the ASLan++ Java library developed in the EU project AVANTSSAR [14].

**Example 7.** As shown in Example 3, SP is marked as optional in message (4) of SAML-SSO. As a result, the XMI translator generates two ASLan++ models: (i) the first version includes the SP field in the message (4) SAML Authn Response; (ii) the second version does not include SP in the same message. These two ASLan++ files are then translated in two corresponding lower-level ASLan models.

**Formal analysis.** Once obtained the proper input files, the SAT-based bounded model checker SATMC [9] is used to check whether the protocols meet the expected security goals. Given a formal specification (in ASLan) of a protocol and a security goal, SATMC reduces the problem of determining whether the protocol violates the security goal in $k > 0$ steps to the problem of checking the satisfiability of a propositional formula (the SAT problem). Therefore, SATMC relies on a SAT solver to check whether the generated SAT formula is satisfied. This problem is usually solved very efficiently by modern SAT solvers, which can tackle SAT problems of practical relevance in milliseconds. The satisfying assignment (if any) is then turned into the corresponding abstract attack trace. SATMC carries out an iterative deepening strategy on $k$. Initially $k$ is set to 0, and then it is incremented till either an attack is found or an upper bound is reached. More details on SATMC can be found in [9]. When developing the STIATE back-end we also added two important features to SATMC: (i) a multi-attack module to identify multiple abstract attack traces (if any) from a single formal specification; and (ii) a multi-thread environment allowing the execution of SATMC over multiple ASLan++ models in parallel using multiple and dependent threads.

**Example 8.** Let us consider the two ASLan files obtained in Example 7. No attacks are detected for the first file. On the contrary, when considering the second version, SATMC reports an abstract attack trace, i.e., a sequence of operations an attacker can perform leading to a violation of the authentication property (see KP4). Specifically, in the man-in-the-middle attack (depicted in Figure 11), the client $c$ initiates a session of the protocol to access a resource provided by the (malicious) service provider $\sp —$ pretending to be $c$ — and that mischievously reuses the authentication assertion received by $c$ to trick $\sp$ into believing he is $c$ (cf. last message in Figure 11 sent by $\sp(c)$). This is the same attack reported in [13] and already introduced in Section II.

**Mutation Engine.** Even if the formal model of a given security protocol is secure (i.e., no attack traces have been found by SATMC), the implementation may deviate from its formal design because of DevT mistakes or because some assumptions—made at design level—do not hold on the real implementation. This could lead to some vulnerabilities that can be exploited by an attacker in such a way to violate the desired security goals. Previous research [6], [15] have proposed to mutate secure formal models in order to inject well-known vulnerabilities that reflect most common security errors in web applications. These studies also provided evidence that test cases written upon attack traces obtained on mutated models are correlated to specific flaws at the source code level [6]. Following this direction, the STIATE back-end offers a mutation engine, developed as a further Eclipse-based plugin, to generate automatically mutated ASLan++ specifications starting from secure ones. Then, the mutated ASLan++ must be analysed by SATMC (upon previous conversion to ASLan), which checks whether the security goals still hold or the injected vulnerabilities compromise them.

To this aim, the mutation engine provides a set of semantic mutation operators for three categories of most dangerous well-known implementation errors that can lead to serious vulnerabilities in web applications according to the MITRE’s classification\(^3\): (i) missing channel properties, i.e., some assumptions about communication channels (e.g., data encryption, confidentiality etc.) that are true at design level might not

---

\(^3\)http://cwe.mitre.org/top25/index.html
be valid at implementation level; (ii) missing condition checks, which include all cases where a given formal model should contain a proper symbolic function that model a condition or authorization check; (iii) usage of multiple channels for carrying messages between the same pair of agents. Indeed, according to the MITRE’s classification, missing channels properties can cause missing encryption of data vulnerabilities, while missing condition or authorisation checks might lead to missing authentication for critical function or missing authorisation vulnerabilities. Finally, messages exchanged between two agents should be performed on the same pair of (dedicated) channels until the execution of the protocol is ended. When multiple communications between two agents are performed on multiple channels, this may lead to man-in-the-middle attacks.

It is important to highlight that, from a security testing point of view, we are interested in testing only those vulnerabilities that could violate the desired security properties of a given security web protocol, while the other kind of vulnerabilities should be considered/tested in the context of other type of software testing strategies, such as the functional or integration testing. With respect to previous work (e.g., [15]) that is semi-automated (i.e., the user/analyst has to mark explicitly which parts to mutate), the mutation engine of the STIATE back-end is fully automated.

Example 9. From Figure 1 we can see that C and SP communicate twice, first with messages (1)-(2) and later with (5)-(6). At design level it is expected that the channel used for carrying messages (1) and (5) is the same, and similarly for the channel used for carrying messages (2) and (6). This is the case in Example 4. Using the mutation engine, we can automatically mutate the secure ASLan++ file (i.e., version 1 reported in Example 7) in such a way that the corresponding protocol uses two different channel pairs for message exchanges (1)-(2) and (5)-(6). Using SATMC (upon previous conversion in ASLan), we found that the mutated model is affected by another man-in-the-middle attack. The attack found—upon mutation—corresponds to the one in [2], already introduced in Section II.

Test case execution. The result of the formal analysis (either for mutated or non-mutated specifications) is a set of abstract attack traces (if any). Concrete test cases are generated on top of the identified abstract traces and executed against the system under testing (SUT), i.e., the fragment to be tested of the overall system. The STIATE back-end employs a Test generation and Execution engine, specifically an extension of the Instrumentation-based Testing engine [11], an Eclipse-based plugin that supports testers and security experts during this process in a semi-automated fashion.

The concretisation phase amounts to (i) mapping abstract operations and symbols reported in the abstract attack traces to concrete operations (e.g., HTTP messages); (ii) selecting those Java methods that represent actions to be executed by simulated agent, and (iii) sequencing this subset according to the ordering described by the abstract trace. The output of this concretisation phase is a Java program implementing the concrete test case. Executing this Java program within a Java Runtime Environment results in executing the abstract test case against the SUT. This concretisation process is semi-automated since the first step requires an explicit definition of the mapping (via a test adapter). A mapping entry is a triple: ASLan symbol, mapping type, and value. For instance, if the ASLan element resourceurl (cf. first message of Figure 11) is mapped to the value http://www.google.com/calendar/hosted/xxx as a mapping type String, then the test adapter methods, corresponding to those ASLan primitives taking resourceurl as parameter, should expect an object of type String as real-world representation of resourceurl. A detailed description of the mapping definition is available in Sections 4.4.2 and 4.5.2 of [11].

It is worth to mention that binding together the catalog of functions (e.g., HTTP, SAML, etc.) used to annotate the messages (see Section IV) with the testing adapters (set of Java classes implementing construction and de-construction of those function in the real world) significantly reduces the input required from the end user in order to execute the tests. For instance, all security-annotated sequence diagrams that includes the catalog 4-ary function HttpServletRequest, ...
This paper presented STIATE, a toolkit for security threat identification and testing. The tool builds upon previous work, the SPaCioS tool [11] in particular, and fills some important gaps that were hindering the adoption of the solution in industries. In particular STIATE provides (i) a front-end for specifying sequence diagrams enriched with security annotations (including WHAT-IF conditions); (ii) a translator to automatically generate formal models; (iii) a model checker to detector threats arising from the design of the protocol; (iv) a mutation engine to detect threats arising from deviations between models and the real implementations; and (v) a semi-automated approach for generating and testing the threats obtained by the formal analysis against the real implementations. We demonstrated the usage of STIATE against the SAML Single Sign-On protocol showing that it would have enabled DevTs to detect the threats that originated the serious flaws of [13] and [2].

ACKNOWLEDGMENT

This work has partially been supported by the Activity no. 14231, “Security Threat Identification and Testing” (STIATE), funded by the EIT ICT Labs.

REFERENCES