Avoiding Security Pitfalls with Functional Programming: a Report on the Development of a Secure XML Validator

Damien Doligez*, Christèle Faure†, Thérèse Hardin†‡ and Manuel Maarek†§
*Inria, Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France
†SafeRiver, 9 bis rue Delerue, 91120 Montrouge, France
‡UPMC, 4 Place Jussieu 75005 Paris, France
§Heriot-Watt University, EH14 4AS Edinburgh, UK

***Abstract*—While the use of XML is pervading all areas of IT, security challenges arise when XML files are used to transfer security data such as security policies. To tackle this issue, we have developed a lightweight secure XML validator and have chosen to base the development on the strongly typed functional language OCaml.**

The initial development took place as part of the LaFoSec Study which aimed at investigating the impact of using functional languages for security. We then turned the validator into an industrial application, which was successfully evaluated at EAL4+ level by independent assessors.

In this paper, we explain the challenges involved in processing XML data in a critical context, we describe our choices in designing a secure XML validator, and we detail how we used features of functional languages to enforce security requirements.

***Index Terms*—Security, Software Engineering, Functional Programming, XML Security**

I. Introduction

The XML format has reached every field of IT thanks to its interoperable and extensible nature. But the use of XML technologies and libraries implementing XML creates challenges in critical systems for which security is a priority. For instance, the extensibility features of XML are certainly to be restricted when used in a critical context. Critical systems using XML as a communication medium need to secure the processing of XML data files. To address this need we have developed a secure XML validator which allows the processing of data files only if they are well-formed, otherwise the processing is forbidden. We have developed this XML validator using a strongly typed functional language as we believe that such a language provides the software with valuable security properties.

We chose the OCaml language for the following reasons: (a) the language is a strongly typed functional language with high execution efficiency, (b) the OCaml compiler is able to produce directly native-code providing better security and efficiency, (c) in-house expertise in the language was available.

The development of this secure XML validator was done in two stages.

Firstly, a prototyping phase took place during the LaFoSec Study (see below). This phase was intended to demonstrate the security benefits of a functional language-based development. In this paper, we name XSVGen-prototype the prototype resulting from this first phase[1].

Secondly, we launched an industrialisation phase which specialised the initial version to fit the requirements of an industrial customer. We name here this specialised version XSVGen-product.

Both versions of the application have been evaluated. During the LaFoSec Study, XSVGen-prototype was submitted to a vulnerability assessment by independent parties. XSVGen-product was evaluated by independent assessors as part of an EAL4+ system.

The LaFoSec Study

The LaFoSec Study [1] followed the JavaSec Study "Java and security" [2] with the intention to raise awareness about the impact on security that a programming language could have (see [3] for more details). The LaFoSec Study investigated the impact of functional languages on security. It was funded and initiated by the French Network and Information Security Agency (ANSSI) and conducted by a consortium led by SafeRiver[2] and composed of academic and industrial partners. The first part of LaFoSec Study was dedicated to the analysis of the security of three functional languages (OCaml, F# and Scala) and an extended analysis of OCaml's compiler and execution mechanism. The second part was an experiment of the development and evaluation of a functional language application, XSVGen-prototype. The authors of this paper were involved in both parts of LaFoSec Study.

Contributions

The contributions of the work presented in this paper can be summarised as follows.

1) We describe the challenges, choices, outcomes of our experience in developing a security-oriented application in a functional language.

[1] The sources of XSVGen-prototype were made available by ANSSI under the CeCILL-B licence at https://github.com/ANSSI-FR/xsvgen.

[2] http://www.safe-river.com/

Preprint ICSE Software Engineering In Practice (SEIP) 2015. The definitive version was published by IEEE in the Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), DOI: 10.1109/ICSE.2015.149

2) We detail how several functional language features such as types, pattern matching and encapsulation could be used to enforce security requirements and to facilitate the prototyping and specialisation of the application.
3) We explain the challenges involved in processing XML data in a critical context and present the design choices we made for our XML validator to protect against most XML-based attacks.

Plan

The structure of this paper is as follows. In section II, we present the security issues related to XML processing, the requirements for the application and our design choices. In section III, we detail how we used language-based features to address security concerns. In section IV, we explain how we used language-based features to manage the prototyping and the specialisation of the application. In section V, we give information about the outcome of the evaluation process and discuss the feasibility of a comparison with other XML validators.

II. Design-level Security

A system or component taking an XML file as input may contain XML-related weaknesses and may therefore be subject to XML-related attacks. Table I lists Common Attack Patterns[3] (CAPEC) and Common Weaknesses Enumeration[4] (CWE) related to XML processing.

TABLE I
List of XML-related common attacks and weaknesses

Danger	Description
CAPEC-82	Violating Implicit Assumptions Regarding XML Content (aka XML Denial of Service (XDoS))
CAPEC-99	XML Parser Attack
CAPEC-146	XML Schema Poisoning
CAPEC-147	XML Ping of the Death
CAPEC-197	XML Entity Expansion
CAPEC-201	XML Entity Blowup
CAPEC-219	XML Routing Detour Attacks
CAPEC-221	XML External Entities
CAPEC-228	DTD Injection
CAPEC-229	XML Attribute Blowup
CAPEC-230	XML Nested Payloads
CAPEC-231	XML Oversized Payloads
CAPEC-250	XML Injection
CAPEC-484	XML Client-Side Attack
CAPEC-491	XML Quadratic Expansion
CAPEC-528	XML Flood
CWE-91	XML Injection (aka Blind XPath Injection)
CWE-112	Missing XML Validation
CWE-611	Improper Restriction of XML External Entity Reference ('XXE')
CWE-776	Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')
CWE-827	Improper Control of Document Type Definition

An XML attack consists in forming an XML document to elude the security mechanisms of a system. The impacts of such attacks on the system are:

- Denial of Service;
- Information disclosure;
- Diversion from intended purpose.

A. XML Validator Security Role

The role of an XML validator is to protect from XML-related attacks a critical system that makes use of XML files. An XML validator prevents forged dangerous documents from reaching the core function of the system. XML validators operate at syntactic level only and could therefore only identify syntactically dangerous documents. Documents syntactically sane could well be dangerous and must be detected by other tools acting at a semantic level according to the system data processing.

In this paper, we focus on the detection of syntactic or grammatical attacks. We define an XML validator as an XML parser that answers "fail" if the XML file is syntactically or grammatically dangerous or "pass" if it is not. Thus, before using an XML file, the system uses the validator to verify that the file is well-formed or not. If the file is well-formed, the system processes it, otherwise the file is rejected without any processing. This basic behaviour can be adapted to provide more diagnostics on errors or validation.

An XML data file uses XML syntax and XML constructs according to an *XML schema* (i.e. a grammar for XML data). XML schemas are described using languages such as DTD (Document Type Definition), XSD (XML Schema Definition) or Relax NG (REgular LAnguage for XML Next Generation). Thus the syntactic and grammatical dangerousness of an XML file has three possible origins: it is ill-formed with respect to the XML syntax; it makes use of intrinsically dangerous XML constructs; or it is ill-formed with respect to an *XML schema*.

An XML data file is considered valid if it can be parsed according to the given XML schema, the lexing step before parsing having verified conformance to the XML syntax and having imposed the restrictions on the XML constructs used.

It does not necessarily mean that this data is secure for the specific domain of the system. In a critical setting, it is the duty of the designer of the XML schema to embed as much domain-specific constraints as possible, identify constraints that cannot be described in the schema and delegate the verification of this subset of constraints to another component in the system.

The validator takes two files as input: an XML schema and an XML file and produces the validation result as output. The output can take various forms depending on the need:

- A basic dual signal output (0 or 1) indicating if the data is valid or invalid;
- A comprehensive result either stating that the data is valid or giving explanatory reasons why the data is deemed invalid;
- A verbose output providing details on the steps taken by the validation process whether it resulted in the data being considered valid or invalid, i.e. a demonstration accompanying the validation result.

In order to ensure that the validator does not introduce any security vulnerability, thus becoming the weakest link of the

[3] https://capec.mitre.org/
[4] https://cwe.mitre.org/

whole processing, it must itself be developed in conformity with security requirements:

- Not be subject to false negatives. Incorrect files should not be accepted.
- Limit false positives. Correct files should not be rejected. Otherwise, the responses of the system to its inputs can arrive too late, can become harmful if the functionality of the system is degraded, can even produce some denial of service.
- Guarantee the integrity of the input files (the input file should not be altered by the validator since if valid, it is processed by further components of the system).
- Guarantee the confidentiality of the input files. Not only the contents of the file should remain confidential but all information such as logging of validation activity or diagnosis of invalidity can be exploited to attack the system.
- Execute safely in reasonable time.

For XSVGen, we have chosen to describe XML schemas using W3C's XSD language because it is widely accepted, it is more recent and expressive than DTD and it is more commonly used than Relax NG.

B. Design Choices

A major design choice we made for XSVGen and which we detail in section II-B1 is to conceive XSVGen as a generator of XML validators. Section II-B2 details the restriction and extensions we made when implementing XML standards in XSVGen for improving security.

1) Developing a Generator of Validators: Contrary to most XML validators, each XSVGen validator is specialised to one given XML schema (expressed in XSD). Traditional XSD-based XML validators would take as arguments XSD files describing the schema and an XML file to be validated according to this schema. The choice we made for XSVGen is to generate a standalone validator for a given XML schema (described by XSD files). XSVGen therefore takes an XSD schema (one or more XSD files) as argument and produces the source code of the specialised validator. Once compiled, this validator takes as argument the file to validate. We name such a compiled validator XSVal.

In a critical setting, the XSD file describing the schema is provided by a trusted user (typically, a system administrator) while there is absolutely no guarantee on the XML data file provider (possibly an external unauthenticated user and therefore potentially an attacker). This two-phase deployment allows adjusting security checks to each of these input files.

Moreover, generating a specialised validator reduces the effort of processing the XSD files: processing the XSD files is done only once at the generation step that produces a validator. It gains efficiency and reduces the possibility of an attack targeting the XSD file since there is no occurrence of this file in the executable XML validator.

However, deploying an updated version of the XSD schema or a different XSD schema is more complex than simply changing XSD files as it requires producing a new executable XML validator by performing the generation and compilation steps for the new version. As a result, XSVGen is well-adapted to situations where the XSD schema remains unchanged during a certain lapse of time. When the XSD files are dynamically obtained from an untrusted user, XSVGen is probably not suitable.

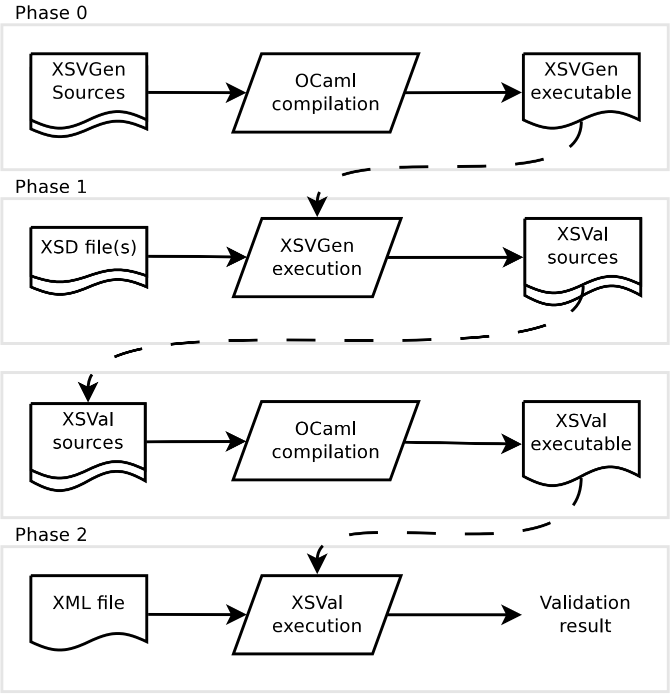

Fig. 1. XSVGen workflow
The executable XSVGen is first compiled (phase 0). An executable validator XSVal is generated and compiled for a given XML schema described by XSD files (phase 1). The validator is executed on each XML file to be validated (phase 2).

Figure 1 illustrates the deployment workflow of XSVGen. Given an XML schema (described by XSD files) as input, XSVGen outputs the OCaml sources of a validator of XML files according to this XML schema. These sources are composed of an XML validation engine (this part is shared by all generated validators) and a validation table produced by XSVGen. The validation engine is composed of a set of validation functions that are not specific to the input XML schema. The validation table is a translation of the XML schema into a flattened and preprocessed format ready to be used by the validation engine. This table is composed of values of a predefined data type. The table value is generated into a single OCaml source file and is the only source code effectively generated by XSVGen (first step of Phase 1 in Figure 1). The rest of any output validator source code is generic and could therefore be analysed and evaluated before any deployment.[5] The table value does not hold any functional

[5] In certain critical settings, a re-evaluation might be triggered by substantial source code changes. While a change of the XSD schema implies a change of the preprocessed validation table, such change does not introduce any new executable code.

value or application of function. This means that the compilation (second step of Phase 1 in Figure 1) of any validator will result in exactly the same code part of the executable file, the only difference will be in the data part of the executable file which contains the preprocessed parsing table.

We use a native-code OCaml compilation (as opposed to a bytecode OCaml compilation) to obtain a standalone executable validator which contains the validation mechanism, the internal representation of the XML schema and the OCaml runtime system. A standalone executable file provides additional protection at execution time as there is no need to protect the XSD file from alteration or access (this is particularly the case when Phase 1 and Phase 2 are performed in different locations).

2) Restricting XML Standards: The XML standards (XML [4], XML Namespace [5], XSD [6], [7]) were designed by the W3C with the intention to be general, interoperable and extensible. Such goals leave many doors open to possible attacks. In this section, we list the various accommodations we made to these standards to achieve a comprehensive level of security. These accommodations span from the management of the XML syntax to restrictions and extensions of the XSD language.

1) *Limiting the set of XML entities.* In XML, a sequence of characters could be represented by the characters themselves, by references to their numeric representations or by a reference to an entity. An entity is either predefined, user-defined in the XML schema or user-defined directly in the XML file. Considering the possibilities of uncontrolled expansion caused by XML entities, we have limited their use to the five predefined XML entities and to non-ASCII characters references. We have ruled out user-defined entities (in XSD by the Entity construction and in XML by the DocType preamble). This limitation is similar to the expansion of entities for a normal form of XML as proposed in Canonical XML [8].
2) *Forbidding XML comments.* The syntax of comments is often a source of variation in interpretations of the same file by different parsers of the same language. To avoid such issues with XML parsing, we chose as default behaviour for XSVGen to reject comments in XML files.
3) *Forbidding document type self-declaration.* The XML syntax provides the possibility for an XML document to contain the declaration of its own document type in the form of an embedded or referenced DTD, or of references to XSD files. We forbid the embedding or reference to a DTD and choose to restrict XSD declarations to a predefined URI.
4) *Restricting XSD modularity to local XSD.* An XML schema could be obtained from various XSD files using the modular XSD constructions of importation and inclusion. The XSD recommendations allow such modular composition of XSD files to be built with XSD spanning across the network using URIs to locate each file. In XSVGen, we have restricted the modular composition of XSD to local XSD files only. Files referenced by URIs need to be provided locally for the generation of a validator to take place. Additionally, we deemed insecure the referencing of external content within an XML file. It is therefore not a feature of XSVGen.
5) *Complementing XSD with size controls.* The XSD language provides ways to define an XML structure. XML is a tree representation and XSD allows defining the content of leaves (Simple-Types [7] define textual content that could appear as attribute values or in between elements' tags) and the content of nodes (Complex-Types [6] define the number, ordering and variety of children of a node). The designer or administrator of a system uses these XSD types to define the structure and content of XML data to be communicated. While XSD is an expressive language, it also offers extensive freedom that could have security implications. By default, XSD does not limit the nesting of some constructs which could trigger Denial of Service attacks. The list of XSD-valid constructs that need to be bounded in a critical environment is as follows:
   - Leaf data type,
   - Number of children for a node,
   - Loop in the definitions of Complex-Types (cyclic references of Complex-Types),
   - Wild-card XSD constructs:
     - Setting the processing to accept trailing or repeated white space characters,
     - Wild-card XSD construct authorising undefined attributes,
     - Wild-card XSD construct authorising any sequence of characters,
     - Wild-card XSD construct authorising a subtree to be populated by unknown elements from other name-spaces.

These permissive features allow a wide range of arbitrary high-length data contents or high-depth XML trees. They could be capped by setting, for instance, the XSD `maxLength` and `maxOccurs` limits for each simple and complex type defined or by avoiding to use the wild-card constructs. However, making sure that an XSD file contains the necessary restrictions in place requires scrutiny. In XSVGen, we made the choice to extend the user control over XML limits that could not be easily characterised and reviewed in XSD. We created a set of parameters for controlling the:

- Depth-limit: maximum element depth of the XML tree;
- Contents-limit: maximum number of content children of a single element;
- Length-limit: maximum byte-length of a single textual data;
- Attributes-limit: maximum number of attributes of a single element.

These controls extend XSD with limits dictating the global size and shape of a valid XML file.

Note that this list of accommodations to the XML standards are limiting XSVGen to security-oriented uses, making XSVGen less suitable for uses without security concerns.

C. Security Improvement

The protections offered by a well-designed XML validator to validate XML files before use are summarised in Tables II and III. The use of XSVGen protects from most XML attacks and weaknesses listed in public common enumerations.

TABLE II
LIST OF PROTECTIONS OFFERED BY XSVGEN

Protection	Description and dangers covered
Use of XSVGen	*Validating XML data before processing* CAPEC-82 CAPEC-99 CAPEC-219 CAPEC-250 CAPEC-484 CWE-91 CWE-112
Design choice II-B1	*Compiling a standalone specialised validator* CAPEC-146
Design choice II-B2.1	*Limiting the set of XML entities* CAPEC-197 CAPEC-201 CAPEC-221 CAPEC-491 CWE-91 CWE-611
Design choice II-B2.2	*Forbidding XML comments* CWE-91
Design choice II-B2.3	*Forbidding document type self declaration* CAPEC-228 CWE-611 CWE-776 CWE-827
Design choice II-B2.4	*Restricting XSD modularity to local XSD* CAPEC-146
Design choice II-B2.5	*Complementing XSD with size controls* CAPEC-229 CAPEC-230 CAPEC-231 CAPEC-528

TABLE III
LIST OF DANGERS COVERED BY THE PROTECTIONS OFFERED BY XSVGEN

Danger	Protection
CAPEC-82	Use of XSVGen
CAPEC-99	Use of XSVGen
CAPEC-146	Design choices II-B1 and II-B2.4
CAPEC-147	NA
CAPEC-197	Design choice II-B2.1
CAPEC-201	Design choice II-B2.1
CAPEC-219	Use of XSVGen
CAPEC-221	Design choice II-B2.1
CAPEC-228	Design choice II-B2.3
CAPEC-229	Design choice II-B2.5
CAPEC-230	Design choice II-B2.5
CAPEC-231	Design choice II-B2.5
CAPEC-250	Use of XSVGen and XSD design
CAPEC-484	Use of XSVGen
CAPEC-491	Design choice II-B2.1
CAPEC-528	Design choice II-B2.5
CWE-91	Use of XSVGen, design choices II-B2.1 and II-B2.2
CWE-112	Use of XSVGen
CWE-611	Design choice II-B2.1 and II-B2.3
CWE-776	Design choice II-B2.3
CWE-827	Design choice II-B2.3

III. LANGUAGE FEATURES FOR SECURITY

In this section, we present some aspects of the design and implementation of XSVGen and more precisely how we relied on some features of the OCaml language [9] to implement the security features of the application. This section extends [10], which focused on illustrating the use of the OCaml language as a formal Integrated Development Environment (IDE).

A. Types for Traceability

During the validation of an XML file with respect to XSD files (representing an XML schema) the various constructs of the XSD and XML languages are explored. The validation process is decidable but the number of XML and XSD constructs, their combination and the number of validation rules specified in the W3C Recommendations increase the complexity of both the development and evaluation of the validator. The development of our XML validator was based on development rules elaborated by the LaFoSec Study [1, Recommandations]. For example, the use of record and union types and corresponding pattern-matching is strongly recommended for data management as it enables automatic verifications by the compiler. Indeed we heavily relied on OCaml union types and record types not only to encode data but also to record the application of some W3C validation rules. We can say that a significant portion of the task of verifying the conformance of the validator to the W3C recommendations was done by the OCaml compiler itself through typing and pattern-matching. The OCaml compiler systematically verifies that each operation defined by pattern-matching on the values of these types covers every possible case. This alleviates the burden of verification from the developer and improves the robustness of the validator. Applying such development rules provides automatic verification by the compiler of the complete handling of:

- XML constructs,
- XSD constructs,
- XSD validation rules (CVC),
- Predefined validation and execution errors.

1) Errors and Exceptions: We have defined a union type `Error.t` for which each type constructor represents a unique predefined error. We also defined a unique OCaml exception `Error.E` which takes a value of `Error.t` as argument.

```
(* file error.ml *)
type t =
| UTF8_invalid
| UTF8_overlong
| ...
| XMLL_encoding_missing
| XMLL_encoding_not_supported
exception E of t
```

We enforced the use of the exception in `Error.E` as sole exception in the source code of XSVGen. In OCaml, while it is possible to define a function over each constructor of a union type by pattern matching and to have the completeness verified by the OCaml compiler, the completeness of exception handling is not verified by the compiler (since the type `exn` of exceptions is extensible). Catching exceptions by matching exception constructors of the type `exn` is not done exhaustively since the list of values of type `exn` that could be raised at a given location in the code is not known at compile time. Since we were interested in checking the exhaustive handling of exceptions, we have defined and used throughout the source code of XSVGen a single exception taking as argument a value of `Error.t`.

This allows to a systematic discrimination between error cases as shown by the following example

```
try
  ...
with
| Error e ->
  begin
    match e with
      | UTF8_invalid
      | UTF8_overlong
      | ... ->
        exit 1
      | XMLL_encoding_missing
      | XMLL_encoding_not_supported
      | ... ->
        exit 2
  end
```

The complete handling of error cases is therefore guaranteed by the pattern matching verification performed by the OCaml compiler.

2) Constraining Data and Computations with Types: Whenever possible, we used the properties of OCaml types to enforce constraints on the manipulated data and on their processing.

For example, the abstract type we have defined to represent XML trees is a record type composed of an XML declaration and a root node. We defined an XML node as being composed of an element and a list of contents, and a content as being either a node or a leaf. In our program, each function returning a value of type `Xml.tree` is therefore guaranteed to have exactly one and only one root and to be a well-formed tree with an element at each node.

```
(* file xml.ml *)
type 'element tree =
    { declaration : declaration;
      root : 'element node
    }
and 'element node =
    { node : 'element;
      contents : 'element content list }

and 'element content =
  | Node of 'element node
  | Leaf of Stringdata.t
```

Elements' name, namespace and attributes are identified by the type parameter `'element`. We then defined two types for elements.

- The type of elements for which the namespace expansion has not been performed: `unexpanded_element` is composed of an un-expanded XML name (with or without prefix) and a set of attributes with un-expanded names
- The type of elements for which the namespace expansion has been performed: `expanded_element` is composed of a qualified XML name (URI – name pair) and a set of attributes with qualified XML names.

```
(* file xml.ml *)
type ('name,'attributes) element =
    { element_name : 'name;
      attributes : 'attributes }

type unexpanded_name =
    { prefix : Stringdata.t option;
      local_name : Stringdata.t }

type expanded_name =
    { uri : Stringdata.t;
      name : Stringdata.t }

type unexpanded_element =
    (unexpanded_name,unexpanded_attributes)
    element

type expanded_element =
    (expanded_name,expanded_attributes)
    element
```

We use these types to define the namespace expansion step of the XML lexing as being a transformation from `unexpanded_element tree` to `expanded_element tree`. The implementation of this transformation is statically verified by the type checker of the OCaml compiler to systematically return a tree with qualified names.

B. Functional Approach to Control Execution Flow

As a security component, the validator is required to provide a negative validation result when provided with a file that does not comply with either the XML syntax or the given XSD (absence of false negatives). It is also required to give a positive validation result when provided with a valid file (low rate of false positives). It is therefore important to analyse the control flow of the validator to ensure that the application implements correct and complete computations.

Our choice when implementing the validation functions of XSVal was to rely solely on purely functional constructs (by using only immutable values and avoiding exceptions and side-effecting operations) for several reasons. First, being purely functional means that calls to the validation function do not depend upon the memory state. Tracing in the source code every computation step of the validation is straightforward by following the syntax. Second, the use of pure and typed functions helps to demonstrate the termination of the validation operation. If a function can be typed, then its execution either ends with a result of the intended type or loops forever. This last case has been eliminated by a simple study of recursive calls by code inspection showing a decreasing measure of the recursive argument call. We can therefore state that the XML validator supplies a validation result in every case.

As OCaml is not purely functional, we identified the imperative constructs of the language and forbid their use within the code composing the validation function. More precisely, the following features were forbidden:

- Mutable variables as they make the data flow more complex to follow by proofreading,
- Exceptions for nominal computation as they disrupt the execution flow of the program,
- Non-exhaustive pattern matching as it would introduce possible failures in the application.

To forbid such constructs, we relied on OCaml compiler safeguards which reject programs containing non-exhaustive pattern matching[6] and we have imposed some programming rules to follow. These rules could be verified by proofreading the source code. Some of them correspond to static verifications on the source code or on intermediate abstract representations which could be performed by a more elaborate compiler. As we forbid the use of exception in the validation function, every failure of the validation is explicitly implemented. We defined a type for validation results which: discriminates valid and invalid results (type constructors `OK` and `KO` of `'a result` type); references the XSD validation rule used (in the following example, the CVC rule length valid [6, B.1]); and locates its application in the XML input file (type `Stringdata.t option`).

```
if length_test (length_fun sd)
then
  Lib.OK [(Lxsd.CVC_length_valid,Some sd)]
else
  Lib.KO [(Lxsd.CVC_length_valid,Some sd)]
```

A validation result, such as the one output by `length_test`, is of type `vresult`.

```
type vresult =
  ((Lxsd.cvc * Stringdata.t option) list,
   (Lxsd.cvc * Stringdata.t option) list)
  Lib.result
```

C. Encapsulation as Data Protection

To guarantee that neither the XML content to be validated nor the XML schema used by the XML validator are modified by the application, we needed to protect their internal data representation against alteration. In OCaml, values are immutable by default. OCaml is a statically strongly typed language which prevents the user from directly manipulating the memory with pointers and pointer arithmetic. The memory initialisation and manipulations are automatically performed and handled by the compiler and the garbage collector (GC) of the runtime system. This offers memory safety which is extended by dynamic bound checking for string and array accesses. Thus, already acquired values could not be changed if no mutable values are used.

We therefore decided to use only immutable values. Then, we were faced with an OCaml (mis)feature: the `string` values are mutable in OCaml[7]. XSVGen makes intensive manipulations of strings and we needed to retain the efficiency of native strings. To benefit from the native `String` library capabilities while at the same time forbidding the mutation of these values, we have defined a safe string module, called `Stringdata`, which encapsulates calls to the standard string library. Throughout the program, we made use of this string wrapper to safeguard the program from string alterations. This module internally uses OCaml's strings but does only provide manipulation functions that do not alter the values themselves. To guarantee that the string values manipulated by the program are not altered we have (a) used our alternative version `Stringdata` of `string`, (b) encapsulated the use of strings in the `Stringdata` module in an abstract type `Stringdata.t` (the internal representation of `Stringdata` is not accessible outside of the module itself), and (c) reinforced this encapsulation by following coding rules which prevent bypassing the modular encapsulation (forbidding the use of the `Obj` module, forbidding exception overloading; a complete list and explanation could be found in [1, Recommandations]).

Another important gain in using OCaml is the fact that data and executable code are strictly separated in the native code produced by the compiler. To reinforce this separation, we forbid the use of dynamic code loading and the use of bytecode execution which are the only situations when data could be executed. As OCaml does not have pointers, the integrity of both data and executable code are guaranteed.

IV. Prototyping and Specialisation

This section reports on our experience in using features of the OCaml language to set up a robust development process suitable for prototyping and specialisation.

An XML validator requires various components such as a character decoder, an XML lexer & parser, an XSD-based validator. Reading the schema from XSD files[8] also requires a character decoder, an XML lexer & parser and in addition an XSD transcoder (from XSD's abstract representation to XSVGen's internal validation table representation).

For the development of XSVGen-prototype, we have set up a development plan which involved several intermediate functional prototypes. Each prototype was to implement a set of validation functionalities. We developed in parallel the application's components (character decoder, XML lexer, XML parser, XSD parser, XSD Simple-Type validator, XSD Complex-Type validator, etc.). Each new prototype added new features to some or all components of the previous prototype. The first prototype of a generator/validator was only able to implement a small set of Encoding/XML/XSD features but was a full-fledged application for this small subset of features. This allowed us to validate from the first prototype the functional breakdown and component interfaces that we had designed. Using this strategy, it was also immediately possible to run unit, integration, component and system tests.

An alternative would have been to develop one component at a time. This would have delayed the tool validation to the end of the development because the XML validation feature could not have been tested before the integration phase.

A. Support Provided by the Language

The initial step of our development consisted in setting up the interfaces between the various components. We defined

[6] Warning 8 of the OCaml compiler is concerned with identifying partial match (missing cases in pattern-matching). It is activated by default and triggers a compilation error with option `-warn-error +8`.

[7] Following the LaFoSec Study [1], OCaml version 4.02 (August 2014) incorporates the `-safe-string` option which makes standard strings immutable, this behaviour may become the default in future versions.

[8] XSD files are themselves encoded in XML.

types representing the values that would transit between components. These interface types also included error handling types used to discriminate different cases of validation failure and explicitly trace current prototype limitations.

The interface types were primarily union types where each type constructor corresponds for instance to an XML construct, to an XSD construct or to a positive or negative validation result. The processing associated to each type constructor was described by pattern matching in the source code of each component.

At each new prototyping stage, we added new type constructors representing the new features to be implemented and we removed those type constructors representing negative validation results being deprecated because they just handle the limitations of the previous prototype.

With this method of encoding the prototyping features through interface types we could rely on the OCaml compiler to verify the completeness of our implementation. The OCaml compiler verifies that every pattern matching is complete according to the list of type constructors defined and raises an error locating the incomplete pattern matching and the missing patterns[9].

Relying on the standard pattern matching verification was not enough in our case due to the possibility offered by *catch-all* patterns. In pattern matching expressions, a catch-all pattern, which is represented by the underscore symbol `_`, stands for all the cases not covered by the previous patterns. For example, in a pattern matching for a type composed of three constructs (e.g. `type t = A | B | C`), a catch-all pattern could be used to discriminate every other case but the first one (e.g. `let fragile_f = function A -> ... | _ -> ...`). When adding new type constructors to an interface type, a matching containing a catch-all would still be seen as complete while the new construct might need a specific processing. Considering our previous example, the catch-all pattern might become erroneous with the introduction of a fourth construct later in the development. Indeed, nothing was known about this fourth case at the time the pattern was written. For instance, if case `D` is added to type `t` (we now have: `type t = A | B | C | D`), the typing of `fragile_f` would still hold while `D` did not exist when the body of function `fragile_f` was written.

A matching containing a catch-all is called *fragile*. During our development process we enforced the use of non-fragile pattern matchings to guarantee that the addition of new cases in a type imposes to reconsider each processing of values of this type. Non-fragile patterns make an explicit matching of each case of the user defined type. A robust version of `fragile_f` can be defined by explicitly listing all cases: `let robust_f = function A -> ... | B | C -> ...`. When later adding a new case `D` to type `t`, the OCaml compiler would require an additional matching case `| D -> ...` to the function `robust_f`.

The OCaml compiler offers the possibility to raise a warning in case of fragile pattern matching with option `-w +4`. Furthermore, the compiler can also be made to reject any use of fragile patterns with the additional `-warn-error +4` option. Compiling `fragile_f` with these options would give the following error:

```
Warning 4: this pattern-matching is fragile.
It will remain exhaustive when constructors
   ↳are added to type t.
val fragile_f : t -> ... = <fun>
Error: Error-enabled warnings (1
   ↳occurrences)
```

The use of OCaml types and the analysis performed by the type system has proven decisive for systematically covering the numerous cases of XML/XSD validation. It has also enabled the possibility of a reliable development by prototyping stages and later by specialisation developments. For each new prototype or new specialisation, the types used were either extended to take into account the additional functionality or modified to highlight the specialisation. The OCaml type system would point at every location in the code that needed an update to properly handle the new or modified functionalities. For example, the first prototype version addressed only the simple cases of XML patterns. The next versions added new patterns and new rules of increasing complexity. Later on the specialisation required a modification of the application behaviour with regard to a specific XML construct, we altered the name of the OCaml type constructor representing this XML construct and, thanks to compiler message errors, we systematically propagated this change to the entire program. Having simply to extend or modify some data types to identify the amendments to make in the program gave the assurance that updating the features of the program was conservative over the already-developed parts.

V. Outcomes

A. Evaluation

An important aspect to take into account in security application development is the security evaluation process. Security assessors investigate every aspect of the application: specification, design, implementation deployment, execution, and maintenance.

The vulnerability of the XSVGen-prototype version of the application was assessed by independent security experts as part of the LaFoSec Study. This evaluation was however not a certification, and the details of its process and results are not publicly available. The evaluation was targeting a hypothetical gateway system for which XSVGen was validating external files. The evaluation has identified an issue in the handling of command line arguments which were not properly sanitised. The issue is not OCaml-specific and reminds of the importance to sanitise every input. The evaluation has not been able to identify any flaw related to the primary purpose of the application as a validator. Nevertheless, the fact that OCaml applications have rarely been evaluated for security meant that tools for analysing its source code and compiled code are rare.

[9] Note that by default OCaml issues a warning, an extra option `-warn-error +8` is required to turn this warning into error.

Commonly used binary code analysers are of little help for the assessors who needed to rely on other methods of analysis.

The XSVGen-product version of the application was integrated into a security system that obtained a Common Criteria EAL4+ certification. No security problem was found on the product during the evaluation phase by the independent experts. The Security Target of the certification, the evaluation process and the results of the evaluation are not publicly available.

The timescale between the initial development as part of the LaFoSec Study and the integration as a security component in a security system was only two years, each phase involving a short design and development period of 5 months. Reaching an industrial and reliable level in such a short period of time was seen as an important achievement. We consider that the industrial success of the development of XSVGen is due in part to the appropriate programming language choice (augmented by the LaFoSec development rules [1, Recommandations]), as this paper demonstrates.

B. Compliance to W3C Recommendations

XML and XSD are complex descriptive languages which are defined in details by the W3C. The XML Recommendation [4] specifies the XML syntax and provides a set of constraints characterising a well-formed XML document and a valid XML document. Similarly, the W3C Candidate Recommendation [6], [7] for the XSD language specifies the XSD constructs and provides a set of Validation Rules to determine if an XML document is valid according to a given XSD file. Additionally, a set of Schema Representation Constraints and Schema Component Constraints are defined to determine if an XSD file is valid.

To trace the application of these rules by XSVGen we made use of the values of union types representing each rule (as we saw in Sections III-A and III-B). This practice made it possible to identify the role each source code chunk plays in the validation process and therefore to verify the compliance to the W3C recommendations.

C. Comparison with Other XML Validators

There exists a number of XML validators based on one or more languages (XSD, DTD, Relax NG, ...) to describe the expected structure of the XML files. Most of these applications implement the widest range of features of XML. Several works have been made to compare and benchmark various XML implementations [11], [12].

The first phase of the development of XSVGen (XSVGen-prototype) was intended to test the feasibility to develop a secure application in OCaml. A comparison with other XML-XSD validators could be drawn but has limitation due to the prototype nature of XSVGen-prototype. Not every XML and XSD features is implemented in XSVGen-prototype for two reasons. First, some features in the XML XSD specification were explicitly ruled out or altered due to their intrinsic insecurity (see section II-B2). Second, some other features were ruled out to ensure the feasibility of such a development in a short time. As a matter of priority for the project, having a functioning application which could be evaluated with regard to security was more important than having the whole of the XML XSD Recommendations covered. We made a choice to keep the XSD XML features that are most widely used in well-established XSD-defined formats. As a result, a reliable comparison with other XML validators was not possible. We nevertheless used the W3C XML Test Suites [13] and W3C XSD Test Suite [14] to test XSVGen-prototype. The results of these tests showed a 100% success for XML tests (limited to the XML subset we presented in section II-B2). We grouped the XSD tests by language constructs and selected some 74000 tests that matched the XSD constructs implemented by XSVGen-prototype. The results showed 92% success for the handling of XSD files (between 50% and 100% for individual groups) and 99% success for the handling of XML files (between 82% and 100% for individual groups). The lower figure for the handling of XSD files is due to the fact that the validation of the input XSD files was not a requirement of XSVGen-prototype. We conducted a review of some 5000 XSD tests which indicated that the range of results per individual groups is mostly due to difficulties in reducing the discrepancy between the XSVGen-prototype requirements and the test selected among the tests of the W3C XSD Test Suite.

For the industrial specialisation XSVGen-product, the strict compliance with the W3C Recommendations was not anymore a requirement. As a result this later development bears limited specification in common with compliant XML-XSD validators. For this reason, a strict comparison would not produce significant results.

VI. Conclusion

The experience we share in this paper is the development of XSVGen: a secure XML validator implemented in OCaml. While this paper has a security focus on the one hand and a functional programming focus on the other, the two aspects are related because we made extensive use of the semantics of OCaml's language features to fit the security requirements of XSVGen. Such demonstrated semantical properties enforce software robustness and security at the source level (no format string attack, no null pointer, no uninitialised variable, no memory leak, no double free), at the compilation level (no improper accesses to values, no non-conform data processing, no break in data and code confinement, no partial declarations), and at execution level (no buffer overflows). The addition of design and coding rules to these intrinsic protections eliminates many residual safety and security dangers. We also described how these features could deliver valuable feedback for a reliable incremental development (from prototyping to specialisation). Finally, this experience demonstrated the industrial relevance of a functional language such as OCaml with strong typing, a powerful module system, automatic memory management and a direct compilation to binary code instead of bytecode.

REFERENCES

[1] ANSSI, "LaFoSec: Sécurité et langages fonctionnels," ANSSI, the French Network and Information Security Agency, Tech. Rep., 2013, documents available in French at http://www.ssi.gouv.fr/fr/anssi/publications/publications-scientifiques/autres-publications/lafosec-securite-et-langages-fonctionnels.html.

[2] ——, "JavaSec: Sécurité et langage Java," ANSSI, the French Network and Information Security Agency, Tech. Rep., 2010, documents available in French at http://www.ssi.gouv.fr/fr/anssi/publications/publications-scientifiques/autres-publications/securite-et-langage-java.html.

[3] É. Jaeger, O. Levillain, and P. Chifflier, "Mind your Language (s) – A discussion about languages and security (Long Version)," in *LangSec*, 2014.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan, "Extensible Markup Language (XML) 1.1 (Second Edition)," W3C Recommendation, http://www.w3.org/TR/2006/REC-xml11-20060816, Aug. 16 2006.

[5] T. Bray, D. Hollander, A. Layman, R. Tobin, and H. S. Thompson, "Namespaces in XML 1.0 (Third Edition)," W3C Recommendation, http://www.w3.org/TR/2009/REC-xml-names-20091208/, Dec. 08 2009.

[6] S. Gao, C. M. Sperberg-McQueen, and H. S. Thompson, "W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures," W3C Candidate Recommendation, Jul. 21 2011, disponible en ligne http://www.w3.org/TR/2011/CR-xmlschema11-1-20110721/.

[7] D. Peterson, S. Gao, A. Malhotra, C. M. Sperberg-McQueen, and H. S. Thompson, "W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes," W3C Candidate Recommendation, Jul. 21 2011, disponible en ligne http://www.w3.org/TR/2011/CR-xmlschema11-2-20110721/.

[8] J. Boyer, "Canonical (XML) 1.0," W3C Recommendation, http://www.w3.org/TR/xml-c14n, Mar. 15 2001.

[9] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, "The Objective Caml system release 3.12 – Documentation and user's manual," INRIA, Jun. 2010.

[10] D. Doligez, C. Faure, T. Hardin, and M. Maarek, "Experience in using a typed functional language for the development of a security application," in *F-IDE*, ser. EPTCS, C. Dubois, D. Giannakopoulou, and D. Méry, Eds., vol. 149, 2014, pp. 58–63.

[11] D. Barbosa, I. Manolescu, and J. X. Yu, "XML benchmarks," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer US, 2009, pp. 3576–3579.

[12] S. Chilingaryan, "The XMLBench Project: Comparison of Fast, Multi-platform XML libraries," in *Database Systems for Advanced Applications, DASFAA 2009 International Workshops: BenchmarX, MCIS, WDPP, PPDA, MBC, PhD, Brisbane, Australia, April 20-23, 2009*, ser. Lecture Notes in Computer Science, L. Chen, C. Liu, Q. Liu, and K. Deng, Eds., vol. 5667. Springer, 2009, pp. 21–34.

[13] W3C XML Core Working Group and OASIS XML Conformance Technical Committee, "W3C XML Test Suites," W3C Conformance Test Suites, http://www.w3.org/XML/Test/, Aug. 2008.

[14] W3C member organizations, "XML Schema Test Suite," W3C Conformance Test Suites, http://www.w3.org/XML/2004/xml-schema-test-suite/index.html, Jul. 2011.