Verifying the Application of Security Measures in IoT Software Systems with Model Learning

Sébastien Salva and Elliot Blot
LIMOS - UMR CNRS 6158, Clermont Auvergne University, France

Keywords: Model Learning, Model Checking, Expert System, IoT Software Systems.

Abstract: Most of today's software systems log events to record the events that have occurred in the past. Such logs are particularly useful for auditing security over time. But, the growing sizes and lack of abstraction of the logs make them difficult to interpret manually. This paper proposes an approach combining model learning and model checking to help audit the security of IoT software systems. This approach takes as inputs an event log and generic security measures described with LTL formulas. It generates one formal model for every component of an IoT system and helps auditors make the security measures concrete in order to check if the models satisfy them. The LTL formula instantiation is semi-automatically performed by means of an expert system and inference rules that encode some expert knowledge, which can be applied again to the same kind of systems with less efforts. We evaluate our approach on 3 IoT systems against 11 security measures provided by the European ENISA institute.

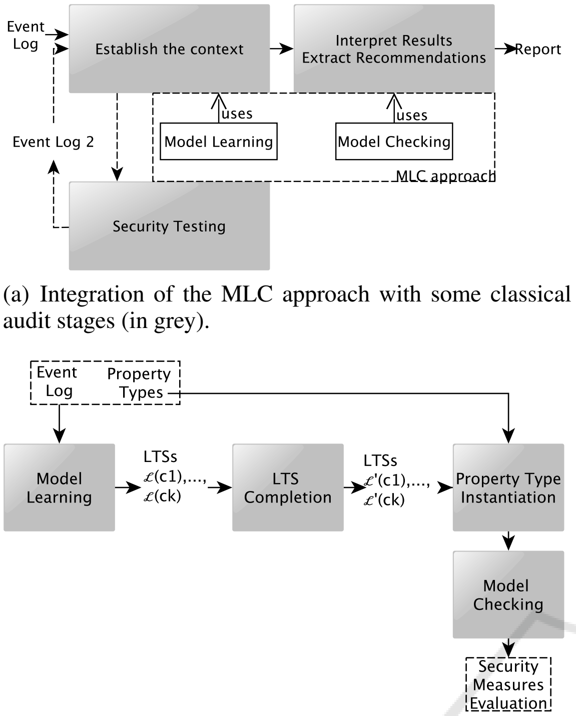
1 INTRODUCTION

The Internet of Things (IoT) is a broad concept comprising a wide ecosystem of interconnected services and devices, such as sensors, or industrial and health components, connected to the Internet. As many kinds of software are involved within the IoT concept, e.g., for data collection, device cooperation, or real-time analytic, it is not surprising to observe that IoT systems are vulnerable to a wide range of security attacks. After the alarming surge of cyber-attacks on IoT systems revealed during the few past years, organisations have started to be aware about the importance of including cyber-security in their IoT solutions. More and more organisations appeal to both internal and external auditors to check the security of their IoT-based services and to assess the related risks. Most of these activities are time-consuming and sometimes challenging, especially if they are performed from scratch.

Some papers or documents provide guidelines to establish a security audit process. For instance, the European ETSI and the U.S. NIST institutes have proposed methods and activities dedicated to undertake testing and risk assessment activities (NIST, 2018; ETSI, 2015). Others documents, e.g., the reports of the European Union Agency for Cybersecurity (ENISA) (ENISA, 2017), put forward security measures, good practices, and threats taxonomy. Furthermore, some approaches and tools have been proposed to automate some steps of an audit process, and particularly the security testing stage. Some of them adopt Model-based Testing (MbT) (Ahmad et al., 2016; Matheu-García et al., 2019), i.e. test cases are derived from a (formal) model that expresses the behaviour of the system. Other passive approaches monitor IoT systems in order to detect the violation of formal properties (Siby et al., 2017). Despite the strong benefits brought by these approaches, the efforts required for writing (formal) models usually hamper their adoptions.

In this paper, we propose a Model-Learning-Checking solution (shortened MLC) to this problem, which combines model-learning to generate formal models of IoT systems and model-checking to evaluate whether security properties hold on these models. Figure 1(a) illustrates its integration with some security audit stages given in the NIST or ETSI security audit frameworks. The step "Establish the context", which is often manually done by interpreting diverse documents, is partly performed here by means of a model learning algorithm, which builds, from logs, behavioural models capturing what happens in the IoT system. From these models, MbT approaches can be used to test whether the IoT system is vulnerable. While testing, more logs may be collected and models

Salva, S. and Blot, E.
Verifying the Application of Security Measures in IoT Software Systems with Model Learning.
DOI: 10.5220/0009872103500360
In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT 2020)*, pages 350-360
ISBN: 978-989-758-443-5
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved



(a) Integration of the MLC approach with some classical audit stages (in grey).

(b) Approach overview.

Figure 1: The MLC approach.

can be re-generated to capture more behaviours. In the meantime, the models can be analysed to detect further security issues. Usually, security properties are expressed with formulas that are evaluated with a model-checker. These formulas may express different security aspects, e.g. vulnerabilities. But, as the security testing stage may also be used to detect them, we prefer focusing on formulas expressing whether security measures used to protect an IoT system against threats are correctly implemented. Usually, such formulas need to be adapted for every model so that they share an alphabet. This activity is known to be difficult and time consuming. To make this activity easier, our MLC approach helps instantiate formulas by means of an expert system. In summary, the major contributions presented in the paper are:

- a Model-Learning-Checking solution to help audit the security of IoT systems. We use the Labelled Transition System (LTS) to express the behaviours of every component of an IoT system. We express the security measures proposed by the ENISA institute with LTL formulas composed of predicates. Our approach verifies whether the LTSs satisfy these measures and returns counterexamples when issues are detected. The former may be used to interpret the results and provide countermeasures;
- a technique to assist auditors in instantiating LTL formulas with less effort. Intuitively, we make use of an expert system to help complete LTS transitions by injecting some predicates used to formulate the security measures. The expert system automates the completion, and saves time as the expert knowledge encoded in its inference rules may be applied again to the same kind of IoT systems;
- an implementation of our MLC approach for HTTP devices. We also formulated the ENISA measures related to communications. These are used to evaluate our approach on 3 IoT systems.

The paper is organized as follows: Section 2 gives some preliminaries. Section 3 presents our MLC approach and its algorithms. The next section summarises the results of experiments used to evaluate the sensitivity and specificity of the MLC approach. We discuss related work in Section 5. Section 6 summarises our contributions and draws some perspectives for future work.

2 BACKGROUND

2.1 The ENISA Security Measures

Several papers propose lists of recommendations to improve security (Zhang et al., 2015; Khan and Salah, 2018; OWASP, 2003). Among them, the ENISA organisation issued several documents exposing guidelines and security measures to implement secure IoT software systems with regard to different contexts (smart plants, hospitals, clouds, etc.). The security measures come from several other documents written by different organisations or institutes, e.g., ISO, IETF, NIST, ENISA or Microsoft. We have chosen to focus on the paper related to security baselines in the context of critical information infrastructures (ENISA, 2017). This document gathers 57 security technical measures that should be implemented and valid on IoT systems along with the threats that are addressed. The security measures cover a wide range of security considerations, such as security by design, data protection, risk analysis, etc.

2.2 The LTS Model and the Linear Temporal Logic

We consider the LTS to model the behaviours of every component of an IoT system. This model is defined in terms of states and transitions labelled by label sets, which express what happens.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Definition 1 (LTS.) *A Labelled Transition System (LTS) is a 4-tuple $\langle Q, q0, L, \rightarrow \rangle$ where:*

- *Q is a finite set of states; $q0$ is the initial state;*
- *L is a finite set of labels,*
- *$\rightarrow \subseteq Q \times (\mathcal{P}(L) \setminus \{\emptyset\}) \times Q$ is a finite set of transitions (where $\mathcal{P}(L)$ denotes the powerset of L. A transition (q, L', q') is also denoted $q \xrightarrow{L'} q'$.*

An execution trace is a finite sequence of sets of labels. ε denotes the empty sequence.

Furthermore, we express security properties with LTL formulas, which concisely formalise them with the help of a small number of special logical operators and temporal operators (Holzmann, 2011). Given a set of atomic propositions AP and $p \in AP$, LTL formulas are constructed by using the following grammar $\phi ::= p \mid (\phi) \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}\phi \mid \phi\mathbf{U}\phi$. Additionally, a LTL formula can be constructed with the following operators, each of which is defined in terms of the previous ones: $\top \overset{def}{=} p \vee \neg p$, $\bot \overset{def}{=} \neg\top$, $\phi_1 \wedge \phi_2 \overset{def}{=} \neg(\neg\phi 1 \vee \neg\phi_2)$, $\mathbf{F}\phi \overset{def}{=} \top\mathbf{U}\phi$, $\mathbf{G}\phi \overset{def}{=} \neg\mathbf{F}(\neg\phi)$.

3 THE MLC APPROACH

We present in this section our Model-Learning-Checking (MLC) approach, which aims at helping audit an IoT system, denoted SUA. It takes as inputs an event log and generic LTL formulas composed of predicates. Following the terminology used in (Beschastnikh et al., 2015), we call these formulas *property types*. These are generic properties having a pattern-level form, which have to be instantiated before being evaluated.

Figure 1(b) illustrates the successive steps of the MLC approach. It starts by learning models from an event log. For every component $c1$ of SUA, it generates one LTS $\mathcal{L}(c1)$ expressing the behaviours of $c1$ along with one dependency graph $Dg(c1)$ expressing how $c1$ interacts with the other components of SUA. Both help understand the architecture of SUA and its general functioning.

The two next steps, which use these models, help auditors verify whether every LTS satisfies the property types. As these are generic, the LTSs and property types do not share the same alphabet. Therefore, the auditor should re-formulate all the property types for every LTS. Instead, our approach tries to ease this task as follows. Given a LTS $\mathcal{L}(c1)$, the step "LTS Completion" extends the LTS semantics; it analyses the LTS paths and injects new labels on transitions. These labels correspond to some predicates of the property types whose variables are assigned to concrete values. The step automates the label injection by using an expert system made up of inference rules, which encode some expert knowledge about the kind of system under audit. The step produces a new LTS $\mathcal{L}'(c1)$. The next step "Property type instantiation" covers every new LTS and automatically instantiates the property types by means of the labels added in the previous step. This step returns a set of LTL formulas $\mathcal{P}(\mathcal{L}'(c1))$ exclusively written with atomic propositions. We call them property instances. The final step calls a model-checker to check whether the LTS $\mathcal{L}'(c1)$ satisfies the LTL formula of $\mathcal{P}(\mathcal{L}'(c1))$.

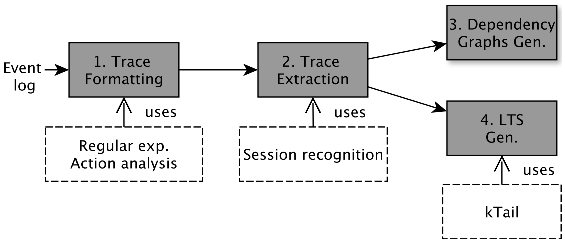


Figure 2: Model learning with the CkTail approach.

These steps are now detailed in the following and illustrated with an example.

3.1 Model Generation

We proposed a model learning approach called Communicating system kTail, shortened CkTail, to automatically learn models of communicating systems from event logs. We summarise in this section the functioning of CkTail, but we refer to (Salva and Blot, 2020a) for the technical details. The CkTail's algorithms rely on some assumptions, which are given below:

- **A1 Event Log:** we consider the components of SUA as black-boxes. We assume that each device, server, or gateway is physically secured and that we only have access to the network and user interfaces. Event logs are collected in a synchronous environment. Furthermore, the messages include timestamps given by a global clock for ordering them. We consider having one event log;
- **A2 Message Content:** components produce messages that include parameter assignments allowing to identify the source and the destination of the message. Other parameter assignments may be used to encode data. Besides, a message is either identified as a request or a response.
- **A3 Device Collaboration:** components can run in parallel and communicate with each other. The components of SUA follow this strict behaviour: they cannot run multiple instances; requests are

processed by a component on a first-come, first served basis. Besides, every response is associated to the last request w.r.t. the request-response exchange pattern.

The assumption A3 helps extract sessions of the system in event logs, i.e. a temporary message interchange among components forming a behaviour of the whole system from one of its initial states to one of its final states. Usually, the use of session identification strongly facilitates the trace extraction. Unfortunately, we have observed that this technique is seldom used with IoT systems.

Figure 2 illustrates the 4 steps of CkTail. The event log is firstly formatted with tools or regular expressions into a sequence of events of the form $a(\alpha)$ with a a label and α some parameter assignments. In reference to A1, A2, an event $a(\alpha)$ indicates the sources and destinations of the messages with two parameters $from$ and to. The other parameter assignments may capture acknowledgements or sensor data. P denotes the parameter assignment set. Figure 3 illustrates a simple example of event sequence obtained after its first step. For simplicity, the labels directly show whether an event encodes either a request or a response.

The second step "Trace Extraction" segments the event sequence with an algorithm that tries to recognise sessions by means of the previous assumptions. In the meantime, this algorithm detects dependencies among the components of SUA. We have defined the notion of component dependency by means of three expressions formulating when a component relies on another one. Intuitively, the two first expressions illustrate that a component c_1 depends on another component c_2 when c_1 queries c_2 with a request or by means of successive nested requests of the form $req1(from := c_1, to := c)req2(from := c, to := c_2)$. The last expression deals with data dependency. We say that c_1 depends on c_2 if c_2 has sent an event $a_1(\alpha_1)$ with some data, if there is a unique chain of events $a_1(\alpha_1)\dots a_k(\alpha_k)$ from c_2 sharing this data and if $a_k(\alpha_k)$ is a request whose destination is c_1.

The third step "Dependency graph Generation" builds one dependency graph $Dg(c_i)$ for every component $c_i \in C$ of SUA. These show in a simple way how the components interact together or help identify central components that might have a strong negative impact on SUA when they integrate faults or are vulnerable to security attacks. Figure 3 illustrates the models generated by CkTail from the event sequence. CkTail detects that SUA is made up of three components and builds three dependency graphs. For instance, as the event sequence includes a request from the component c1 to c2, CkTail builds a dependency graph for c1 showing that c1 depends on c2.

The last step "LTS Generation" of CkTail builds one LTS, denoted $\mathcal{L}(c_i)$ for every component c_i participating in the communications of SUA. The LTS are reduced by calling the kTail algorithm (Biermann and Feldman, 1972), which merges the (equivalent) states having the same k-future, i.e. the same event sequences having the maximum length k. Figure 3 illustrates the 3 LTSs generated from the event sequence used in this example. The LTS transitions are labelled by sets composed of one event of the form $a(\alpha)$, which is either an input or an output. For instance the event Req3(from:=c2to:=c3, switch:=on) in the initial event sequence, has been doubled with an output !Req3 and an input ?Req3. The former is labelled on the transition $q_4 \rightarrow q_5$ of the LTS $\mathcal{L}(c2)$ to express that an output !Req3 is sent by the component c2; the latter is labelled on the transition $q_0 \rightarrow q_1$ of $\mathcal{L}(c3)$ to express that c3 expects to receive the input ?Req3.

3.2 Property Type

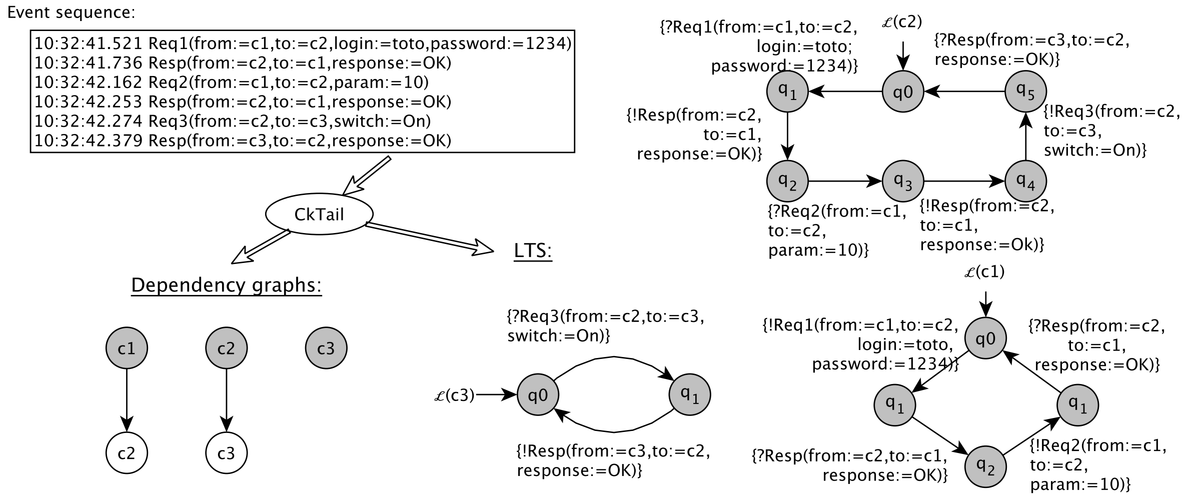
A property type formulates a general feature that is independent of the type of system under audit. A property type is a specialised LTL formula that captures the temporal relationships between predicates. A predicate is either an expression of one or more variables defined on some specific domains, or nullary predicate, that is, an atomic proposition.

Definition 2 (Property Type.)

- *Pred denotes a set of predicates of the form* p *(nullary predicate) or* $p(x_1,\dots,x_k)$ *with* $x_1,\dots,x_k$ *some variables that belong to the set* X*;*
- *The domain of a predicate variable* $x \in X$ *is denoted* $Dom(x)$*;*
- *A property type* Φ *is a LTL formula built up from predicates in Pred.* $\mathcal{P}$ *denotes the set of property types.*

As property types are composed of predicates, they should be instantiated before being given to a model-checker. The instantiation of a property type is called a property instance. It has the same LTL structure as its property type, but the predicate variables are assigned to values to form propositions.

Definition 3 (Property Instance.) *A property instance* ϕ *of the property type* Φ *is a LTL formula resulting from the instantiation of the predicates of* Φ.



Event sequence:

```
10:32:41.521 Req1(from:=c1,to:=c2,login:=toto,password:=1234)
10:32:41.736 Resp(from:=c2,to:=c1,response:=OK)
10:32:42.162 Req2(from:=c1,to:=c2,param:=10)
10:32:42.253 Resp(from:=c2,to:=c1,response:=OK)
10:32:42.274 Req3(from:=c2,to:=c3,switch:=On)
10:32:42.379 Resp(from:=c3,to:=c2,response:=OK)
```

CkTail
Dependency graphs:
c1
c2
c3
c2
c3
LTS:
{?Req1(from:=c1,to:=c2, login:=toto; password:=1234)}
ℒ(c2)
{?Resp(from:=c3,to:=c2, response:=OK)}
q1
q0
q5
{!Resp(from:=c2, to:=c1, response:=OK)}
{!Req3(from:=c2, to:=c3, switch:=On)}
q2
q3
q4
{?Req2(from:=c1, to:=c2, param:=10)}
{!Resp(from:=c2, to:=c1, response:=Ok)}
ℒ(c1)
{?Req3(from:=c2,to:=c3, switch:=On)}
ℒ(c3)
q0
q1
{!Resp(from:=c3,to:=c2, response:=OK)}
{!Req1(from:=c1,to:=c2, login:=toto, password:=1234)}
q0
{?Resp(from:=c2, to:=c1, response:=OK)}
q1
q1
q2
{?Resp(from:=c2,to:=c1, response:=OK)}
{!Req2(from:=c1, to:=c2, param:=10)}

Figure 3: Example of model generation with CkTail.

The function that instantiates a property type to one property instance, i.e. which associates each variable of the predicates to a value, is called a binding:

Definition 4 (Property Binding.) *Let X' be a finite set of variables $\{x_1, \dots, x_k\} \subseteq X$. A binding is a function $b : X' \to Dom(X')$, with $Dom(X') = Dom(x_1) \times \dots \times Dom(x_k)$.*

We recommend writing property types by firstly formulating general security concepts with predicates, and by applying or composing the LTL patterns given in (Dwyer et al., 1999) on those predicates. These patterns help structure LTL formulas with precise and correct statements that model common situations, e.g., the absence of events, or cause-effect relationships.

As the LTSs generated by CkTail express communications among the components of SUA, we formulated the 11 security measures provided by the ENISA organisation related to communications, which cover the following domains: Authentication, Privacy, Secure and Trusted Communications, Access Control, Secure Interfaces and Network Services, Secure Input and Output Handling, Trust and Integrity Management. Due to lack of room, these are given in (Salva and Blot, 2020b). We defined the unary and nullary predicates given in Table 1. All the unary predicates have variables that can be assigned to values in $C \cup P$ with C the component set of SUA and P the parameter assignments given in the messages.

For instance, the ENISA measure GP-TM-24 recommends encrypting authentication credentials. This measure is formulated as: $\mathbf{G}((loginAttempt(c) \wedge credential(x)) \rightarrow encrypted(x))$, which intuitively means that every time a component attempts to log in to another component c by using credentials x, x must be encrypted. The measure GP-TM-53, which suggests that error messages must not expose sensitive information, can be written with $\mathbf{G}((errorResponse \vee$

Table 1: Predicates defined from the 11 ENISA measures related to communications.

Predicate	Short Description
begin	beginning of a new session
end	end of a session
$from(c)$	message came from c
$to(c)$	message sent to c
Request	message is a request
Response	message is a response
input	message is an input
output	message is an output
getUpdate(x)	response including an update file
cmdSearch-Update	the component received the order to search for an update
$sensitive(x)$	data x includes sensitive data
$credential(x)$	data x includes credentials
$encrypted(x)$	data x is encrypted
searchUpdate	component request for an update
$loginAttempt(c)$	authentication attempt with c
$authenticated(c)$	successful authentication with c
$loginFail(c)$	failed authentication with c
$lockout(c)$	c is locked due to repetitive authentication failures
password-Recovery	component uses a password recovery mechanism
blackListed-Word	message includes black listed words
validResponse	correct response with correct status
errorResponse	response containing an error message
Unavailable	component that received the request is unavailable
$XSS(x)$	data x includes an XSS attack
$SQLinjection(x)$	data x includes an SQL injection attack

$\neg validResponse) \rightarrow (\neg(blackListedWord)$. This formula intuitively means that every HTTP response composed of an error message or having a status higher than 299 must not include blacklisted words. If we apply the binding $\{c \rightarrow c1, x \rightarrow login := toto\}$ on the first property type, we obtain the property instance $\mathbf{G}((loginAttempt(c1) \wedge credential(login := toto)) \rightarrow encrypted(login := toto))$.

3.3 LTS Completion

As stated previously, our approach aims at checking whether a LTS $\mathcal{L}(c1)$ describing the behaviours of a component $c1$ of SUA satisfies property types. This activity is relevant on condition that the LTS alphabet shares some predicates of the property types. At the beginning of this step, this requirement is not met as our property types are generic. This step helps auditors complete LTSs by injecting some predicates of *Pred* on LTS transitions. As a LTS encodes concrete behaviours, this step actually adds instantiated predicates or propositions, i.e. predicates of *Pred* whose variables are assigned to concrete values found in the LTS events. Injecting propositions comes down to analysing/interpreting LTS transitions or paths and to add propositions on transitions to extend the LTS semantics. Our approach uses an expert system, made up of inference rules, to automate the proposition injection and to save time by allowing to reuse the rules on several IoT systems.

We represent inference rules with this format: *When conditions on facts Then actions on facts* (format taken by the Drools inference engine ("Red-Hat-Software", 2020)). The facts, which belong to the knowledge base of the expert system, are here the transitions of $\rightarrow_{\mathcal{L}(c1)}$. To ensure that the transition completion is performed in a finite time and in a deterministic way, the inference rules have to meet these hypotheses:

- (finite complexity:) a rule can only be applied a limited number of times on the same knowledge base,
- (soundness:) the inference rules are Modus Ponens (simple implications that lead to sound facts if the original facts are true).

We devised 28 inference rules, which are available in (Salva and Blot, 2020b). These can be categorised has follows:

- Structural information: two rules are used to add the propositions "begin" and "end", which describe the beginning and end of sessions in LTS paths;

```
rule "validResponse"
when
$t : Transition(isReq() == false,
isOk())
then
$t.addLabel("validResponse");
end
```

```
rule "authentication"
when
$t1: Transition(isRequest(), contains("
login"), contains("password"))
$t2: Transition(isResponse(),isOK(),
sourceState = $t1.targetState)
then
$t1.addLabel("loginAttempt("+
compoSender($t1) + ")");
$t2.addLabel("authenticated("+
compoSender($t1) + ")");
end
```

Figure 4: Inference rule examples.

- Nature of the events: 9 rules add information about the nature of the events of $\mathcal{L}(c1)$. Given a transition $q_1 \xrightarrow{\{a(\alpha)\}} q_2$, some rules analyse the parameter assignments in α and complete the transition with new propositions expressing that $a(\alpha)$ is a request or a response, an input or an output, the component which sent $a(\alpha)$, etc. Other rules analyse α to interpret if the message is an error response (analysis of the values in α to detect words like "error" or analysis of HTTP status, etc.). The first rule of Figure 4 adds the proposition validResponse if the transition is a response whose HTTP status is between 200 and 299. The status control is performed by the method isOk();
- Security information: the other rules add information related to security on LTS transitions. For instance, we devised a rule that checks whether the message content is encrypted. Other rules analyse the LTS paths to try recognise specific patterns (transition sequences containing specific words), e.g., authentication attempts, successful or failed authentications. Intuitively, these rules try to detect these patterns in the LTS paths. For instance, the second rule of Figure 4 detects a correct authentication. It adds the proposition "loginAttempt(c)" on the transition labelled by the credentials and "authenticated(c)" on the transition whose event encodes a correct authentication with the external component c.

Once the expert system has completed the LTS $\mathcal{L}(c1)$, we obtain a new LTS denoted $\mathcal{L}'(c1)$. Now, the auditors have to inspect $\mathcal{L}'(c1)$ to assess the cor-

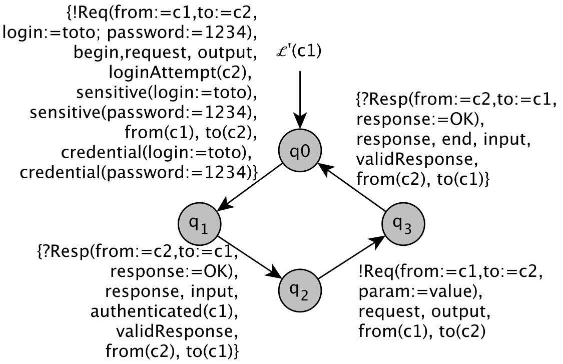


Figure 5: Example of LTS completion. New propositions (begin, end, credential, sensitive, validResponse, etc.) are injected on transitions.

rectness of the LTS completion. On the one hand, some propositions may be missing on account of missing or incomplete rules. To help detect the missing propositions, this step returns the predicates of *Pred* that are not used for every LTS completion. On the other hand, some propositions might be wrong on account of misinterpretations, e.g., an incorrect recognition that messages are encrypted.

Figure 5 illustrates the LTS $\mathcal{L}'(c1)$ completed by this step from the LTS $\mathcal{L}(c1)$ of Figure 3. The transitions of $\mathcal{L}'(c1)$ are still labelled by the events of the original LTS, but several new propositions are available. For instance, the expert system has detected in the transition $q_0 \rightarrow q_1$ that login and password are sensitive data, used as credentials.

3.4 Property Type Instantiation

Given a LTS $\mathcal{L}'(c1)$, this step aims at instantiating the property types of $\mathcal{P}$ to obtain a set of property instances that can be evaluated on the paths of $\mathcal{L}'(c1)$. We denote $\mathcal{P}(\mathcal{L}'(c1))$ this set of property instances. Intuitively, a property type Φ is instantiated with bindings, which assign values to all of its predicate variables. We compute these bindings from the values of the instantiated predicates added by the expert system previously.

This step is implemented by Algorithm 1, which takes as inputs a LTS $\mathcal{L}'(c1)$, the property type set $\mathcal{P}$ and returns $\mathcal{P}(\mathcal{L}'(c1))$. The algorithms starts by building the special domain $Dom(deps)$, which gathers the dependent components of $c1$. This domain is computed by covering the dependency graph $Dg(c1)$ of $c1$. The special variable *deps* is used with some property types encoding the notion of dependency among components. Then, Algorithm 1 instantiates every property Φ of $\mathcal{P}$ one after another. It covers each label of the transitions $q_1 \xrightarrow{L} q_2$ of $\mathcal{L}'(c1)$ (lines 4-7). If a label $P(v_1,\dots,v_k)$ corresponds to an instantiation of a predicate $P(x_1,\dots,x_k)$ used in Φ, then the values assigned to the variables $x_1,\dots,x_k$, are added to the sets $Dom(x_1),\dots,Dom(x_k)$. The variables $x_1,\dots x_n$ are added to a variable set X', which gathers the variables that are assigned to values. Once all the labels are covered, the algorithm checks whether all the predicate variables of Φ can be assigned to values (line 8). If this is false, a warning explaining that Φ cannot be instantiated is returned. This kind of warning is used to once more help auditors control that there is no missing instantiated predicate. Otherwise, the algorithm computes all the possible bindings of $D^{X'}$ with D the Cartesian product $D = Dom(x_1) \times \cdots \times Dom(x_n)$. And it repetitively instantiates Φ for each binding of $D^{X'}$ to produce the property instance ϕ, which is finally added to $\mathcal{P}(\mathcal{L}'(c1))$.

Let 's illustrate this step with the LTS $\mathcal{L}'(c1)$ of Figure 5 and the property type $\mathbf{G}((loginAttempt(c) \wedge credential(x)) \rightarrow encrypted(x))$ derived from the ENISA measure GP-TM-24. By covering the labels of $\mathcal{L}'(c1)$, we obtain $Dom(c) = \{c2\}$ and $Dom(x) = \{login := toto, password := 1234\}$. Two bindings can be constructed along with two property instances.

Algorithm 1: Property Type Instantiation.

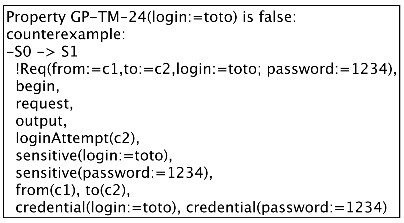
```
   input : LTS $\mathcal{L}'(c1)$, property type set $\mathcal{P}$
   output: Property instance set $\mathcal{P}(\mathcal{L}'(c1))$
1  Compute $Dom(deps)$ from $Dg(c1)$;
2  $X' := \emptyset$;
3  foreach $\Phi \in \mathcal{P}$ do
4      foreach $l \in L$ with $s_1 \xrightarrow{L} s_2 \in \rightarrow_{\mathcal{L}'(c1)}$ do
5          if $l = P(v_1,\dots,v_k)$ and $P(x_1,\dots,x_k)$ is a predicate of $\Phi$
           then
6              $Dom(x_i) = Dom(x_i) \cup \{v_i\} (1 \le i \le k)$;
7              $X' := X' \cup \{x_1,\dots,x_k\}$;

8      if $X'$ is not equal to the set of predicate variables of $\Phi$ then
9          return a warning;
10     else
11         $D := Dom(x_1) \times \cdots \times Dom(x_n)$ with $X' = \{x_1,\dots,x_n\}$;
12         foreach binding $b \in D^{X'}$ do
13             $\phi := b(\Phi)$;
14             $\mathcal{P}(L'(c1)) = \mathcal{P}(L'(c1)) \cup \{\phi\}$;
```

3.5 Property Instance Verification

Given a LTS $\mathcal{L}'(c1)$, and a property instance set $\mathcal{P}(\mathcal{L}'(c1))$, the last step calls a model-checker to check whether $\mathcal{L}'(c1)$ satisfies the property instances of $\mathcal{P}(\mathcal{L}'(c1))$. If the model-checker finds a counterexample path that violates a property instance ϕ, our approach reports to the user that the related se-

SCIENCE AND TECHNOLOGY PUBLICATIONS



```
Property GP-TM-24(login:=toto) is false:
counterexample:
-S0 -> S1
  !Req(from:=c1,to:=c2,login:=toto; password:=1234),
  begin,
  request,
  output,
  loginAttempt(c2),
  sensitive(login:=toto),
  sensitive(password:=1234),
  from(c1), to(c2),
  credential(login:=toto), credential(password:=1234)
```

Figure 6: Example of results with the ENISA measure GP-TM-24. The LTS does not satisfy a property instance related to GP-TM-24 because the login is not encrypted.

curity measure is not valid. The counterexample is also returned as it may be analysed to elaborate recommended actions. A counterexample is particularly useful when a security measure was incorrectly applied during the development of the IoT system. The counterexample should help understand why a component $c1$ does not meet the measure and should help localise a problem in the LTS $\mathscr{L}'(c1)$.

Figure 6 shows a example of result returned by the model-checker NuSMV (Cimatti et al., 2002) after having evaluated if the LTS $\mathscr{L}'(c1)$ satisfies the a property instance related to the ENISA measure GP-TM-24. The interpretation of the counter-example helps deduce that the login should have been encrypted. Such counter-examples may be used to develop an audit report, which should include recommendations or treatments to security issues.

4 PRELIMINARY EVALUATION

Our approach is implemented as a tool chain, which gathers the model learning tool CkTail, and a tool called SMVmaker, which completes LTSs by means of an expert system, instantiates property types and calls the NuSMV model-checker. The tools, examples of property types and traces are available in (Salva and Blot, 2020b). With this implementation, we conducted some experiments on three use-cases of IoT systems in order to evaluate the sensitivity and the specificity of our MLC approach, that is its capability to uncover active security measures that are correctly implemented in an IoT system, and to identify security measures that are not considered or incorrectly implemented.

Setup: the three use-cases are based on devices, gateways and external cloud servers communicating over HTTP. We provide below some details about these systems:

- **UC1** is composed of 3 motion sensors that turn on a light-bulb when they detect movements. They communicate through a gateway; all of them have user interfaces, which may be used for configuration. The main purpose of this use-case is to focus on classical attacks (code injection, brute force, availability) and on the related security measures;
- **UC2** includes a motion sensor, a switch and a camera. The motion sensor communicates with the switch in clear-text, and with the camera with encrypted messages. The camera also communicates with external clouds requiring authentication and encryption. In this use-case, we mainly focus on vulnerabilities and security measures related to encryption and authentication;
- **UC3** is composed of 3 cameras, which communicate with external cloud servers. Two cameras can check for updates and install them. This use-case aims at focusing on vulnerabilities and security measures related to the update mechanisms.

For every use-case, we collected a first log that includes traces of ”normal behaviours” and generated first models with CkTail. We identified the testable components, on which we applied a set of penetration testing tools. During this testing stage, we collected a second larger log. Testing was only possible on the components of UC1. We manually analysed the source code of the software (when available) and the logs to list the ENISA security measures that have been implemented and those that are not applied. Then, we generated LTSs along with dependency graphs. We used the 11 property types formulating the security measures provided by the ENISA organisation related to communications. We called SMVmaker to generate property instances and NuSMV to check whether these property instances hold on LTSs.

Results: Table 2 summarises the experiment results. Col. 2,3 show for every use-case the number of known components and the number of generated LTSs. The difference # LTSs - # components gives the number of unknown external components, i.e. cloud servers with our use-cases. The remaining columns give the results obtained for the known components only because we can only give the sensitivity and specificity of these components by comparing the results given by our tool chain with our observations. Col. 6,7 give the number of property types that have been instantiated and the instance number. Col.8 shows the number of measures that are not correctly implemented. The two last columns provide the sensitivity (rate of satisfied property instances that are correctly evaluated) and the specificity (rate of unsatisfied property instances that are correctly evaluated).

With Table 2 (col. 3,4), we can firstly observe that the models along with the property instance sets are large. This confirms that manually analysing the security of real IoT systems and writing LTL formu-

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 2: Experiment results: col. 8,9 give the sensitivity and specificity of our approach on 3 use-cases.

IoT syst.	# components	# LTSs	# states	# transitions	# property type instantiated/total	# property instances	# detected issues	Sensitivity	Specificity
UC1	5	5	1332	1553	38/55	402	17	99.5%	99.1%
UC2	3	8	1416	2174	14/33	31	3	100%	88.9%
UC3	3	11	168	853	15/33	32	4	100%	92.8%

las is difficult. None of these use-cases allows us to instantiate all the property types. This result was expected here as the components do not implement all the security features captured by the predicates of *Pred*, e.g., password recovery for UC1, or authentication for some components of UC2, UC3. We also observed that a few property types were not instantiated on account of the lack of precision of some rules of the expert system. For instance, we observed that the rule dedicated to recognise encrypted messages does not always return correct results. At the moment, the rule computes the message entropy to decide if it is encrypted or not. Unfortunately, the entropy is not precise enough, but we didn't find any better solution in the literature. SMVmaker warns the user when some predicates of *Pred* are not used or when some property types are not instantiated. As stated in Section 3, he or she still has to check if this is on account of wrong/missing LTS completions or of missing security features not implemented in the components. The overall sensitivity is 99,6% and the specificity is 98.7%. These results tend to show that our MLC approach is effective to help auditors check if security measures are implemented. The results are especially relevant on UC1, which is the only system experimented with penetration testing tools. Indeed, the latter indirectly increased the log file size, the models and the property instance number. On these 3 use-cases, we obtain few false positives or negatives though. After inspection of the LTSs, we observed that all of them come from under-approximations (rejection of valid behaviours) in the models. At the moment, none of the passive model learning tool generates exact models, they usually are under- or over-approximated (acceptance of invalid behaviours). We showed in (Salva and Blot, 2020a) that CkTail is one the most effective approach for generating models of communicating systems, but here under-approximation leads to less than 2% of incorrect results anyway.

5 RELATED WORK

5.1 Model Learning

Some model learning approaches, which infer models of communicating systems, have been proposed in the literature. In the field of active learning, these papers (Petrenko and Avellaneda, 2019; Groz et al., 2008) present algorithms that recover models through active testing. This kind of active technique implies that the system is testable and can be queried. In (Groz et al., 2008), the learning of the models is done in isolation. This constraint is lifted in (Petrenko and Avellaneda, 2019) by testing a system with unknown components by means of a SAT solving method. An active learning approach specialised to IoT protocols has also been proposed in (Tappler et al., 2017). It requires several implementations to perform a comparison of several models, which allows the detection of potential bugs. In contrast, our approach is passive, and learns models from logs. The requirements are hence quite different than those used in these papers.

As stated previously, we have proposed the model learning approach CkTail in (Salva and Blot, 2020a). The first step of the MLC approach uses CkTail to generate models. The two next steps, which correspond to the main contribution of this paper, assist auditors instantiate property types with an expert system and verify whether the LTSs satisfy property instances. We presented a comparison of CkTail with most of the passive model learning techniques of communicating systems (Mariani and Pastore, 2008; Beschastnikh et al., 2014; Salva and Blot, 2019) in (Salva and Blot, 2020a). In summary, we showed that CkTail builds more precise models thanks to its trace segmentation algorithm. Besides, compared to CSight, CkTail requires less constraints (CSight requires specific trace sets, which are segmented with one subset by component, the components have to be known in advance, etc.). Furthermore, CkTail is the only approach among these ones that detects component dependencies and expresses them with DAGs. The latter are used in our MLC approach to instantiate LTL formula.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

5.2 IoT Audit

A plethora of surveys or papers have exposed the opportunities, challenges, requirements, threats or vulnerabilities involved in the IoT security. Among them, several approaches have been proposed to audit IoT systems. Some papers (Nadir et al., 2019; Lally and Sgandurra, 2018) propose to audit devices with check lists or threats models, which have been devised or extended from the recommendations proposed by the OWASP organisation (OWASP, 2003). Other works focus on the audit of IoT device by decrypting the traffic sent via TLS (Wilson et al., 2017) so that TLS traffic can be verified without compromising future traffic. This king of technique could be used prior our MLC approach to obtain readable logs.

Many approaches rely on models to analyse the security of IoT systems. In (Ge et al., 2017), security models are devised from data collected from an IoT system. A security analysis is then performed to find potential attack scenarios, evaluate security metrics and assess the effectiveness of different defense strategies. Other works introduce MbT security testing methods. Some of them are said to be active, i.e. test cases are generated by hands or from a given specification, and are later used to experiment IoT systems (Gutirrez-Madroñal et al., 2019; Ahmad et al., 2016; Tappler et al., 2017; Matheu Garcia et al., 2019). Others are called passive, i.e. they are based upon monitoring tools, which security issues by checking rule satisfiability in the the long run (Siby et al., 2017; Chaabouni et al., 2019; Maksymyuk et al., 2017). The tool IoTSAT (Mohsin et al., 2016) is a SMT based framework, which helps analyse the security of IoT systems. IoTSAT models the device-level interactions as in our approach, but also policy-level behaviours and network-level dependencies. In comparison to our MLC approach, the works (Ge et al., 2017; Matheu Garcia et al., 2019) go further in the risk assessment by proposing the evaluation of metrics. IoTSAT also goes further in the modelling of the IoT system environment. But all of them require models or formal properties, which need to be manually devised. Most of the active testing technique could complement our MLC approach to get mode logs, as illustrated in Figure 1(a).

6 CONCLUSION

We have proposed an approach combining model learning and model checking to help audit the security of IoT systems. Security properties are modelled with generic LTL formulas called property types. This approach assists auditors in the instantiation of these property types by means of an expert system made up of inference rules, which encode some expert knowledge about the kind of system under audit.

As future work, we firstly plan to evaluate our MLC approach on further kinds of communicating systems, e.g., web services. We observed that the use of an expert system offers a great potential for instantiating property types. However, this benefit strongly depends on the successful implementation of the expert system rules. We indeed observed in the experimentations that few property types were not completely instantiated on account of the lack of precision of some rules. This is why the property type instantiation and LTS completion steps are performed in a semi-automatic manner to allow the detection of potential issues. We will investigate how to alleviate the need of this manual inspection. A first direction is to complete the approach with a preliminary step allowing to evaluate or test the expert system rules. Another direction is to study if restrictions on the property formulation could make the rule definition easier and make the property instantiation automatic.

REFERENCES

Ahmad, A., Bouquet, F., Fourneret, E., Le Gall, F., and Legeard, B. (2016). Model-based testing as a service for iot platforms. In Margaria, T. and Steffen, B., editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 727–742, Cham. Springer International Publishing.

Beschastnikh, I., Brun, Y., Abrahamson, J., Ernst, M. D., and Krishnamurthy, A. (2015). Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering*, 41(4):408–428.

Beschastnikh, I., Brun, Y., Ernst, M. D., and Krishnamurthy, A. (2014). Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, New York, NY, USA. ACM.

Biermann, A. and Feldman, J. (1972). On the synthesis of finite-state machines from samples of their behavior. *Computers, IEEE Transactions on*, C-21(6):592–597.

Chaabouni, N., Mosbah, M., Zemmari, A., Sauvignac, C., and Faruki, P. (2019). Network intrusion detection for iot security based on learning techniques. *IEEE Communications Surveys Tutorials*, 21(3):2671–2701.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). Nusmv 2: An opensource tool for symbolic model checking. In Brinksma, E. and Larsen, K. G., editors, *Computer Aided Verification*, pages 359–364, Berlin, Heidelberg. Springer Berlin Heidelberg.

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 411–420.

ENISA (2017). Baseline security recommendations for iot in the context of critical information infrastructures, https://www.enisa.europa.eu/publications/baseline-security-recommendations-for-iot, technical report.

ETSI (2015). Methods for testing & specification; risk-based security assessment and testing methodologies, https://www.etsi.org/, technical report.

Ge, M., Hong, J. B., Guttmann, W., and Kim, D. S. (2017). A framework for automating security analysis of the internet of things. *Journal of Network and Computer Applications*, 83:12 – 27.

Groz, R., Li, K., Petrenko, A., and Shahbaz, M. (2008). Modular system verification by inference, testing and reachability analysis. In Suzuki, K., Higashino, T., Ulrich, A., and Hasegawa, T., editors, *Testing of Software and Communicating Systems*, pages 216–233, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gutirrez-Madroñal, L., La Blunda, L., Wagner, M. F., and Medina-Bulo, I. (2019). Test event generation for a fall-detection iot system. *IEEE Internet of Things Journal*, 6(4):6642–6651.

Holzmann, G. (2011). *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition.

Khan, M. A. and Salah, K. (2018). Iot security: Review, blockchain solutions, and open challenges. *Future Generation Computer Systems*, 82:395 – 411.

Lally, G. and Sgandurra, D. (2018). Towards a framework for testing the security of iot devices consistently. In Saracino, A. and Mori, P., editors, *Emerging Technologies for Authorization and Authentication*, pages 88–102, Cham. Springer International Publishing.

Maksymyuk, T., Dumych, S., Brych, M., Satria, D., and Jo, M. (2017). An iot based monitoring framework for software defined 5g mobile networks. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, IMCOM17, New York, NY, USA. Association for Computing Machinery.

Mariani, L. and Pastore, F. (2008). Automated identification of failure causes in system logs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 117–126.

Matheu Garcia, S. N., Hernndez-Ramos, J., and Skarmeta, A. (2019). Toward a cybersecurity certification framework for the internet of things. *IEEE Security & Privacy*, 17:66–76.

Matheu-García, S. N., Ramos, J. L. H., Gómez-Skarmeta, A. F., and Baldini, G. (2019). Risk-based automated assessment and testing for the cybersecurity certification and labelling of iot devices. *Computer Standards & Interfaces*, 62:64–83.

Mohsin, M., Anwar, Z., Husari, G., Al-Shaer, E., and Rahman, M. A. (2016). Iotsat: A formal framework for security analysis of the internet of things (iot). In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 180–188.

Nadir, I., Ahmad, Z., Mahmood, H., Shah, G., Shahzad, F., Mujahid, M., Khan, H., and Gulzar, U. (2019). An auditing framework for vulnerability analysis of iot system. pages 39–47.

NIST (2018). Framework for improving critical infrastructure cybersecurity, version 1.1, https://doi.org/10.6028

OWASP (2003). Owasp testing guide v3.0 project, http://www.owasp.org/index.php/category: owasp_testing_project#owasp_testing_guide_v3.

Petrenko, A. and Avellaneda, F. (2019). Learning communicating state machines. In *Tests and Proofs - 13th International Conference, TAP 2019, Held as Part of the Third World Congress on Formal Methods 2019, Porto, Portugal, October 9-11, 2019, Proceedings*, pages 112–128.

”Red-Hat-Software” (2020). The business rule management system drools, https://www.drools.org/, march 2020.

Salva, S. and Blot, E. (2019). Reverse engineering behavioural models of iot devices. In *31st International Conference on Software Engineering & Knowledge Engineering (SEKE)*, Lisbon, Portugal.

Salva, S. and Blot, E. (2020a). Cktail: Model learning of communicating systems. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020, Prague, CZECH REPUBLIC, May 5-6, 2020.*

Salva, S. and Blot, E. (2020b). Verifying the application of security measures in iot software systems with model learning, companion site. (Date last accessed march 2020).

Siby, S., Maiti, R. R., and Tippenhauer, N. O. (2017). Iotscanner: Detecting and classifying privacy threats in iot neighborhoods. *CoRR*, abs/1701.05007.

Tappler, M., Aichernig, B. K., and Bloem, R. (2017). Model-based testing iot communication via active automata learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 276–287.

Wilson, J., Wahby, R., Corrigan-Gibbs, H., Boneh, D., Levis, P., and Winstein, K. (2017). Trust but verify: Auditing the secure internet of things. pages 464–474.

Zhang, Z.-K., Cho, M. C. Y., and Shieh, S. (2015). Emerging security threats and countermeasures in iot. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS15, pages 1–6, New York, NY, USA. Association for Computing Machinery.