ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Ověřování XML dokumentů s pokročilými elektronickými podpisy
Student:	Bc. Martin Švorc
Vedoucí:	Ing. Tomáš Vaněk, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

1. Seznamte se s moderními formáty pro podepisování a dlouhodobé ukládání elektronických dat, zejména ETSI TS 101 903 (XAdES).
2. Navrhněte a zrealizujte softwarový nástroj pro validaci dokumentů ve formátu XAdES, zaměřte se na baseline profil dle normy ETSI TS 103 171, zejména ve variantě B a T. Nástroj bude zejména analyzovat obálku a validovat podepsané struktury, jeho výstup bude standardizován dle normy ETSI TS 102 853. Implementujte vhodné rozhraní pro práci s nástrojem, například pomocí webových služeb. Diskuse a volba implementační platformy je součástí práce.
3. Funkčnost nástroje důkladně otestujte na sadě testovacích dokumentů, které dodá vedoucí práce.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 1. února 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Ověřování XML dokumentů s pokročilými elektronickými podpisy

Bc. Martin Švorc

Vedoucí práce: Ing. Tomáš Vaněk, Ph.D.

3. května 2016

Poděkování

Chtěl bych poděkovat Ing. Tomáši Vaňkovi, Ph.D. za vedení této práce. Také bych chtěl poděkovat svým nejbližším za trpělivost a za jejich podporu po celou dobu studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 3. května 2016 .....

Odkaz na tuto práci

Švorc, Martin. *Ověřování XML dokumentů s pokročilými elektronickými podpisy.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato diplomová práce se zabývá ověřováním elektronických podpisů v pokročilém formátu XAdES. Důraz je kladen především na *baseline* profil podpisů a standardizovaný proces jejich ověření. Práce nejprve představuje normativní podklady, na jejichž základě je navrženo a implementováno jádro nástroje pro validaci a verifikaci těchto podpisů. Pro ověřování je vytvořena webová služba, která jádro využívá. Implementace je nakonec otestována pomocí scénářů z oficiálního ETSI Plugtestu.

Klíčová slova XAdES, elektronický podpis, ověření podpisu, webová služba

Abstract

This master thesis deals with the validation of advanced electronic signatures in XAdES format. The emphasis is placed on the *baseline* profile of these signatures and the standardized validation process. At first, normative documents are introduced and the core of the validation tool is designed and implemented based on them. A configurable web service is created using this core. The implementation is tested using the oficial ETSI Plugtest scenarios.

Keywords XAdES, electronic signature, signature validation, web services

Obsah

Seznam obrázků

Úvod

Motivace a cíle práce

Cílem práce je vyvinout softwarový nástroj schopný ověřovat pokročilé elektronické podpisy XML dokumentů ve formátu XAdES. Taková řešení už sice existují, ale jejich podrobnější zkoumání ukazuje, že většina z nich je zaměřena hlavně na vytváření elektronických podpisů a ověřování podporují spíše okrajově. K tomu obvykle trpí jedním nebo více neduhy – nejedná se o univerzální řešení, jsou špatně rozšiřitelná a integrovatelná, nepodporují základní profily XAdES nebo jsou jejich výstupy obtížně interpretovatelné.

Hlavním výstupem práce by proto měl být nástroj, který by tyto nedostatky neobsahoval – výsledný software by měl být dostatečně obecný, aby umožňoval ověření libovolného podpisu ve formátu XAdES, zároveň by měl být snadno rozšiřitelný, a to jak z hlediska kontrol prováděných v rámci ověření, tak z hlediska interně použitých algoritmů.

Nástroj bude podporovat ověření elektronického podpisu dle předdefinovaných *profilů* (sad pravidel, které musí ověřovaný podpis splňovat, pokud má být označen za platný) zejména dle *baseline* profilu ([2]), jehož hlavním cílem je zvýšit interoperabilitu systémů omezením variablity podpisů.

Výsledky zpracování podpisu budou standardizovány a základní algoritmus ověření bude implementován dle specifikace [1], aby bylo docíleno co nejvyšší míry integrovatelnosti s konzumenty výstupů nástroje.

Aby mohly být tyto požadavky naplněny v úplné míře, omezuje zadání množinu podpisů, které musí nástroj být schopen takovým způsobem zpracovat, pouze na nearchivní formy XAdES.

V návaznosti na to by mělo vzniknout rozhraní pro pohodlnou práci s nástrojem, umožňující ověření nejen provádět, ale i parametrizovat podle aktuálních potřeb uživatele tohoto rozhraní a tím naplno využít modularity implementovaného mechanismu ověření.

Struktura práce

V kapitole 1 jsou shrnuty základní standardy a specifikace týkající se problematiky pokročilých XML podpisů. Jsou jimi standardy XML-DSig (1.1) a XAdES (1.2), specifikace základních XAdES profilů (1.3) a specifikace způsobu ověření elektronického podpisu (1.4). Její přečtení by mělo čtenáři pomoci proniknout do problematiky a pochopit základní koncepty.

V kapitole 2 jsou analyzovány požadavky pro implementaci ověřovacího nástroje (2.1) a jeho použití pomocí rozhraní webových služeb (2.2.3). Kapitola 3 popisuje existující řešení pro ověření XAdES a srovnává jejich zaměření a přínosy s cíli této práce.

Kapitola 4 popisuje prostředky zvolené k realizaci dílčích úloh, jako je práce s XML-DSig (4.2), časovými razítky (4.3), certifikáty (4.4) nebo framework pro sestavení komponent nástroje (4.5).

Na základě vydefinovaných požadavků a rešerše existujících řešení je v kapitole 5 navrženo výsledné řešení ověřovacího nástroje. Nejprve jsou popsány komponenty tohoto řešení (5.1), dále potom způsob, jakým jádro nástroje bude rozhodovat o celkovém stavu podpisu (5.2), pracovat s výjimkami (5.3) a krátce jsou zmíněny možnosti konfigurace nástroje (5.4).

Kapitola 6 se věnuje některým zajímavým implementačním problémům a způsobům, kterými nástroj přistupuje k jejich řešení (6.1, 6.2). Následně popisuje integraci jádra nástroje s frameworkem Spring (6.3) a implementaci webové služby využívající jádro nástroje (6.4).

Poslední kapitola 7 se věnuje testování jádra nástroje (7.1) a webových služeb (7.2).

Kapitola 1

Přehled důležitých standardů

V této kapitole jsou popsány standardy a normy, které se přímo týkají tématu diplomové práce. Většina z těchto norem je značně rozsáhlá, a proto je tento popis koncipován tak, aby čtenář především pochopil základní koncepty a procesy, kterým se tyto dokumenty věnují; pro jejich aplikaci v širším kontextu nebo popis detailů, který by přesahoval rozsah této práce, je čtenář odkázán na konkrétní normu.

První z důležitých dokumentů je norma popisující schéma podpisu XML dokumentu a procesy související s jeho vytvářením a ověřováním (1.1). Další dvě normy se již týkají standardu XAdES, který je rozšířením právě základního XML podpisu – jedná se o základní specifikaci standardu (1.2) a doplňkovou specifikaci *baseline* úrovní pokročilých elektronických podpisů (1.3). Posledním zpracovaným dokumentem je specifikace popisující procesy ověřování rozšířeného elektronického podpisu (1.4).

1.1 XML Signature (XML-DSig)

XML Digital signature ([3]) je doporučení[1] konsorcia W3C popisující syntaxi digitálního podpisu pomocí XML dokumentů a jeho zpracování. Standard definuje XML strukturu, která může být využita k podepsání libovolného datového objektu; může se jednat o XML dokument, jehož součástí je i podpis, nebo se může jednat o externí binární data, která jsou digitálním podpisem referencována – pak se jedná o tzv. odpojený (detached) podpis (1.1.3).

1.1.1 Struktura XML podpisu

Strukturu digitálního podpisu zachycuje ukázka 1.1-1 ([3]). Význam jednotlivých elementů je následující:

[1] Označení *doporučení* může být zavádějící, ve skutečnosti tento termín znamená uznávaný standard konsorcia [4].

- `Signature` je kořenovým elementem digitálního podpisu.
- `SignedInfo` je element, který je skutečně kryptograficky podepsán. Obsahuje popis kanonikalizační metody (způsob, jakým je tento element jednoznačně transformován na vstup algoritmu digitálního podpisu), metody podpisu (algoritmy použité k vytvoření a ověření tohoto podpisu) a především jednu či více referencí jednoznačně identifikujících podepisované datové objekty. Reference jsou podrobněji rozebrány v podsekci 1.1.1.1.
- `SignatureValue` je hodnota digitálního podpisu elementu `SignedInfo` vypočtená pomocí algoritmu specifikovaného v elementu `SignatureMethod` zakódovaná pomocí Base64.
- `KeyInfo` může obsahovat data, která umožní ověřující straně získat klíč k ověření digitálního podpisu. Podporována je celá řada metod, od přímé specifikace parametrů veřejného klíče algoritmů DSA nebo RSA přes URI umístění veřejného klíče až po přímé vložení X.509 certifikátu do elementu `X509Data`. Posledně jmenovaný způsob je jediný použitelný ve standardu XAdES [5].
- `Object` je element s libovolným obsahem. Je možné do něj například vložit podepisovaný fragment XML a na tento obalující `Object` odkázat referencí (1.1.1.1), pak se jedná o tzv. obalující podpis (1.1.3).

1.1.1.1 Element `Reference`

Skutečným cílem podepisující strany je podepsat jí zvolená data, nikoliv XML element `SignedInfo`. Děje se tak nepřímo – podepsán je pouze jejich otisk a odkaz na ně. Obě tyto položky jsou (společně s několika dalšími pomocnými informacemi) uvedeny v elementu `Reference`, který je vložen do `SignedInfo` a podepsán.

Jeho atribut `URI` identifikuje datový objekt, který je cílem reference dle RFC 3986 ([6]). Může se tedy jednat o aktuální XML dokument, jeho fragment nebo libovolný externí zdroj identifikovatelný pomocí URI. V rámci ověření podpisu musí být datový objekt identifikován a získán, v případě neúspěchu musí být celý podpis považován za neplatný (1.1.2). Proces získání datového objektu z reference se nazývá *dereferencování*.

Na takto získaný objekt mohou být aplikovány transformace – typicky se jedná opět o kanonikalizaci (pokud je cílem reference XML dokument), XPath nebo XSLT transformace (typicky v situaci, kdy se jedná o zabalený podpis, viz 1.1.3), transformace týkající se kódování atp. Transformace může mít za následek i cílenou ztrátu některých dat (například XPath transformace může z XML stromu vybrat jen určitý podstrom), tato data pak nejsou použita v průběhu výpočtu otisku a tedy ani podepsána; je proto důležité si

```
<Signature ID?>
  <SignedInfo>
    <CanonicalizationMethod/>
    <SignatureMethod/>
    (<Reference URI? >
      (<Transforms>)?
      <DigestMethod>
      <DigestValue>
    </Reference>)+
  </SignedInfo>
  <SignatureValue>
  (<KeyInfo>)?
  (<Object ID?>)*
</Signature>
```

Ukázka 1.1-1: Schéma XML podpisu. Symbol **?** za elementem či atributem označuje jeho volitelnost, elementy se symbolem + se musí vyskytnout nejméně jednou, * za elementem označuje libovolnou násobnost. Elementy bez označení se v podpisu vyskytují právě jednou.

uvědomit, že při použití transformací není podepisován původní cíl reference, ale transformovaný obsah. Transformace jsou provedeny v pořadí, v jakém jsou uvedeny v elementu `Transforms`.

`DigestMethod` specifikuje algoritmus použitý k vypočtení otisku referencovaného datového objektu (po aplikaci případných transformací), hodnota otisku je uložena v elementu `DigestValue`.

1.1.1.2 Element `Manifest`

Dalším užitečným elementem souvisejícím s referencemi je `Manifest`. Podobně jako `SignedInfo` obsahuje jednu nebo více referencí na podepisovaná data a tedy i otisk těchto dat. Některou z referencí v elementu `SignedInfo` je pak obvykle odkazováno na tento `Manifest`; otisk v této referenci je pak otiskem tohoto `Manifest`u a ne přímo samotných dat. Hodnota otisku uvnitř reference v `SignedInfo` je však na těchto datech stále závislá, neboť `Manifest` obsahuje přímo otisky těchto dat. Manifest se může v podpisu vyskytnout například uvnitř některého elementu `Object`.

Význam této konstrukce může být ozřejměn jednoduchým příkladem: XML podpis je aplikován na několik externích souborů, o nichž je předem známo, že v budoucnu mohou přestat existovat (tj. hrozí, že je nebude možné získat pomocí příslušné URI). Pokud by byly tyto soubory referencovány přímo v `SignedInfo`, pak by v případě nedostupnosti byť jediného z těchto souborů musel být celý podpis považován za neplatný, neboť by nebylo možné ověřit reference. Pokud by naproti tomu soubory byly referencovány ve vnořeném manifestu, v rámci *core verification* (1.1.2) nebude počítán otisk samotných

souborů, ale manifestu – XML podpis bude dle standardu považován za platný, i kdyby otisky souborů referencovaných manifestem neodpovídaly v manifestu uvedeným hodnotám nebo nebyly dohledatelné. Je pak záležitostí aplikace, aby tyto vnořené reference ověřila a v daném kontextu rozhodla o platnosti.

1.1.2 Ověření XML podpisu

Ověření podpisu se skládá ze dvou hlavních kroků:

Validace referencí V tomto kroku jsou nejprve dereferencovány datové objekty odkazované atributem `URI`. Na cílová data jsou aplikovány případné transformace uvedené v elementu `Transforms` příslušné reference. Z výstupu je vypočítán otisk algoritmem uvedeným v elementu `DigestMethod`. Pokud je hodnota otisku shodná s hodnotou uvedenou uvedenou v `DigestValue`, je reference považována za platnou. Tato část validace nepředepisuje, jakým způsobem mají být data dereferncována (může být např. použita lokální cache).

Validace podpisu V tomto kroku je aplikována ověřovací část algoritmu specifikovaného v `SignatureMethod` na podpis v `SignatureValue`. Veřejný klíč musí být schopna nalézt ověřující strana, v tom ji mohou pomoci informace v `KeyInfo`, to ale standard nevyžaduje. Porovnávaným vstupem pro ověřovací algoritmus je (kanonikalizovaný) element `SignedInfo`. Pokud ověřovací algoritmus vyhodnotí podpis jako platný a všechny reference byly úspěšně zvalidovány, je celý XML podpis považován za platný; v opačném případě musí být celý podpis považován za neplatný.

Souhrnně jsou tyto dva kroky označovány jako *core validation* ([3]).

1.1.3 Formy XML podpisu

Standard dále definuje tři formy XML podpisu:

Zabalený (*Enveloped*) podpis V tomto druhu podpisu je kořenový element `Signature` potomkem elementu, který sám podepisuje, tj. odkazuje na něj některá reference. Z toho vyplývá, že tato reference musí obsahovat takovou transformaci, která ze vstupního obalujícího XML stromu vyjme samotný element `Signature`, např. pomocí XPath transformace; jinak by nebylo možné zabalený podpis vytvořit, protože jeho výstup by byl závislý sám na sobě.

Obalující (*Enveloping*) podpis V tomto podpisu jsou referencovány elementy `Object` nebo jejich potomci – podepisovaný obsah se tedy nachází uvnitř podstromu, jehož kořenem je element `Signature`.

Odpojený (*Detached*) podpis Data referencovaná v `SignedInfo` se nachází mimo aktuální XML dokument (někdy se tento výraz používá i v situaci, kdy jsou podepisovány elementy aktuálního XML dokumentu, které jsou sourozenci elementu `Signature` – nejedná se pak totiž o zabalený ani o obalující podpis, viz [3]). Může se jednat o jiný XML dokument nebo libovolný datový objekt identifikovatelný pomocí URI.

V rámci jednoho podpisu se může objevit jedna nebo více forem – jedna reference může být obalující, druhá zabalující a jiná odkazovat na externí soubor.

1.1.4 Nedostatky standardu

Standard XML-DSig popisuje způsob, jakým je možné podepsat libovolně mnoho libovolných datových objektů pomocí jediného digitálního podpisu, přičemž podepisujícímu poskytuje poměrně velkou volnost v možnostech tvorby podpisu – výsledná podoba podpisu tak může být být značně variabilní. Z toho a ze spíše technického pojetí normy pak vyplývají některé její nedostatky, které mohou být problematické v praktickém nasazení:

- *Nedefinuje způsob, jakým identifikovat podepisující stranu.* V praktickém použití elektronického podpisu je obvykle důležité znát identitu podepisujícího; jinak může být podpis použit k ověření integrity podepisovaných dat, ale již nemůže naplňovat požadavek na nepopiratelnost původu. XML-DSig sice *umožňuje* podepisujícího identifikovat (např. přiložit certifikát v elementu `SignedInfo`), nepožaduje ale zabezpečení (podepsání) této informace.

- *Nepopisuje podepsaná data.* Především při použití elektronicky podepsaných dat člověkem je potřeba znát formát podepsaných dat (obecněji mít k dispozici metadata podepsaného datového objektu).

- *Nedefinuje mechanismus pro použití bezpečnostních politik*, např. omezení kryptografických algoritmů, identit podepisujících atd.

- *Nepodporuje dlouhodobou archivaci*, tedy proces ochrany podepsaných dat a možnost ověření podpisu dat i poté, co vypršela platnost původního certifikátu použitého k podepsání těchto dat.

Částečně se některé z těchto nedostatků standard snaží řešit volitelným elementem `SignatureProperties`, do kterého je možné vložit libovolné atributy podpisu ([3]). Systematičtěji k těmto nedostatkům (především k otázce dlouhodobého podepisování) přistupuje standard XAdES.

1.2 XML Advanced Electronic Signature (XAdES)

XAdES je formát elektronického podpisu za použití XML definovaný v technické specifikaci ETSI 101 903 [5]. Jeho autorem je Evropský ústav pro telekomunikační normy a standard byl vyvinut především pro potřeby naplnění požadavků Evropské unie týkajících se elektronických podpisů [7]. Je založen na využití infrastruktury veřejného klíče a digitálních certifikátů.

Standard rozšiřuje původní XML-DSig specifikaci a obohacuje ji o možnost přidat do podpisu *vlastnosti* (1.2.1), tedy bližší informace o různých aspektech tohoto elektronického podpisu. K tomuto účelu je definováno nové XML schéma popisující, jak mají být tyto vlastnosti sestaveny a jak mají být zapojeny do základního (XML-DSig) podpisu.

Dále je definováno několik úrovní pokročilého elektronického podpisu v závislosti na použitých vlastnostech – od základního elektronického podpisu až k jeho archivní formě (1.2.5) – a pravidla pro generování a ověřování podpisů, které mají splňovat kritéria dané formy.

Protože se implementační část práce soustředí především na *baseline* formy XAdES-B a XAdES-T (1.3.1), jsou v této části podrobněji popsány pouze ty části normy, které se týkají odpovídajících základních forem, tedy XAdES-BES, XAdES-EPES a XAdES-T (1.2.5).

V dalším textu je pro rozlišení elementů definovaných v rámci XML-DSig použit jmenný prostor `ds`, pro elementy definované standardem XAdES jmenný prostor `xades`.

1.2.1 Qualifying properties

Jádrem XAdES je sada metadat podpisu označovaná jako *qualifying properties*. Jelikož v českém jazyce zatím pro tento technický termín neexistuje překlad, bude pro něj v dalším textu použita pouze zkratka QP nebo obecný výraz *vlastnosti*.

Narozdíl od přístupu XML-DSig, který pouze zavádí obálku `ds:SignatureProperties`, XAdES přichází s komplexní strukturou elementů s jednoznačně definovanou sémantikou. Do podpisu tak lze vložit například metadata týkající se identity podepisující strany, jejích certifikátů, metadata podpisu nebo metadata podepsaných datových objektů.

Standard rozlišuje dvě skupiny QP – **podepsané vlastnosti** a **nepodepsané vlastnosti** podpisu. Jak napovídá jejich pojmenování, první skupina vlastností je podepsána společně s podepisovanými datovými objekty, tj. v podpisu existuje `ds:Reference`, která pro svůj vstup použije element `xades:SignedProperties` obsahující všechny podepsané vlastnosti. Naproti tomu nepodepsané vlastnosti nejsou podpisem, jehož jsou součástí, pokryty.

Samoté XSD schéma nepředepisuje povinnost žádných QP, což plyne z možnosti vkládat různé QP na různá místa (1.2.1.1). Povinnost je určena úrovní podpisu, kdy daná úroveň přítomnost některých vlastností přímo vyžaduje, jiných přímo zakazuje a začlenění ostatních nechává na vytvářející straně (1.2.5). Jejich strukturu až do úrovně XAdES-T zachycuje 1.2-2.

1.2.1.1 Umístění QP

QP mohou být v XAdES umístěny dvojím způsobem

- Přímo, uvnitř elementu `xades:QualifyingProperties`, který se nachází uvnitř některého `ds:Object` v těle podpisu.
- Nepřímo, pomocí odkazu URI v elementu `xades:QualifyingPropertiesReference`, který ukazuje na samotný element `xades:QualifyingProperties`. Vlastnosti se pak mohou nacházet ve stejném XML dokumentu mimo tělo podpisu, nebo i v jiném souboru. Samotný odkaz se ale musí nacházet uvnitř `ds:Object` základního podpisu.

Obě možnosti je v omezeném rozsahu možné kombinovat – část QP může být umístěna přímo a část nepřímo. Pokud jsou vlastnosti vloženy kombinovaně, musí být uvnitř téhož `ds:Object`; `xades:QualifyingPropertiesReference` se může vyskytovat vícekrát, `xades:QualifyingProperties` nejvýše jednou.

Při využití první možnosti jsou podepsané vlastnosti umístěny přímo v podpisu, tento XAdES je alespoň částečně podpisem obalujícím; při použití možnosti druhé může být typ podpisu libovolný. Také reference na podepisované datové objekty mohou být různého typu (1.1.3).

1.2.2 Podepsané QP

Tato skupina vlastností je podepsána stejným podpisem, jako podepisované datové objekty, vytváří je tedy podepisující strana. Strukturu podepsaných vlastností až do úrovně XAdES-T zachycuje 1.2-2.

Podepsané vlastnosti jsou rozčleněny do dvou skupin. První z nich je sada vlastností týkající se podpisu nebo podepisující strany; druhá skupina elementů umožňuje specifikovat metadata týkající se samotných podepsaných datových objektů.

1.2.2.1 Podepsané vlastnosti podpisu

Všechny zde uvedené vlastnosti jsou potomky elementu `xades:SignedSignatureProperties`.

`SigningTime` Čas, který podepisující strana prohlašuje za okamžik vytvoření podpisu. Důvěryhodnějším zdrojem informací o času vzniku podpisu je pak časové razítko TSA (1.2.4).

```
<QualifyingProperties>
  <SignedProperties>
    <SignedSignatureProperties>
      <SigningTime>
      <SigningCertificate>
      <SignaturePolicyIdentifier>
      <SignatureProductionPlace>
      <SignerRole>
    </SignedSignatureProperties>
    <SignedDataObjectProperties>
      <DataObjectFormat>
      <CommitmentTypeIndication>
      <AllDataObjectsTimeStamp>
      <IndividualDataObjectsTimeStamp>
    </SignedDataObjectProperties>
  </SignedProperties>
  <UnsignedProperties>
    <UnsignedSignatureProperties>
      (<CounterSignature>)*
      (<SignatureTimeStamp>)*
    </UnsignedSignatureProperties>
  </UnsignedProperties>
</QualifyingProperties>
```

Ukázka 1.2-2: Schéma QP úrovní BES, EPES (zeleně) a T (modře). Schéma nezachycuje povinnosti, pouze násobné elementy jsou označeny hvězdičkou.

`SigningCertificate` Tímto elementem je možné identifikovat jeden nebo více certifikátů, a to vložením otisku tohoto certifikátu (element `CertDigest`), jeho sériového čísla a názvu vydávající certifikační autority (`IssuerSerial`), v atributu `URI` se může nacházet odkaz na tento certifikát. Mezi certifikáty se musí nacházet certifikát, který má být použit k ověření podpisu (jehož soukromý klíč byl použit k jeho vytvoření); mohou se zde nacházet i další certifikáty, které jsou součástí některé certifikační cesty (těch může existovat několik, viz např. [8]) – tyto certifikáty pak musí být obsaženy v certifikační cestě, která je použita pro ověření platnosti podpisu.

Tento mechanismus zajišťuje, že certifikát použitý k podepsání může být jednoznačně identifikován; navíc se jedná o podepsanou vlastnost podpisu a proto je odolný proti substitučnímu útoku (výměna certifikátu v `ds:KeyInfo` za jiný certifikát se stejným veřejným klíčem) a eliminuje tak tento problém základního podpisu (1.1.4).

`SignaturePolicyIdentifier` Položka umožňuje specifikovat podpisovou politiku, tedy sadu pravidel pro vytváření a ověřování elektronického podpisu. Podepisující strana specifikací podpisové politiky říká, že byl podpis vy-

tvořen dle této politiky a že má podle ní být ověřován. Pokud je při ověřování podpisu zjištěno porušení politiky (nebo se politiku nepodaří identifikovat), musí být prohlášen za neplatný, i kdyby měl být bez přítomnosti politiky považován za platný. Tato položka je povinná pro úroveň podpisu XAdES-EPES (1.2.5).

`SignatureProductionPlace` Místo, které podepisující strana prohlašuje za místo své fyzické přítomnosti v okamžiku vytvoření podpisu.

`SignerRole` V tomto elementu lze specifikovat jednu nebo více rolí podepisujícího subjektu, například ve vztahu k organizaci, která vznik podpisu zaštiťuje. Tato role může být v reálné aplikaci podstatnější než samotná identita podepisujícího ([5]). Role lze specifikovat buď pomocí elementu `ClaimedRole` (jeho hodnota závisí na konkrétní implementaci, standard podobu nepředepisuje), nebo pomocí `ClaimedCertifiedRole`, který umožňuje vložit atributový certifikát ([9]).

1.2.2.2 Podepsané vlastnosti datových objektů

Tyto vlastnosti jsou potomky elementu `xades:SignedDataObjectProperties`.

`DataObjectFormat` Element obsahuje člověkem čitelná metadata podepsaného datového objektu – jedná se zejména o jeho MIME typ ([10]), informaci o kódování objektu a slovní popis. Každý element `xades:DataObjectFormat` musí také obsahovat odkaz na element `ds:Reference`, který referencuje tento podepsaný datový objekt.

`CommitmentTypeIndication` Podepisující strana zde může specifikovat své závazky, tj. jaké kroky týkající se podpisu provedla a přijímá za ně zodpovědnost (např. se doznává, že vytvořila podepisované datové objekty, nebo že objekty nevytvořila, ale pouze přejala a schválila, nebo že objekty pouze podepsala). Určit je lze pomocí URI (příklady výše jsou přebrány z [11]; konkrétní implementace standardu mohou definovat a používat vlastní URI) nebo aplikačně specifickým způsobem.

`AllDataObjectsTimeStamp` Pomocí tohoto elementu lze umístit XAdES časovou značku (1.2.4) otisku, který je vypočítán nade všemi datovými objekty referencovanými v `ds:Reference` uvnitř `ds:SignedInfo` (s výjimkou vlastních `xades:SignedProperties`, jinak by nebylo podpis možné spočítat). Lze tak do podpisu umístit důvěryhodnější informaci o okamžiku vzniku podpisu od jedné nebo více autorit časového razítka.

`IndividualDataObjectsTimeStamp` Podobně jako u předchozího elementu se jedná o XAdES časovou značku, tentokrát ale vytvořenou nad otiskem jediného datového objektu.

1.2.3 Nepodepsané QP

Element `xades:UnsignedProperties` sdružuje všechny QP, které nejsou pokryty původním podpisem podepisujícího; většinou se jedná o objekty podepsané jinými subjekty nebo jiným způsobem (bez podpisu by bylo snadné jejich obsah podvrhnout). Podobně jako u podepsaných vlastností jsou i zde rozlišeny dvě skupiny.

Do první z nich, zastřešené elementem `xades:UnsignedDataObjectProperties`, lze vložit nepodepsané vlastnosti podepsaných datových objektů. Standard nedefinuje způsob použití těchto vlastností.

Druhou skupinu tvoří nepodepsané vlastnosti podpisu, potomci elementu `xades:UnsignedSignatureProperties`. Do úrovně XAdES-T se zde mohou vyskytovat pouze dva druhy nepodepsaných vlastností ([5]).

1.2.3.1 Countersignature

Do těla položky `xades:CounterSignature` může být vložen podpis původního podpisu, konkrétně hodnoty jeho `ds:SignatureValue` [2]. Standard diktuje pro použití speciální hodnotu atributu URI tohoto vnořeného podpisu, která jej umožňuje při zpracování snadněji identifikovat jako *countersignature*.

Vnořeným podpisem může být XML-DSig nebo opět XAdES, který obsahuje v `ds:SignedInfo` referenci na hodnotu podpisu (`ds:SignatureValue`) obalujícího podpisu. V druhém z případů může tento vnořený podpis opět obsahovat jeden nebo více vlastních vnořených podpisů. Schéma ilustruje ukázka 1.2-3.

Tímto mechanismem může být docíleno toho, že podepisující není schopen v budoucnu změnit hodnotu podpisu (v situaci, kdy by se změnil některý podepisovaný objekt, tím hodnota jeho otisku a bylo by nutné vytvořit novou hodnotu podpisu).

1.2.3.2 Časové razítko podpisu

Zejména pro dlouhodobé uchovávání elektronicky podepsaných dokumentů je důležité mít důkaz o existenci podpisu v určitém okamžiku, nejlépe co nejdříve po vytvoření samotného podpisu. Pokud v budoucnu ověřující strana bude vědět, že podpis existoval k tomuto okamžiku, bude moci rozhodnou o platnosti podpisu na základě platnosti použitých certifikátů v onom okamžiku, nikoliv v okamžiku kontroly (to už mohou být certifikáty revokované nebo může vypršet jejich platnost).

XAdES k označení podpisu časovým razítkem (1.2.4) používá jeden nebo více výskytů elementu `xades:SignatureTimeStamp`. Vstupem pro vznik časového razítka je element `ds:SignatureValue`; je tak možné vložit časové razítko

[2] Použití českého termínu *kontrasignace* neboli *spolupodpis* by mohlo být zavádějící, protože má odlišný právní význam, proto práce používá původní výraz.

```
<ds:Signature>
  <ds:SignedInfo>
    <ds:Reference>
      <!-- odkaz na podepisovaný datový objekt -->
    </ds:Reference>
    <!-- ... další reference podpisu ... -->
  </ds:SignedInfo>
  <ds:SignatureValue Id="SignatureValue"> ... </ds:SignatureValue>
  <!-- další elementy podpisu -->
  <ds:Object>
    <!-- ... podepsané vlastnosti ... -->
    <xades:UnsignedSignatureProperties>
      <xades:CounterSignature>
        <ds:Signature>
          <ds:SignedInfo>
            <ds:Reference URI="#SignatureValue">
              <!-- reference na hodnotu vnějšího podpisu -->
            </ds:Reference>
            <!-- ... další reference countersignature ... -->
          </ds:SignedInfo>
      <!-- ... další elementy countersignature ... -->
      </xades:CounterSignature>
    </xades:UnsignedSignatureProperties>
  </ds:Object>
  <!-- ... další obsah podpisu ... -->
</ds:Signature>
```

Ukázka 1.2-3: Způsob začlenění *countersignature* (zvýrazněn modře)

dokazující, že podpis existoval v jím určený okamžik. Tato položka je povinná pro úroveň podpisu XAdES-T (1.2.5).

1.2.4 Časová razítka

XAdES využívá mechanismu tzv. časových razítek. Časové razítko dle [12] je podpis důvěryhodné třetí strany (označované jako *autorita časového razítka*, TSA), která poskytne důvěryhodný časový časový údaj (časovou značku), který společně s předaným otiskem podepíše svým soukromým klíčem. Strana zpracovávající orazítkovaná data potom může zjistit, zda data skutečně existovala k okamžiku vytvoření časového razítka.

Zjednodušený životní cyklus časového razítka je následující:

- Strana žádající o časové razítko pošle žádost TSA. Žádost musí obsahovat minimálně otisk razítkovaných dat.

- TSA k otisku přidá aktuální časovou značku a vypočítá podpis pomocí soukromého klíče svého certifikátu. Žádající straně pošle zpět časové razítko ve formátu trojice původní otisk, časová značka a tento podpis.

- Ověřující strana (pokud této TSA důvěřuje) ověří její podpis, srovná otisky a pokud tyto souhlasí, má důkaz, že orazítkovaná data již existovala k okamžiku časové značky uvedené v časovém razítku.

Časová razítka jsou v XAdES použita pro označení datových objektů (1.2.2.2), celého podpisu (1.2.3.2) a také k orazítkování objektů týkajících se vyšších forem XAdES ([5]).

1.2.5 Formy XAdES

Standard definuje několik úrovní podpisu v závislosti na množství informací obsažených v podpisu. Každá vyšší forma podpisu splňuje požadavky kladené na podpis nižšími formami a do podpisu přidává nové informace.

1.2.5.1 XAdES-BES

Forma **BES** (*Basic Electronic Signature*, základní elektronický podpis) je nejjednodušší formou XAdES. Jediným požadavkem této formy je zabezpečení (podpis) podepisujícího certifikátu, jehož lze dosáhnout dvěma způsoby:

- Vložením podepisujícího certifikátu do `ds:KeyInfo` a uvedením `ds:Reference` v `ds:SignedInfo`, která je sestavena takovým způsobem, že výsledný podpis podepisuje i tento certifikát.

- Použitím podepsané vlastnosti `SigningCertificate` (1.2.2.1), která obsahuje alespoň odkaz na podepisující certifikát a jeho otisk. Způsob vložení této vlastnosti forma nepředepisuje (1.2.1.1).

V podpisu mohou být volitelně obsaženy další podepsané či nepodepsané vlastnosti. Zabezpečením podepisujícího certifikátu forma naplňuje právní požadavky definované v [7].

1.2.5.2 XAdES-EPES

Vložením podepsané vlastnosti `SignaturePoliceIdentifier` se z formy BES vyvine forma **EPES** (*Explicit Policy Electronic Signature*, elektronický podpis s výslovnou politikou). Použití politiky norma nedefinuje, záleží na konkrétní implementaci standardu.

1.2.5.3 XAdES-T

Do podpisu sice může podepisující vložit vlastní specifikaci času vytvoření podpisu, ta ale nemusí být důvěryhodná (1.2.2.1). Teprve vložením důvěryhodné časové značky je podpis povýšen na formu **T** (*with Time*, s časem). Časová značka může být do podpisu vložena buď pomocí nepodepsané vlastnosti `SignatureTimeStamp` (1.2.3.2), nebo může být na požádání vydána využitím mechanismu TSP ([11]). Tím je k okamžiku uvedeném v časové značce zaručena existence podpisu.

1.2.5.4 XAdES-C

Ve formě C (*with Complete validation data references*, s kompletními referencemi validačních dat) podpis obsahuje data potřebná k ověření všech certifikátů uvedených v podpisu. Jedná se především o kompletní certifikační cesty těchto certifikátů (tj. odkazy na všechny certifikáty v certifikační cestě) a odkazy na všechna CRL všech použitých certifikátů (včetně certifikátů v certifikační cestě). Zahrnutí všech validačních dat je základním předpokladem pro dlouhodobé ověřování dokumentu (neboť v budoucnu již validační data nemusí být k dispozici).

1.2.5.5 Rozšířené formy XAdES

Standard dále definuje tři rozšířené podoby XAdES, které lze z nižších forem vytvořit přidáním některých nepodepsaných vlastností.

Forma X (*eXtended signature with time indication*, rozšířený podpis s označením času) je rozšířením formy C. Do podpisu přidává QP obsahující časová razítka vytvořená nad validačními daty.

Forma X-L (*eXtended Long electronic signatures with time*, dlouhý rozšířený podpis s označením času) je rozšířením formy X. Do podpisu přidává QP obsahující další certifikáty a revokační data, obvykle použitá pro dlouhodobé ověřování.

Forma A (*Archival*, archivní) je nejvyšší formou XAdES. Je rozšířením formy X-L a obsahuje archivní časová razítka a další nepodepsané QP, které umožňují bezpečné dlouhodobé ověřování a archivaci elektronicky podepsaných dat.

Standard [5] se popisu jednotlivých forem věnuje podrobněji.

1.3 XAdES baseline profile

Specifikace XAdES definuje mnoho volitelných dodatečných elementů pro vložení informací o podpisu a podepisovaných datech (QP). Mohou tak vznikat podpisy stejné úrovně (1.2.5), které se ale značně liší v obsažených QP nebo jen způsobu jejich vložení do podpisu. V důsledku toho může být pro stranu pracující nějakým způsobem s podpisem (typicky ověřujícího, ale i pro stranu povyšující podpis do vyšší formy) obtížné správně interpretovat různé podoby podpisu, resp. interpretovat je stejným způsobem, jako podepisující.

V zájmu zvýšení interoperability aplikací využívajících XAdES proto vydalo ETSI technickou specifikaci **XAdES Baseline Profile** [2], která značně omezuje volnost ve vytváření podpisu, již zavádí původní specifikace [5]. Podpisy vytvořené podle této specifikace jsou plnohodnotnými XAdES podpisy a navíc mohou být označovány jako *baseline conformant*. Následující část práce shrnuje tyto omezující podmínky.

1.3.1 Úrovně baseline profilu

Zavedením omezujících podmínek pro původní formy XAdES definuje *baseline* profil čtyři úrovně (*conformance levels*) podpisu. Podobně jako u původních forem XAdES i zde platí, že vyšší úrovně musí splňovat požadavky kladené na podpis nižšími úrovněmi.

Úroveň B (*B-Level, Basic*) omezuje původní formy BES a EPES (1.2.5). Hlavním přínosem této úrovně je sjednocení způsobu jednoznačné identifikace podepisujícího certifikátu a způsob jeho podepsání (1.3.3.1).

Úroveň T (*T-Level, Trusted time for signature existence*, podpis s důvěryhodným časem existence podpisu) omezuje původní formu T (1.3.3.2).

Úroveň LT (*LT-Level, Long Term*, dlouhodobý podpis) omezuje původní rozšířenou formu X-L.

Úroveň LTA (*LTA-Level, Long Term with Archive time-stamps)*, dlouhodobý podpis s archivním časovým razítkem) omezuje původní rozšířenou formu A.

Jak je z výčtu úrovní patrné, v *baseline* profilu jsou vynechány formy C a X, a to právě z důvodů snížení různorodosti podpisů a zvýšení interoperability.

1.3.2 Obecné požadavky

Baseline profil klade několik obecných požadavků na libovolný XAdES, a to jak na proces a výstup jeho vytváření, tak na proces jeho ověřování.

1.3.2.1 Požadavky na použité algoritmy

Specifikace zakazuje použití algoritmu MD5 ([13]). Z právního pohledu pak musí použité algoritmy a jejich parametry (např. délka klíče) splňovat právní požadavky na elektronický podpis daného státu, aby bylo možné podpis legislativně považovat za platný.

Dále specifikace vymezuje množinu preferovaných algoritmů kanonikalizací a transformací, ostatní algoritmy ale nezakazuje (jmenované algoritmy jsou doporučeny klíčovým slovem **should** dle [14], jiné hodnoty jsou tedy přípustné); proto tato doporučení nejsou pro ověřování podpisu zajímavá (podepisující stejně může použít libovolný algoritmus a naplnit požadavky specifikace) a práce se jim dále nevěnuje.

1.3.2.2 Způsob vložení QP

Původní specifikace umožňuje QP vkládat přímo, tj. do elementu `QualifyingProperties`, nebo nepřímo, tj. v elementu `QualifyingPropertiesReference` uvést odkaz na skutečné umístění QP (1.2.1). *Baseline* profil druhý ze způsobů zakazuje a nařizuje vkládat všechny QP přímo. Pro aplikace zpracovávající formát XAdES tím odpadá nutnost umět získat podmnožinu QP ze specifikovaného odkazu, ze všech sesbíraných podmnožin QP sestavit kompletní sadu vlastností a zkontrolovat správnost této vzniklé sady.

1.3.2.3 Požadavky na ověření

Ověřující strana musí být schopna přijmout a zpracovat XAdES dle původní specifikace ([5]), i kdyby tento podpis nespňoval požadavky kladené na něj *baseline* profilem. Procesem zpracování se specifikace přímo nezabývá a odkazuje na příslušnou normu (1.4).

1.3.3 Požadavky na jednotlivé formy

1.3.3.1 Úroveň B

Kromě společných požadavků profilu musí úroveň B profilu splňovat následující požadavky kladené na QP.

Podepisující certifikát Původní specifikace požaduje zabezpečení podepisujícího certifikátu a uvádí dva způsoby, jak toho docílit (1.2.5.1). *Baseline* profil vyžaduje, aby byla použita druhá varianta, tedy aby byl do podpisu vložen element `xades:SigningCertificate` obsahující otisk, vydavatele a sériové číslo podepisujícího certifikátu, který se musí nacházet v elementu `ds:X509Certificate` uvnitř `ds:KeyInfo`.

Profil dále zakazuje použití volitelného atributu `URI`, který jinak může specifikovat umístění certifikátu; v tomto profilu je jeho umístění jasně předepsáno.

`SigningCertificate` může obsahovat i další certifikáty, které potom musí být součástí certifikační cesty. Platí pro ně stejná omezení jako pro podepisující certifikát, tedy se musí nacházet v `KeyInfo` a nesmí obsahovat atribut `URI`.

Čas podepsání Podpis musí obsahovat `xades:SigningTime` specifikující okamžik vytvoření podpisu.

Formát podepsaných datových objektů Podepisující strana je povinna pro každý podepsaný datový objekt (s výjimkou podepsaných QP, tedy elementu `xades:SignedProperties`) vložit do podpisu element `xades:DataObjectFormat`, který popisuje formát konkrétního datového objektu. Tento element musí obsahovat alespoň typ MIME v potomkovi `MimeType` a odkaz na podepsanou `ds:Reference` datového objektu v atributu `ObjectReference`.

Specifikace za částí týkající se formátu podepsaných objektů uvádí zajímavou poznámku týkající se umístění `ds:Reference`. Konkrétně zdůrazňuje, že odkazovaná reference nemusí být přímým potomkem elementu `ds:SignedInfo`, ale může být například i potomkem podepsaného `ds:Manifest`, což z ní také dělá podepsanou referenci. Takovým případům a z nich vyplývajícím komplikacím se věnuje část 6.2.

1.3.3.2 Úroveň T

Jediným dalším požadavkem kladeným na úroveň T je vložení alespoň jednoho časového razítka do elementu `xades:SignatureTimeStamp`, nebo poskytnutí okamžiku existence podpisu použitím mechanismu TSP. Nejsou tedy definována omezení původní formy T (1.2.5), pro dosažení úrovně T stačí naplnit obecné požadavky profilu, požadavky úrovně B a požadavky původní formy XAdES-T.

1.4 Signature verification procedures and policies

S narůstající komplexitou vyšších forem pokročilých podpisů narůstá i náročnost jejich korektního ověření tak, aby v něm byla zahrnuta všechna bezpečnostní kritéria a aby toto ověření zohledňavolo všechny aspekty elektronického podpisu, od jeho formální podoby přes použité certifikáty až po podpisovou politiku.

Dokument [1] vydaný ETSI popisuje obecný algoritmus ověření elektronického podpisu, který tyto aspekty důsledně zohledňuje. Jedná se o abstraktní algoritmus aplikovatelný na libovolný pokročilý podpis (nejen XAdES, ale například i PAdES [15] nebo CAdES [11]).

Kromě samotného algoritmu ověření, který bude realizační část této práce implementovat, popisuje specifikace i některé obecné mechanismy aplikované při ověřování; nejdůležitějším z nich – třístavovému mechanismu ověření a podpisové politice – se věnuje i tato část práce.

1.4.1 Indikace výsledku ověření

Standard definuje tři možné výsledky ověření elektronického podpisu:

VALID Tento výsledek ověření značí, že podpis je v pořádku, tedy splňuje formální požadavky kladené na podpis, je kryptograficky platný a jsou splněna veškerá omezení diktovaná podpisovou politikou.

INVALID Tento výsledek značí, že nebyl splněn alespoň jeden požadavek z předchozího bodu a že ověřující strana má jistotu, že je podpis neplatný.

INDETERMINATE Tento výsledek značí, že ověřující strana nemá dostatek informací k tomu, aby mohla rozhodnout, zda je podpis platný či nikoliv.

Tato označení jsou aplikována nejen na celkový výsledek ověření, ale také na výsledek ověření každého dílčího omezení (ať už diktovaného podpisovou politikou nebo vyplývajícího z formátu použitého pokročilého podpisu).

Pro výsledky INVALID a INDETERMINATE standard definuje množinu upřesňujících stavů, které musí být součástí výstupu ověření, pokud je s jedním z těchto výsledků ověření skončeno. Upřesňující stavy mají pomoci straně, která žádá o ověření, identifikovat příčinu neúspěchu ověření. Jedná se o řetězcové konstanty a jsou tak interpretovatelné i strojově; v některých případech musí dle specifikace ověřující strana na výstupu uvést ještě zpřesňující textovou zprávu obsahující člověkem čitelný popis chyby.

1.4.2 Podpisová politika

Použitá podpisová politika je z pohledu specifikace sada omezení, která vstupuje do procesu ověření a má vliv na jeho výsledek. Omezení plynoucí z podpisové

politiky mohou výsledek ověření podpisu změnit v zásadě libovolně, rozdělit je lze do dvou základních skupin:

- Omezení zpřísňující; tato omezení mohou způsobit, že podpis, který by validační autorita bez jeho přítomnosti ověřila s výsledkem VALID (resp. INDETERMINATE), může být ověřen s výsledkem INDETERMINATE nebo INVALID (resp. INVALID). Typickými omezeními z této skupiny jsou omezení identity podepisujícího, omezení podepsaných datových objektů atp.

- Omezení relaxující; tato omezení mohou způsobit, že podpis, který by validační autorita bez jeho přítomnosti ověřila s výsledkem INVALID (resp. INDETERMINATE), může být ověřen s výsledkem INDETERMINATE nebo VALID (resp. VALID); označení omezení zde tedy není zcela přesné. Typickým omezením z této skupiny může být ignorování skutečnosti, že je některý z použitých certifikátů odvolán.

Ověření mnohých omezení (teoreticky libovolného z nich, viz [1]) plynoucích z podpisové politiky může být na požádání z procesu ověření vynecháno, ověřující aplikace pak ale musí jako součást výstupu ověření uvést kontroly, které záměrně nebyly provedeny. Žádoucí takové chování může být v případě, kdy strana žádající o ověření ví o nějakém nedostatku podpisu, ale přeje si tento nedostatek ignorovat a zjistit výsledek některé z dalších fází ověření, která by jinak neproběhla.

1.4.3 Algoritmus ověření

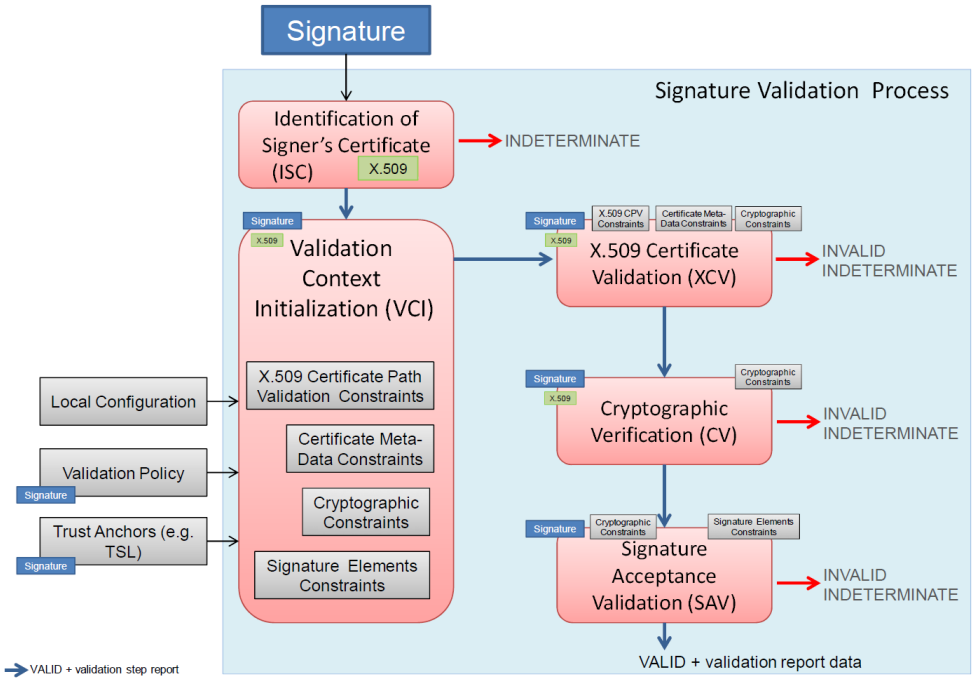
Abstraktní algoritmus popisovaný specifikací je rozdělen na několik vzájemně nezávislých bloků. Standard nepředepisuje, že se tyto bloky musí provádět v pořadí, ve kterém jsou uvedeny (viz 1.1), ale výstup implementace algoritmu musí být vždy shodný s výsledkem, se kterým by skončilo ověření dodržující toto pořadí.

1.4.3.1 Ověření struktury podpisu

V této fázi (kterou algoritmus implicitně předpokládá) musí být nejprve ověřeno, že podpis splňuje požadavky kladené na něj základní specifikací, aby bylo možné bezpečně provést fáze následující. Pro XAdES to znamená především ověření validity vstupního XML dle XML schématu, ověření správné struktury základního XML-DSig a ověření korektního způsobu vložení QP (přímo nebo nepřímo). Množina vložených QP musí také odpovídat některé ze základních forem podpisu (1.2.5).

V případě, že jakákoliv z provedených kontrol skončí s chybným výsledkem, ověřující aplikace musí ukončit ověření s výsledkem INVALID a upřesňujícím stavem FORMAT_FAILURE.

Obrázek 1.1: Zjednodušený proces ověření podpisu dle [1]



1.4.3.2 Identifikace podepisujícího certifikátu

V této fázi je třeba identifikovat a získat podepisující certifikát. V případě XAdES je buď vložen v `ds:KeyInfo` a přímo podepsán (1.2.5.1), nebo podpis obsahuje QP `SigningCertificate`, který pomocí otisku a identifikace vystavitele a sériového čísla certifikátu jednoznačně podepisující certifikát identifikuje. Druhý způsob je předepsán *baseline* profilem, který navíc požaduje, aby byl (nepodepsaný) certifikát přiložen v `ds:KeyInfo` (1.3.3.1).

V případě, že se nepodaří certifikát identifikovat, nebo se jej podaří identifikovat, ale ne získat, musí ověření skončit s výsledkem INDETERMINATE. Jinak tato fáze končí s výsledkem VALID.

1.4.3.3 Inicializace validačního kontextu

V této fázi musí ověřující aplikace zpracovat všechny vstupy tak, aby disponovala všemi daty potřebnými pro ověření elektronického podpisu. Těmito vstupy jsou implicitní konfigurace ověřující aplikace, informace obsažené uvnitř podpisu (např. podpisová politika nebo specifikace závazku (1.2.2.2) a případně i dodatečné parametry, které mohou být součástí požadavku na ověření podpisu.

Na konci této fáze by měla ověřující aplikace mít k dispozici následující množinu dat:

- Sadu omezení a požadavků kladených na certifikační cestu (např. důvěryhodné kotvy).
- Sadu metadat certifikátů, například zda je konkrétní certifikát kvalifikovaným certifikátem ([7]).
- Sadu kryptografických omezení (např. požadavky na délku klíče nebo použité algoritmy).
- Sadu požadavků kladených na přítomnost elementů v podpisu. Takový požadavek může vynucovat přítomnost (nebo naopak absenci) elementu, který je pro ověřovanou formu XAdES volitelný.
- Podepisující certifikát identifikovaný a získaný ve fázi 1.4.3.2.
- Samotný podpis, tedy XAdES XML dokument.

Pokud se aplikaci podařilo zpracovat všechny vstupy, ověření pokračuje další fází; v opačném případě musí aplikace v závislosti na chybě ukončit ověření s výsledkem INVALID (např. při chybě zpracování podpisové politiky nebo neznámém typu závazku) nebo INDETERMINATE (pokud se podpisovou politiku nepodaří identifikovat).

1.4.3.4 Ověřování certifikátů

V této fázi musí ověřující aplikace ověřit platnost podepisujícího certifikátu a sestavit certifikační cestu, která splňuje následující požadavky:

- Na jejím konci je některá z důvěryhodných kotev uvedených v omezeních certifikační cesty.
- Ke všem certifikátům, které jsou součástí certifikační cesty, jsou k dispozici čerstvá revokační data (CRL nebo OCSP).
- Žádný z certifikátů, který je součástí certifikační cesty, nebyl revokován.
- Jsou splněna všechna další omezení a požadavky kladené na certifikační cestu (1.4.3.3).
- Jsou splněna všechna kryptografická omezení týkající se algoritmů specifikovaných v certifikátu (způsob jeho podpisu apod.).

V případě, že je se nepodaří nalézt žádnou certifikační cestu končící u některé důvěryhodné kotvy, končí ověření s výsledkem INDETERMINATE; pokud je certifikační cesta nalezena, ale nesplňuje všechny požadavky na ní kladené, končí ověření dle závažnosti nedostatku s výsledkem INVALID (nejsou splněna doplňující omezení pro certifikační cestu) nebo INDETERMINATE (nejsou k dispozici čerstvá revokační data, certifikát CA v cestě byl revokován apod.).

1.4.3.5 Kryptografické ověření

V této části je ověřován samotný podpis dle standardu XML-DSig, tedy otisky všech referencovaných datových objektů a hodnota podpisu (1.1.2).

V případě, že se nepodaří některý z referencovaných datových objektů získat pro výpočet otisku, končí ověření s výsledkem INDETERMINATE. Pokud vypočtený otisk datového objektu neodpovídá hodnotě uvedené v referenci nebo se nepodaří ověřit hodnotu podpisu, končí ověření s výsledkem INVALID.

Specifikace neuvádí, jak má ověřující strana přistupovat k ověřování případných vnořených referencí (1.1.1.2), záleží tedy na konkrétní implementaci.

1.4.3.6 Rozhodnutí o přijetí podpisu

V poslední fázi přichází na řadu veškeré další kontroly, které nebyly provedeny v předchozích fázích. Spadají sem například všechny požadavky na podobu podpisu popsané *baseline* profilem, omezení vynucující výskyt QP, ostatní omezení plynoucí z podpisové politiky apod. Také jsou zde ověřena všechna zbylá kryptografická omezení.

Výsledkem poslední fáze ověření je INDETERMINATE, pokud některý z kryptografických algoritmů použitých v podpisu již není považován za bezpečný a není tak možné bezpečně určit, zda je podpis platný; INVALID, pokud není splněno některé z ostatních omezení kontrolovaných v této fázi; VALID, pokud všechny kontroly skončily úspěchem.

Kapitola 2

Specifikace požadavků

Kapitola shrnuje požadavky na funkcionalitu jádra nástroje a rozhraní pro práci s ním a nefunkční požadavky pro oba tyto výstupy. Zdrojem pro vytvoření seznamu bylo v první řadě zadání práce, v druhé řadě požadavky plynoucí ze specifikací popsaných v kapitole 1. Jedná se spíše o rámcový seznam, který ale v souhrnu postihuje všechny důležité součásti nástroje.

Požadavky jsou kromě rozdělení na funkční a nefunkční sdruženy dle aspektů nástroje, kterých se týkají.

2.1 Funkční požadavky na jádro nástroje

Tato sekce obsahuje výčet funkčních požadavků, které by mělo jádro splňovat; jedná se o požadavky na formát vstupů a výstupů, ověřovací proces a podporované formy XAdES.

2.1.1 Požadavky na vstup a výstup nástroje

Jádro nástroje musí:

- být schopno přijmout jako vstup libovolný podpis ve formátu XAdES; pokud podpis nesplňuje některý z elementárních požadavků formátu, musí podpis odmítnout ještě před započetím ověřovacího procesu;
- i v případě libovolné chyby nebo neočekávaného stavu dokončit ověřovací proces a žadatele o ověření informovat o stavu podpisu; v případě chyby nástroje musí tento stav být INDETERMINATE;
- být schopno získat další vstupy, které mohou být potřeba pro ověření podpisu, zejména odkazované podepsané datové objekty a další data referencovaná pomocí URI;

- výstup uzpůsobit tak, aby byl strukturovaný a interpretovatelný dle specifikace TS 102 853 (1.4); konkrétní interpretace výstupu bude na prezentační vrstvě a nástroj ji neřeší.

2.1.2 Požadavky na proces ověření podpisu

Jádro nástroje musí:

- implementovat proces ověření dle TS 102 853, tedy provést všechny fáze ověření specifikované tímto dokumentem (1.4.3), pokud to stav podpisu umožní (nemá například smysl provádět kryptografické ověření, pokud se nepodařilo identifikovat certifikát pro ověření);

- pokračovat v provádění kontrol, které mu stav podpisu umožní, i pokud by měl dle algoritmu v některé fázi z důvodu chyby podpisu skončit – výsledek ověření podpisu se v takovém případě nesmí odchylovat od specifikace; přínosem je přítomnost informací ve výstupu, které by algoritmus dle původní specifikace neposkytl;

- rozpoznat formu XAdES a přizpůsobit jí proces ověření – například pro podpisy ve formě T správně zohlednit časovou značku;

- umožnit přeskočit určitý element (konkrétní kontrolu) v procesu ověření na vyžádání žadatele o ověření nebo v důsledku použité podpisové politiky a na tuto skutečnost upozornit ve svém výstupu;

- podporovat zpracování `Manifest`ů (1.1.1.2) a případně umožnit konfiguraci chování v případě jejich přítomnosti, tedy podle kontextu požadovat nebo nepožadovat ověření otisků datových objektů, přítomnost QP `DataObjectFormat` (1.2.2.2) apod.

2.1.3 Požadavky na podporované formy XAdES

Jádro nástroje musí:

- přijmout XAdES v libovolné formě (1.2.5) i pokud tuto formu neumí správně zpracovat (tedy libovolnou formu vyšší než T); v takovém případě musí žadatele o ověření informovat, že podpis neumí ověřit;

- správně ověřovat podpisy ve formě XAdES-BES a XAdES-EPES vytvořené podle podle *baseline* profilu, tedy *baseline* úrovně XAdES-B (1.3.1). Pokud ověřovaný BES nebo EPES podpis nebude úrovně *baseline* XAdES-B a nástroj detekuje, že nedokáže podpis správně ověřit, musí proces ověření skončit s výsledkem INDETERMINATE;

- správně ověřovat podpisy ve formě XAdES-T podle *baseline* profilu, tedy *baseline* úrovně XAdES-T. Pokud ověřovaný T podpis nebude *baseline* úrovně XAdES-T a nástroj detekuje, že nedokáže podpis správně ověřit, musí proces ověření skončit s výsledkem INDETERMINATE.

2.2 Nefunkční požadavky na jádro nástroje

Protože je jádro nástroje koncipováno jako knihovna, je na něj kladeno i několik požadavků spíše nefunkčního charakteru, které se týkají především jeho rozšiřitelnosti a modularity.

2.2.1 Požadavky na rozšiřitelnost nástroje

Jádro nástroje musí:

- umožnit definovat ověřovací proces (*profil*), podle kterého se má ověření určitého podpisu provést;
- vystavovat rozhraní pro implementaci dílčích algoritmů používaných v procesu ověření, jako jsou algoritmy pro sestavení a ověření certifikační cesty, získání odkazovaných datových objektů apod.;
- umožnit konfiguraci použití těchto rozhraní v rámci ověřovacího procesu;
- poskytovat minimálně základní funkční implementaci těchto rozhraní, aby vždy byla zaručena funkčnost nástroje;
- umožňovat implementaci dalších kontrol, které mohou být prováděny v rámci ověřovacího procesu a umožnit zavedení těchto kontrol do ověřovacího procesu.

2.2.2 Požadavky na implementační platformu

Jádro nástroje musí:

- být integrovatelné v prostředí JVM – tento požadavek vzešel dodatečně od vedoucího práce, protože hotový nástroj by měl být po svém dokončení součástí většího celku, který je již v tomto prostředí nasazen;
- proto být implementováno v některém z programovacích jazyků, který umožní nasazení do tohoto prostředí, jako je Java, Python nebo Scala;
- formou knihoven využívat existující a ověřená řešení pro zpracování komplexních úloh nesouvisejících přímo s cílem práce, zejména pro kryptografické úlohy (algoritmy pro výpočet podpisu, otisku apod.) a práci s digitálními certifikáty, pokud taková řešení existují a jsou integrovatelná do prostředí JVM.

2.2.3 Další požadavky

Jádro nástroje musí být:

- funkční bez internetového připojení; pokud podpis ke svému ověření připojení nevyžaduje, musí toto ověření bez chyb proběhnout a skončit správným výsledkem; pokud je připojení z libovolného důvodu potřeba (např. podepsané datové objekty jsou online) a připojení není k dispozici, musí ověření skončit s výsledkem INDETERMINATE.
- postaveno tak, aby do něj v budoucnu mohla být zavedena podpora pro cachování obsahu; ta nebude součástí vyvíjeného prototypu nástroje.

2.3 Funkční požadavky na rozhraní

Rozhraní bude sloužit jako tenká obálka nad samotným jádrem nástroje. Samo o sobě tak nebude mít žádnou business funcionalitu, pouze jádru zprostředkuje vstupy a jeho výstupy transformuje do vhodné odpovědi.

Jako technologie pro implementaci rozhraní byl z následujících důvodů zvolen protokol SOAP:

- jedná se o zavedený, ověřený protokol s širokou podporou pro implementaci napříč programovacími jazyky a frameworky;
- jeho použití umožňuje realizaci všech požadavků uvedených v této části práce;
- ověřovaný podpis nebo podpisy je možné společně s požadavkem odeslat jako přílohu;
- v budoucnu mohou být použity standardizované mechanismy, jako jsou zabezpečení webových služeb ([16]) nebo asynchronní odpovědi na požadavky ([17]).

2.3.1 Požadavky na vstup a výstup rozhraní

Na vstupu nástroje bude:

- XML dokument obsahující jeden či více XAdES vložený jako příloha;
- volitelně sada parametrů pro pozměnění chování jádra nástroje (2.3.2).

Na výstupu nástroje bude XML struktura odpovídající specifikaci TS 102 853 (1.4), která bude obsahovat:

- celkový výsledek ověření podpisu,
- výsledek všech provedených fází ověření,
- výsledek všech provedených kontrol napříč všemi fázemi,

Dále bude výstup obsahovat následující informace:

- Formu XAdES (1.2.5), pokud byla rozpoznána;
- Informace o certifikátu použitém k ověření podpisu – základní informace o období jeho platnosti, předmětu, vydaveli, sériovém číslu apod.

2.3.2 Požadavky na parametrizaci vstupu

Protože rozhraní bude sloužit především k parametrizování a nastavení jádra validátoru, musí být možné na vstupu předat následující informace:

- důvěryhodné kotvy pro ověření certifikační cesty podepisujícího certifikátu, kterým důvěřuje žadatel o ověření; zároveň bude možné zakázat použití implicitních důvěryhodných kotev (tedy definovaných přímo v jádru nástroje) v ověřovacím procesu;
- další certifikáty, které se jádro nástroje může pokusit použít k sestavení certifikační cesty; takové certifikáty sice mohou být součástí přímo `ds:X509Data` (1.1.1), jejich absence ale může zapříčinit, že ověření skončí s výsledkem INDETERMINATE/NO_CERTIFICATE_CHAIN_FOUND – naopak jejich dodatečné vložení do `ds:X509Data` by mohlo způsobit porušení integrity podpisu, pokud by `ds:KeyInfo` bylo podepsaným elementem;
- podepsané datové objekty společně s URI – může nastat situace, že nástroj nebude schopen získat podepsané datové objekty dereferencováním URI uvedené v podpisu (data nejsou již déle dostupná, jedná se o relativní URI, která v prostředí ověřujícího nástroje identifikuje jiný zdroj, než v prostředí žadatele, URI identifikuje zdroj na lokálním souborovém systému žadatele apod.). V takovém případě je žádoucí, aby byl nástroj schopen získat datové objekty jinou cestou – nástroj proto jednak umožní specifikovat přemapování z původní URI na novou, ze které bude schopen datový objekt získat, jednak umožní s původní URI předat přímo podepsaný datový objekt;

Kapitola 3

Přehled existujících řešení

Tato kapitola obsahuje výsledku průzkumu, který byl proveden v oblasti existujících řešení pro ověřování elektronických podpisů ve formátu XAdES. Důraz je kladen především na jejich integrovatelnost v prostředí JVM a na to, jak tato řešení naplňují požadavky uvedené v 2.

Srovnávána jsou tři existující řešení, která by dle svých dokumentací měla přistupovat k ověřování podpisu jako k netriviálnímu, konfigurovatelnému procesu. Proto srovnání neobsahuje některá jednoduchá řešení, jako [18].

3.1 Digidoc4j

Digidoc4j je knihovnou implementující digitální podpisy ve formátu XAdES a ASiC. Je spravována estonskou vládní autoritou pro informační systémy ([19]). Aktuálně je ve stavu betaverze. Jedná se o pokračovatele staršího projektu *digidoc*, na kterém je postaveno její jádro.

3.1.1 Popis řešení

Knihovna dle dokumentace umožňuje ověřit XAdES podle standardu TS 102 853 a podporuje specifikaci podpisových politik dle doporučení TR 102 038 ([20]).

Implementace je úzce navázána na vlastní formát, který *digidoc4j* používá, *DigiDocXML* ([21]). Jedná se o tenkou obálku vytvořenou nad XAdES, jak je patrné z ukázky 3.1-1. Tento formát navíc nepodporuje všechny QP definované standardem XAdES, a tak ani *digidoc4j* nedokáže pracovat se všemi QP.

V jistém smyslu tak knihovna pracuje s (implicitně určeným) omezeným profilem podpisu, což neporušuje požadavky původní specifikace, plyne z něj ale neuniverzálnost knihovny a nemožnost jejího použití pro zcela libovolný podpis ve formátu XAdES.

```
<SignedDoc>
  <!-- originální datové objekty nebo odkazy na ně -->
  <DataFile/>*
  <!-- jeden nebo více XAdES podpisů -->
  <Signature/>*
</SignedDoc>
```

Ukázka 3.1-1: Schéma formátu DigiDoc. Symbol * za elementem označuje libovolnou násobnost, elementy bez označení se vyskytují právě jednou.

```
public ArrayList validate ( boolean bStrong ) { ... }
```

Ukázka 3.1-2: Hlavička metody `validate` třídy `SignedDoc`

```
public List<DigiDoc4JException> validate (Validate validationType) {
      logger.debug("");
      return this.validationErrors;
}
```

Ukázka 3.1-3: Implementace metody `validate` třídy `Signature`

Proces ověření může být spuštěn ve třech režimech – ověření celého podpisu, ověření časových značek podpisu a ověření podpisové politiky. Pouze prvně zmíněný režim je ovšem zatím implementován, jak je ilustrováno v 3.1.2.

3.1.2 Nedostatky

Samotné API pro ověření podpisu je neintuitivní a působí zastarale. Knihovna pro klíčové metody používá konkrétní implementace standardních java rozhraní, navíc je jádro knihovny stále distribuováno ve verzi bytecodu 49 (Java SE 5), tj. mimo jiné bez generických typů. To je sice praktické pro kompatibilitu se staršími verzemi Javy, nicméně absence dokumentace metod nutí programátora prozkoumávat zdrojový kód a dohledávat, jaké objekty metody skutečně vrací. Kupříkladu metoda třídy `SignedDoc` původního *digidoc* pro ověření podpisu (ve skutečnosti vracející množinu výjimek vyhozených v průběhu ověření) má signaturu 3.1-2.

Setkat se se zjevnou nedotažeností projektu lze při pohledu na kód prakticky kdekoliv; ukázka 3.1-3 je implementací validace podpisu třídou `Signature`. V kombinaci s chabou dokumentací činí podobné nedostatky a nejasnosti integraci knihovny do existujícího kódu velmi náročnou.

Samotný proces ověření je netransparentní a prakticky do něj nelze zasáhnout, neboť knihovna neumožňuje proces konfigurovat, či případně zaintegrovat do něj vlastní kontrolní mechanismy.

3.1.3 Zhodnocení

Nedostatky způsobené omezením se na formát *DigiDocXML* zamezují nasazení knihovny do prostředí pracujícího s obecnými podpisy ve formátu XAdES. Přes očividnou nedokončenost ale knihovna podporuje (nebo do budoucna má podporovat) některé postupy (časové značky s využitím mechanismu TSP, ověření dle TS 102 853), které ostatní řešení postrádají.

3.2 XAdES4j

Nejrozsáhlejším srovnávaným otevřeným projektem zaměřeným pouze na formát XAdES je knihovna XAdES4j. Jedná se o akademický projekt a jeho zdrojové kódy a dokumentace jsou volně dostupné, a to pod licencí GNU GPL ([22]).

3.2.1 Popis řešení

Knihovna nabízí komplexní řešení vytváření a ověřování XAdES až do úrovně C (1.2.5.4). Je architekturně velmi dobře navržena; staví na principu rozšiřitelnosti skrz nejrůznější rozhraní, označovaná jako *service providers*, kterými může být změněno nebo rozšířeno její chování. Pro velkou část těchto rozhraní na druhou stranu knihovna neobsahuje žádnou (nebo v případě nutnosti pouze jednu základní) implementaci.

Druhým hlavním rysem je implementace vytváření XAdES pomocí *profilů*, tedy sad společných nastavení souhrnně definujících celkovou podobu výsledného podpisu, s nimiž lze efektivně opakovaně vytvářet podpisy splňující požadovaná kritéria, například vytvářet podpisy dle *baseline* profilu (1.3).

Za zmínku také stojí implementovaný vysokoúrovňový datový model QP, který poskytuje přístup k informacím kompletně izolovaný od XML reprezentace. Tento přístup je sám o sobě vhodný zejména pro vytváření podpisu, pro který je XML až výstupem; pro ověření je naopak výhodné mít přístup k samotnému dokumentu, aby bylo možné využívat jej jako vstup pro kryptografické operace.

3.2.2 Nedostatky

Knihovna nepodporuje ověření podpisu dle specifikace TS 102 853; i když všechny kontroly popsané specifikací knihovna implementuje (ač v jiném pořadí – takové zpracování ale specifikace dovoluje, pokud je dosaženo shodných výsledků, viz 1.4.3), neumožňuje interpretovat výsledky ověření podle jí definovaného třístavového paradigmatu. Libovolný problém při ověřování má za následek vyhození výjimky, která není v průběhu procesu ověření knihovnou

nikde zachycena, a tak je na klientské aplikaci, aby podle konkrétní instance výjimky rozhodla o důvodu odmítnutí podpisu.

Klientská aplikace, která má rozhodnout o výsledku dle TS 102 853, proto musí být schopná tyto výjimky správně interpretovat (rozhodnout mezi stavy INVALID a INDETERMINATE), což značně zvyšuje komplexitu použití XAdES4j pro ověřování podpisů a zároveň přesouvá zodpovědnost za rozhodnutí o výsledku ověření z knihovny na aplikaci. Z tohoto přístupu plyne také nemožnost pokračovat po vyhození výjimky v provádění dalších kontrol, což značně snižuje informační hodnotu výsledku ověření. Také není možné kontroly žádným způsobem shlukovat nebo kategorizovat, případně záměrně některou z nich vynechat.

Při testování knihovny bylo také nalezeno několik chyb. Knihovna například nesprávně pracuje s kanonikalizací XML c14n11 ([23]) a při pokusu o ověření podpisu, který tuto kanonikalizaci používá, vrací chybu. Chování v takovém případě bohužel není možné ovlivnit – technicky se jedná o problém v knihovně *Apache Santuario* (4.2.2), která je zde použita pro práci s XML-DSig, ale který lze ale při přímém využití této knihovny snadno obejít; XAdES4j nicméně přístup k nastavení *Santuario* nezprostředkovává.

Také kontrola pokrytí podepsaných QP je implementována nedůsledně; při použití XPath transformace, která odstraňuje některou z podepsaných QP, považuje XAdES4j podpis stále za platný.

3.2.3 Zhodnocení

XAdES4j je k realizaci zadání nejblíže ze všech srovnávaných existujících řešení. Některé chyby (které ale mohou být v pozdějších verzích opraveny) znemožňují jeho použití, je to ale především způsob, jakým knihovna přistupuje k odmítnutí podpisu, kvůli kterému ji není možné použít ani jako prostředek v ověřovacím procesu, což je kvůli jinak velmi dobře postavenému řešení škoda.

3.3 Digital Signature Service

Digital Signature Service je softwarový nástroj pro vytváření a ověřování elektronických podpisů vyvinutý oficiálně pod hlavičkou Evropské komise ([24]) a dostupný pod licencí LGPL. Hlavním cílem projektu je poskytnout členským státům EU jednotné rozhraní pro vytváření a ověřování podpisů podle příslušných právních nařízení EU.

3.3.1 Popis řešení

Nástroj je komplexním řešením pro práci s pokročilými elektronickými podpisy a podporuje formáty XAdES a CAdES až do archivní formy A (1.2.5) a PAdES až do formy LTV. Jeho cílem není obecná podpora AdES formátů, ale

podpora v rozsahu definovaném příslušnými právními nařízeními – jedná se zejména o rozhodnutí [25] o minimálních požadavcích na formát podpisu a rozhodnutí [26]. To se ve zkratce zabývá povinností členských států publikovat TSL (*Trusted services list*, seznam důvěryhodných certifikačních služeb) a akceptovat elektronické podpisy na základě informací z TSL ostatních členských států.

Nástroj je distribuován jak ve formě samostatné webové aplikace, která umožňuje podepisovat a ověřovat podepsané elektronické dokumenty, tak ve formě Java SDK, které lze integrovat do vlastního kódu. Součástí distribuce je i „*cook-book*“, tedy dokumentační příručka popisující použití aplikace a především dokumentující samotné SDK.

3.3.2 Nedostatky

Jak již bylo řečeno, validační data pro ověření podpisu jsou získávána na základě TSL. Tím jsou naplněny právní požadavky rozhodnutí Evropské komise [26], ale nelze tím pádem s úspěchem ověřit podpis vytvořený nekvalifikovaným (komerčním) certifikátem – v takovém případě skončí nástroj ihned po nenalezení certifikační cesty ukončí ověření s výsledkem INDETERMINATE (1.4.3.4). V důsledku toho není možné o podpisu zjistit ani žádné další informace.

Nástroj také ignoruje použití `Manifest`ů (1.1.1.2) a nerozpoznává jimi odkazované datové objekty.

Zaměření nástroje na užití v prostředí EU se silně promítá i do samotného SDK, které by mělo podle dokumentace být koncipováno tak, aby bylo použitelné i bez závislosti na mechanismech TSL. To je sice teoretický možné, ale většina tříd implementujících logiku ověření (včetně obecných kroků) je stejně nějakým způsobem na tyto mechanismy navázána. Implementovat ověření zcela bez závislosti na těchto mechanismech by vyžadovalo od začátku implementovat příslušná rozhraní.

3.3.3 Zhodnocení

Jedná se o velmi komplexní řešení celé problematiky AdES, které je v současnosti ale silně závislé na mechanismech TSL, což je na druhou stranu vzhledem k důvodům vzniku projektu a jeho cílům logické.

3.4 Závěr srovnání

Provedený průzkum ukazuje, že existující řešení mají vždy alespoň jeden z následujících nedostatků:

- neimplementují proces ověření dle [1];
- neobsahují podporu pro *baseline* profil;
- zaměřují se pouze na určitou podmnožinu možných podob XAdES (implicitně předpokládají specifickou podpisovou politiku);
- do procesu ověření nelze zasáhnout nebo je obtížně konfigurovatelný.

Implementovaný nástroj by neměl obsahovat žádný z výše uvedených nedostatků.

Kapitola 4

Použité technologie

Následující kapitola stručně popisuje volbu programovacího jazyka a také technologie, které byly zvoleny pro řešení dílčích problémů, které nástroj bude řešit, jak bylo stanoveno v rámci analýzy požadavků (2.2.2). Pokud pro některou úlohu existuje více vhodných řešení, jsou zde tato řešení srovnána a následně je z nich vybráno to, které se pro použití nástrojem jeví nejvhodnější.

Jedná se zejména o technologie pro řešení úloh z oblasti bezpečnosti – práci se základním elektronickým podpisem (4.2), časovými razítky (4.3) a pro budování a ověřování certifikačních cest(4.4). Dále jsou to technologie pro konfiguraci prostředí nástroje (4.5) a implementaci rozhraní webových služeb (4.6).

Některé pasáže popisují způsob práce s knihovnami nebo přímo uvádějí ukázky kódu, který bude k řešení podproblémů použit, čímž se částečně prolínají s kapitolou 6; ta je ale zaměřená především na řešení problémů, které vyvstaly v průběhu implementace nástroje a na konkrétní aplikaci technologií, nikoliv na použité technologie obecně.

4.1 Volba programovacího jazyka

Dle požadavků (2.2.2) musí být nástroj integrovatelný do prostředí JVM a proto i jazyk použitý k jeho realizaci musí být kompatibilní s touto platformou. V zájmu urychlení vývoje připadají v úvahu pouze zavedené jazyky s aktivní uživatelskou základnou a kvalitní dokumentací, jakými jsou Java, Scala, Groovy či Python, naopak nebudou uvažovány jazyky experimentální povahy.

Klíčovými kritérii pro volbu programovacího jazyka jsou (od nejdůležitějšího po nejméně podstatné):

Existence potřebných technologií Zvolený jazyk musí být integrovatelný s knihovnami a technologiemi, které budou v řešení použity. U knihoven se

očekává, že většina z nich bude napsaná v jazyce Java, neboť se stále jedná o nejrozšířenější jazyk pro JVM. To pro většinu jazyků nepředstavuje problém – např. Groovy, Scala nebo Pythonová implementace pro JVM Jython dokáží importovat Java třídy a volat jejich metody.

Pro každý z těchto jazyků stejně jako pro samotnou Javu také existuje řada specializovaných, moderních webových frameworků. Ač se tedy jedná o nejdůležitější kritérium výběru, nevyřazuje žádný z uvažovaných jazyků.

Autorova znalost jazyka Pro pohodlný a efektivní vývoj je důležité znát výhody i nevýhody jazyka a ovládat způsob práce s ním. Nejedná se přitom pouze o syntaxi a idiomy daného jazyka, ale zejména o znalost implementačních detailů, možností paralelizace apod. Z tohoto pohledu vychází jako nejvhodnější volba samotná Java, naopak jako nejméně vhodný potom jazyk Python.

Uživatelská přívětivost, rychlost vývoje Moderní programovací jazyky jsou obvykle objektově orientované, často ale přináší mnohé funkcionální konstrukty a další syntaktické vlastnosti, které mohou programátorovi usnadnit vývoj. Ty jsou pak využívány i konkrétními frameworky, které pro tyto jazyky existují, což dále zesnadňuje práci s nimi.

Pro vývoj nástroje není výslovná potřeba těchto konstruktů očekávána a proto je preferována „konzervativnější“ Java; i ta přitom ve verzi 8 přináší lambda výrazy a další funkcionální prvky, které mohou být v případě potřeby využity ([27]).

Na základě těchto požadavků byla jako implementační jazyk zvolena Java 8.

4.2 Zpracování XML-DSig

Protože zpracování základního XML podpisu je mimo rozsah této práce a pro platformu Java již existují řešení, která implementují standard XML-DSig včetně jeho ověřovacích procedur (1.1.2), bude zvoleno jedno z nich – v úvahu přicházejí *Java XML Digital Signature API* (4.2.1), které je dodáváno jako součást JDK, a *Apache Santuario* (4.2.2), otevřená implementace standardu.

4.2.1 Java XML Digital Signature API

Od verze SE 6 obsahuje Java popis a implementaci standardu JSR 105, **Java XML Digital Signature API** ([28]), tedy API pro práci se základním elektronickým podpisem. Jeho cílem je sjednotit postupy při vytváření a ověřování XML podpisů a poskytnout podklady pro další implementace tohoto standardu a registraci jejich kryptografických služeb v rámci JCA.

```
1  // použití defaultního DOM mechanismu pro zpracování podpisu
2  XMLSignatureFactory factory = XMLSignatureFactory.getInstance("DOM");
3  // získání elementu podpisu (ds:Signature)
4  Node signatureNode = ...
5  // inicializace validačního kontextu klíčem pro ověření a elementem
       podpisu
6  DOMValidateContext validateContext =
7          new DOMValidateContext(new KeySelector() {...}, signatureNode);
8  // získání objektu podpisu z~XML struktury
9  XMLSignature signature =
       factory.unmarshalXMLSignature(validateContext);
10 // informace o~výsledku ověření
11 boolean coreValidity = signature.validate(validateContext);
```

Ukázka 4.2-1: Způsob vyhodnocení platnosti XML-DSig při použití *Java XML Signature API*

Funkcionalita API pro ověření podpisu dle 1.1 je úplná, zároveň ale také velmi omezená, jak ilustruje ukázka 4.2-1 – metodou `validate` je možné pouze zjistit, zda je podpis platný, či nikoliv. Pro zjištění informace o tom, která část *core validation* (1.1.2) selhala, je potřeba projít všechny reference podpisu a pokusit se najít referenci s chybným otiskem. Pokud taková není nalezena, ověření muselo skončit neúspěchem kvůli nesprávné hodnotě podpisu – tuto informaci je možné odvodit pouze tímto implicitním způsobem.

Pouze ve zvláštních případech, kdy se nepodaří získat referencovaná data nebo identifikovat některý z použitých algoritmů, vyhazuje metoda `validate` výjimku.

API je vystavěno tak, aby byl programátor zcela odstíněn od konkrétní implementace; není pro něj proto možné získat přímý přístup k funkcionalitě implementujících tříd, jako jsou právě výpočty otisků nebo podpisu. To neplatí za předpokladu, že programátor ví, které třídy budou v dané situaci využity – takové předpoklady jsou ale z programátorského hlediska nesprávné a i potenciálně nebezpečné, protože:

1. interní třídy se obecně v různých verzích knihovny mohou měnit a do budoucna není zaručena jejich kontinuita ani zachování kontraktů jejich metod;

2. se může změnit konkrétní implementace dodávaná tovární třídou (4.2-1).

API také neumožňuje přímý přístup k podepisovaným datovým objektům v různých fázích jejich transformace, přístup k manifestům (1.1.1.2) nebo k elementům `ds:Object` (1.1.1). Protože XAdES využívá tyto a další mechanismy XML-DSig i na dalších místech, jako jsou kanonikalizace či transformace pro

časová razítka (1.2.3.2) nebo identifikace certifikátu, který má být použit pro ověření podpisu (1.2.2.1), bylo by vhodné pro zpracování těchto informací použít tutéž implementaci, která se bude starat o základní XML podpis, a to z uvedených důvodů není v případě využití *Java XML Digital Signature API* možné.

Dalším důvodem k použití jiné implementace je nemožnost získat původní XML dokument, ať už ve formě DOM nebo surového XML, což je způsobeno vysokoúrovňovým přístupem API, které má být nezávislé na použitém mechanismu (i když výchozí implementace využívá právě DOM).

Při pohledu do zdrojových kódů výchozí implementace API je patrné, že k realizici veškeré logiky a mechanismů, které jsou uvedeny v předchozím odstavci a od kterých je kvůli API programátor odstíněn, je využita knihovna *Apache XML Security*, které se věnuje další část práce.

4.2.2 Apache Santuario

Projekt **Apache Santuario** ([29]) je otevřený projekt realizovaný pod hlavičkou Apache Software Foundation, který poskytuje implementaci dvou klíčových standardů souvisejících s bezpečností XML, a to šifrování [30] a právě podepisování [3]. Implementace je poskytována nezávisle pro jazyky C++ (*Apache XML Security for C++*) a Java (*Apache XML Security for Java*, původně samostatný projekt *Apache XML Security*).

Knihovna pro jazyk Java obsahuje následující komponenty týkající se XML-DSig:

- implementaci JSR 105, která je základem pro výchozí implementaci tohoto API dodávanou jako součást JRE (4.2.1);
- nestandardizované API pro práci s elektronickým podpisem XML založené na DOM, které implementace JSR 105 obaluje a interně používá;
- nestandardizované API pro práci s elektronickým podpisem XML založené na StAX.

Posledně jmenovaná implementace není pro ověřování XAdES příliš zajímavá, protože v rámci provádění kontrol může vyvstat potřeba opakovaně pracovat s jedním datovým objektem nebo jinou součástí podpisu, pro což je proudové zpracování podpisu nevhodné.

Zajímavější je API používané i standardní implementací JSR 105 (4.2.1), založené na DOM reprezentaci. Za cenu toho, že programátor nebude pracovat se standardizovaným API, získá při přímém přístupu k této implementaci řadu výhod:

```
<Configuration target="org.apache.xml.security" xmlns=" ... ">
  <!-- globální proměnné používané knihovnou -->
  <Properties>
    <Property NAME="CACertKeyStorePassword" VAL="changeit"/>
    <!-- ... -->
  </Properties>
  <!-- Algoritmy transformací -->
  <TransformAlgorithms>
    <TransformAlgorithm URI="http://www.w3.org/..."
        JAVACLASS="org.apache.xml ..." />
    <!-- ... -->
  </TransformAlgorithms>
  <!-- mapování algoritmů pro Java cryptography extension API -->
  <JCEAlgorithmMappings>
    <Algorithm URI="http://www.w3.org/2001/04/xmldsig-more#md5"
               JCEName="MD5"
               ... />
    <!-- ... -->
  </JCEAlgorithmMappings>
  <!-- ... -->
</Configuration>
```

Ukázka 4.2-2: Ukázka XML konfigurace knihovny *Apache Santuario*

Přímý přístup k DOM Všechny třídy, které reprezentují součást XML podpisu, jsou potomky třídy `SignatureElementProxy` (potažmo `ElementProxy` pro elementy z jiného jmenného prostoru, než `ds`), ze které je metodou `getElement` možné získat přímo `org.w3c.dom.Element` reprezentující tuto entitu. Pro další manipulaci s ním lze pak snadno použít libovolnou technologii pracující se standardním Java DOM API.

Nastavení kryptografických algoritmů Knihovna v základu obsahuje implementaci nejpoužívanějších algoritmů používaných při zpracování XML podpisu, ať už se jedná o algoritmy pro výpočet podpisu, otisků, provádění transformací nebo dereferencování URI. Při inicializaci knihovny je možné změnit chování knihovny nastavením příslušné systémové proměnné určující umístění XML souboru, který obsahuje mapování mezi identifikátory algoritmů (např. URI) a implementujícími třídami, jak ilustruje ukázka 4.2-2.

Globální nastavení je posléze možné změnit při konrétním použití knihovny, tzn. pro jedno určité ověření podpisu nastavit různé atributy rozdílně.

Přístup k datovým objektům Objekt třídy `Reference`, který reprezentuje element `ds:Reference`, definuje několik metod pro získání dat v různých fázích jejich zpracování:

- `getContentsBeforeTransformation` pro získání datového objektu před provedením libovolné transformace;
- `getContentsAfterTransformation` pro získání datového objektu po provedení všech transformací;
- `getNodesetBeforeFirstCanonicalization` pro provedení všech transformací s výjimkou kanonikalizačních transformací; tato metoda je v aktuální verzi knihovny (2.0.6) implementována chybně (viz 4.2-3), nerozlišuje totiž metodu c14n11 ([23]) jako kanonikalizační transformaci, což má za následek i selhání ověření knihovnou XAdES4j (3.2.2) při použití této transformace.

Přístup k datovým objektům v různé fázi jejich transformování je totiž důležitý i pro XAdES, například pro kontrolu `SignedProperties`, a to zda není výstupem některé z aplikovaných transformací množina QP, která již neobsahuje veškeré QP vložené do původního elementu (1.2.2).

Nezávislost na továrních třídách Protože každý element XML-DSig má v knihovně obraz ve vlastní konkrétní třídě, není potřeba pro vytváření elementů používat tovární třídy tak jako v JSR 105 API. Programátor tak může kdekoliv v kódu bezpečně vytvářet instance elementů podpisu přímo pomocí konstruktorů a tyto instance předávat knihovně k dalšímu zpracování.

4.2.2.1 Nevýhody knihovny

Knihovna trpí dvěma hlavními nedostatky, které se ani tak nedotýkají její funkcionality, jako spíše komfortu jejího použití. Prvním z nich je velmi nízká úroveň dokumentace – některé klíčové části kódu nejsou dokonce dokumentovány vůbec. To nutí programátora hledat funcionalitu metod přímo ve zdrojovém kódu knihovny. Jeho součástí jsou často i zakomentované příkazy, které by v případě provedení měnily chování programu, což programátora vede k zamyšlení, zda se jedná o zakomentovanou chybu, nebo základ zatím nepodporované funcionality.

Druhým nedostatkem je pak nepříliš propracovaný systém výjimek. Téměř každá metoda deklaruje vyhazování obecné výjimky třídy `XMLSignatureException`; v celé knihovně pak existují pouze čtyři konkrétní potomci této třídy a i přesto je často vyhazována přímo instance této nadtřídy. Odchytávat v programu výjimku způsobenou nějakým konkrétním problémem podpisu je pak velmi obtížné, i když ze zprávy obsažené v popisu výjimky lze vyčíst, jaký problém nastal; pro vlastní kód volající knihovnu je ale zpráva výjimky pro rozlišení příčiny jejího vyhození nepoužitelná.

V některých částech kódu knihovny také probíhá rozhodovaní na základě staticky implementovaných podmínek, které ignorují nastavení knihovny, viz například 4.2-3.

```
1  public XMLSignatureInput getNodesetBeforeFirstCanonicalization() {
2    Transforms transforms = this.getTransforms();
3    /* ... */
4    for (int i = 0; i < transforms.getLength(); i++) {
5      /* metoda má ukončit provádění transformací před první
           kanonikalizací */
6      if (uri.equals(Transforms.TRANSFORM_C14N_EXCL_OMIT_COMMENTS)
7       || uri.equals(Transforms.TRANSFORM_C14N_EXCL_WITH_COMMENTS)
8       || uri.equals(Transforms.TRANSFORM_C14N_OMIT_COMMENTS)
9       || uri.equals(Transforms.TRANSFORM_C14N_WITH_COMMENTS)) {
10       break;
11     }
12     /* ... */
13   }
14 }
```

Ukázka 4.2-3: Příklad rozhodování *Apache Santuario* implementovaného přímo ve zdrojovém kódu – kanonikalizace XML verze 1.1 jsou ignorovány

4.2.3 Zvolená technologie

Z obou srovnávaných prostředků vychází *Apache Santuario* jasně jako lepší volba. I když obsahuje některé drobné nedostatky, celkový přístup knihovny je pro implementaci XAdES mnohem vhodnější než obecné API. K tomuto API je možné se navíc kdykoliv v případě potřeby možné obrátit, protože knihovna obsahuje jeho implementaci.

4.3 Zpracování časových razítek

Pro zpracování časových razítek ([12]), které jsou jednou z možností, jak ve formátu XAdES doložit existenci datového objektu v určitém okamžiku (1.2.4), neobsahuje Java v základu žádnou podporu. Je proto potřeba využít některou z knihoven třetích stran, která tento standard implementuje. Nejrozšířenější a nejúplnější knihovnou použitelnou pro tento účel je v současnosti *Bouncy Castle*.

4.3.1 Bouncy Castle

Knihovna **Bouncy Castle** ([31]) je sbírkou implementací řady kryptografických a bezpečnostních standardů pro jazyky C# a Java. Je uvolněna pod licencí MIT.

Součásti knihovny se dají rozdělit do dvou hlavních skupin. První z nich je `Provider` pro JCE architekturu, který v porovnání se základním poskytovatelem podporuje i řadu méně obvyklých kryptografických funkcí. Druhou skupinu tvoří nízkoúrovňové API pro práci s kryptografickými objekty.

```
1  // zakódované časové razítko
2  byte [] tsBytes = ...;
3  // struktura pro libovolný podepsaný objekt ve formátu PKCS#7
4  CMSSignedData cms = new CMSSignedData(tsBytes);
5  // objekt reflektující skutečnou strukturu časového razítka
6  TimeStampToken tst = new TimeStampToken(cms);
7  // OID algoritmu otisku k~ověření integrity orazítkovaných dat
8  ASN1ObjectIdentifier algOID =
       tst.getTimeStampInfo().getMessageImprintAlgOID();
9  // standardní řetězcový identifikátor algoritmu
10 String algId = new
       DefaultAlgorithmNameFinder().getAlgorithmName(algOID);
11 // otisk orazítkovaných dat
12 byte [] digest = tst.getTimeStampInfo().getMessageImprintDigest();
13 // získání certifikátů uložených v~razítku
14 List<X509CertificateHolder> certs =
       timeStampToken.getCertificates().getMatches(new Selector() { ...
       });
15 // třída implementující logiku ověření razítka
16 SignerInformationVerifier verifier = ...;
17 // při ověření razítka je programátor již odstíněn od vnitřní
       reprezentace
18 verifier.verify(tst);
```

Ukázka 4.3-4: Zpracování časového razítka knihovnou *Bouncy Castle*

API je nízkoúrovňové zejména z pohledu přístupu ke kryptografickým objektům – programátor často pracuje přímo s vnitřní reprezentací těchto objektů (knihovna obsahuje i procesor pro čtení a zápis ASN.1 kódovaných struktur), objekty jsou identifikovány přímo OID řetězci apod. Na druhou stranu potom API poskytuje i vyšší logickou vrstvu pro provádění operací nad kryptografickými objekty – obě tyto vrstvy ilustruje ukázka 4.3-4. Skrze toto API je pak implementována velká část standardů podporovaných knihovnou (mimo jiné toto API využívá i JCE `Provider`); pro tuto práci je důležitá zejména implementace protokolu časových razítek a zajímavá by mohla být i podpora protokolu OCSP [32].

Ukázka 4.3-4 také ilustruje, že *Bouncy Castle* obsahuje veškerou podporu potřebnou pro ověření časových razítek získaných z XAdES, tzn.:

1. Získání objektu časového razítka, které je jako pole bajtů (v kódování Base64) uloženo v XAdES na řádku 6; interně je využit ASN.1 parser.

2. Získání certifikátů uložených v časovém razítku na řádku 14. Podle použitého `Selector`u může být získán jen certifikát použitý k vytvoření razítka (pro ověření integrity razítka), nebo všechny obsažené certifikáty.

3. Ověření integrity časového razítka, tedy zda je podepsáno správným certifikátem a zda tento certifikát splňuje požadavky definované v [12] na řádku 18.

4. Získání OID algoritmu otisku na řádku 8 a jeho konverze na standardní řetězcový identifikátor na řádku 10, který je potřeba pro získání instance třídy `MessageDigest`; touto instancí potom může být vypočtena hodnota otisku orazítkovaných dat a porovnána s hodnotou otisku uloženou v razítku, která je získána na řádku 12.

5. Poslední fází ověření razítka musí být ověření certifikační cesty certifikátu použitého k vytvoření časového razítka; ten byl získán v bodě 2. K jejímu ověření bude použito stejných prostředků, jako k ověření všech dalších certifikační cest, viz 4.4.

4.4 Sestavení a ověření certifikační cesty

V procesu ověření podpisu je na několika místech potřeba ověřit platnost certifikátu včetně jeho certifikační cesty v určitém okamžiku – jedná se o samotný certifikát použitý k ověření podpisu, dále pak certifikát použitý pro ověření `xades:Countersignature` (1.2.3.1), certifikáty časových razítek obsažených v podpisu (1.2.4) nebo archivní časová razítka rozšířených forem XAdES (1.2.5.5).

Standardní edice Javy v základu již obsahuje API pro práci s certifikační cestou, a to jako součást *Java PKI API*.

4.4.1 Sestavení certifikační cesty

Dokument [1] specifikuje algoritmus pro výběr certifikační cesty, který je postaven na postupu pro sestavování certifikačních cest definovaném v [33]. Nerozlišuje přitom striktně proces sestavení a ověření cesty; pro soulad s tímto dokumentem je především důležité, aby implementace korektně rozpoznávala následující stavy – u každého z nich je uvedeno, jak je možné jej detekovat pomocí PKI API, viz ukázka 4.4-5:

NO_CERTIFICATE_CHAIN_FOUND Tento výsledek má být vrácen v případě, že nebyla nalezena žádná certifikační cesta. Detekovat tento stav lze, pokud po nastavení důvěryhodných kotev, mezilehlých CA a okamžiku ověření vyhodí metoda `build` na řádku 16 výjimku.

OUT_OF_BOUNDS_NO_POE Tento výsledek má být vrácen v případě, že ve sledovaném okamžiku (nejedná se vždy nutně o aktuální čas) ještě nezapočala nebo již vypršela doba platnosti daného certifikátu. Tento stav

```
1  CertPathBuilder certPathBuilder = CertPathBuilder.getInstance("PKIX");
2  X509CertSelector certSelector = new X509CertSelector();
3  // nastavení ověřovaného certifikátu
4  certSelector.setCertificate(certificate);
5  // parametry pro vytvoření certifikační cesty
6  PKIXParameters params = new PKIXBuilderParameters(trustAnchors,
       certSelector);
7  // nastavení data ověření
8  params.setDate(validationTime);
9  // nastavení mezilehlých CA
10 CollectionCertStoreParameters ccsp = new
       CollectionCertStoreParameters(certificates);
11 CertStore othersCertStore = CertStore.getInstance("Collection", ccsp);
12 params.addCertStore(othersCertStore);
13 // vypnutí revokace - pro certifkáty v~cestě nebudou požadovány a
       použity CRL
14 params.setRevocationEnabled(false);
15 // sestavení certifikační cesty
16 PKIXCertPathBuilderResult result = (PKIXCertPathBuilderResult)
       certPathBuilder.build(params);
17 CertPath certPath = result.getCertPath();
```

Ukázka 4.4-5: Vytvoření certifikační cesty pomocí *Java PKI API*; předpokládanými vstupy jsou ověřovaný certifikát (`certificate`), okamžik ověření (`validationTime`), sada důvěryhodných kotev (`trustAnchors`) a mezilehlých certifikátů (`certificates`)

lze zjistit ještě před samotným započetím budování certifikační cesty, a to metodou `checkValidity` třídy `X509Certificate`.

REVOKED_NO_POE Tento výsledek má být vrácen v případě, že samotný ověřovaný certifikát byl revokován. Detekce revokace je součástí API a může být aktivována na řádku 14, má ale řadu nevýhod – v případě, že nejsou k dispozici revokační data, vyhazuje metoda `build` výjimku třídy `CertPathBuilderException`. Stejná výjimka je přitom vyhozena i v případě, že se nepodaří sestavit žádnou cestu a také právě v případě, že revokační data jsou k dispozici, ale certifikát je revokován. Všechny výjimky se přitom liší pouze v textové zprávě a programátor je proto nemůže v kódu rozlišit.

REVOKED_CA_NO_POE Tento výsledek má být vrácen v případě, že certifikát CA v cestě byl revokován. Bohužel i tento výsledek je obtížně detekovatelný z důvodů, které jsou uvedeny v předchozím odstavci.

TRY_LATER Tento výsledek má být vrácen v případě, že revokační data jsou k dispozici, ale nejsou považována za čerstvá. Detekce tohoto stavu je

```
1  CertPathValidator validator = CertPathValidator.getInstance("PKIX");
2  // inicializace parametrů, shodná s~ukázkou 4.4-5
3  PKIXParameters param = new PKIXParameters(trustAnchors);
4  param.addCertStore(certificates);
5  param.setDate(validationTime);
6  param.setRevocationEnabled(false);
7  // třída kontrolující správnost certifikační cesty
8  PKIXCertPathChecker checker = ...;
9  param.addCertPathChecker(checker);
10 validator.validate(certPath, param);
```

Ukázka 4.4-6: Ověření certifikační cesty pomocí *Java PKI API*; předpokládanými vstupy jsou certifikační cesta (`certPath`), okamžik ověření (`validationTime`), sada důvěryhodných kotev (`trustAnchors`) a mezilehlých certifikátů (`certificates`)

opět obtížná, navíc není možné specifikovat, kdy mají být revokační data stále považována za čerstvá.

Jak je patrné, není použití revokace v průběhu sestavování příliš vhodné. Naštěstí API umožňuje sestavit certifikační cestu bez ohledu na tato data (řádek 14) a až následně sestavenou cestu ověřit.

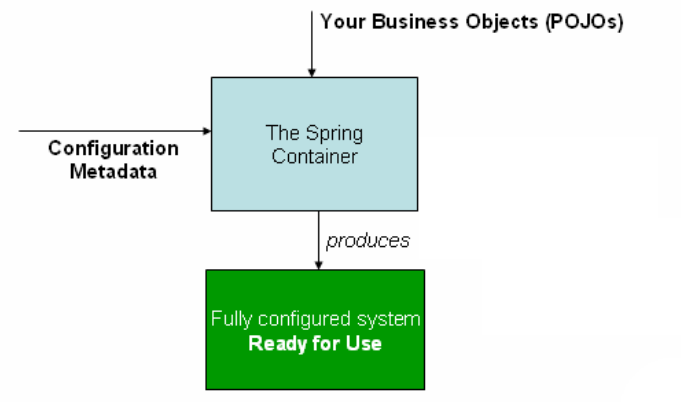
4.4.2 Ověření certifikační cesty

Certifikační cestu sestavenou tak, jak je rámcově uvedeno v ukázce 4.4-5, lze ověřit pomocí instance třídy `CertPathValidator`, viz ukázka 4.4-6. Předané objekty typu `PKIXCertPathChecker` na řádku 9 specifikují, jaké kontroly se mají provádět. Řada kontrol je již implementována v základu (revokace, kontrola použitých algoritmů), v případě neúspěchu ale všechny vyhazují výjimku třídy `CertPathValidatorException`; dochází tedy ke stejnému problému, jako při provádění kontrol v průbehu sestavování certifikační cesty – není možné rozlišit, která kontrola selhala, a podle toho rozhodnout o stavu dle [1]. V tomto případě ale existují dvě řešení:

1. Provádět ověření certifikační cesty opakovaně, pokaždé s právě jednou nastavenou kontrolou (řádek 9). V případě vyhození výjimky je potom zřejmé, která kontrola selhala.

2. Implementovat vlastní `PKIXCertPathChecker` a v případě selhání vyhazovat vlastního potomka výjimky `CertPathValidatorException`; v případě chycení této nové výjimky je pak také zřejmé, která kontrola selhala.

Obrázek 4.1: Architekturní přístup frameworku *Spring* (Zdroj: [34])



4.5 Framework pro jádro nástroje

V kapitole 5 je jádro nástroje navženo tak, aby využívalo dílčí komponenty (kontroly, různé poskytovatele služeb apod.) pro sestavení validačního prostředí a ověřovacího procesu. Zároveň se očekává, že tyto součásti budou často konfigurovatelné.

Seskládání těchto komponent ve funční celek by bylo vhodné realizovat deklarativním způsobem, aby byla snížena jejich provázanost a také aby jejich parametrizace mohla být provedena mimo kód programu. Proto by bylo vhodné mít k dispozici *inversion of control* framework, který by toto nastavení validačního prostředí prováděl.

Na základě předchozích zkušeností byl pro tuto úlohu zvolen framework *Spring*.

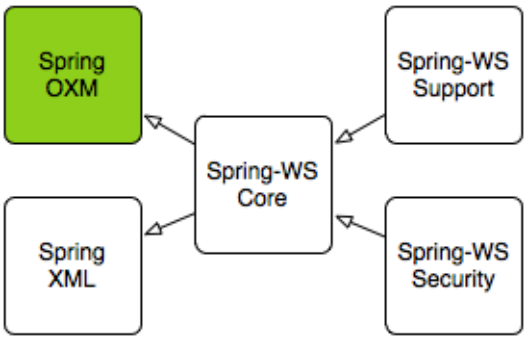
4.5.1 Spring Framework

Framework **Spring** ([34]) je Java framework populární především pro realizaci webových (*Spring MVC*) a jiných JEE aplikací. Jeho jádro (komponenta *Spring Core*) je ale na použití v *enterprise* prostředí nezávislé a je možné ho použít v libovolné aplikaci. Jádro vyvíjeného nástroje bude využívat právě tuto komponentu.

Centrálním prvkem *Spring Core* je *inversion of control* kontejner, který zajišťuje správné sestavení aplikačního prostředí (kontextu) na základě konfigurace. Hlavními součástmi jsou:

Bean factory se stará o vytváření, inicializaci a konfiguraci uživatelem definovaných komponent (nazývaných *beans*). Nespravuje ale už automaticky životní cykly těchto komponent, neumožňuje využívat mechanismu událostí apod. Jedná se o nízkoúrovňovou součást frameworku.

Obrázek 4.2: Moduly *Spring-WS* (Zdroj: [35])



Application context je nadstavbou *Bean factory*. Umožňuje spravování životního cyklu komponent, restart kontextu (znovuvytvoření všech komponent), zasílání zpráv mezi komponentami apod. Jedná se o vysokoúrovňovou součást frameworku.

Konfigurace spočívá v definování jednotlivých komponent (*beans*), typicky pomocí konfiguračního XML souboru (to ale není pravidlem). Součástí konfigurace je i deklarace závislostí mezi jednotlivými komponentami a vyplnění hodnot řídících parametrů – pro tento účel může být použit například Java *properties* formát, čímž je oddělena parametrizace a samotná konfigurace sestavení aplikačního kontextu. Při standardním využití frameworku je konfigurace načtena *Bean factory* a na jejím základě je vytvořen aplikační kontext, který je plně připraven k použití (4.1).

Další výhodou využití *Spring Core* je možnost dále na něj snadno napojovat další produkty z rodiny *Spring* (4.6), které jsou postaveny na stejném architekturním základu.

4.6 Webové služby

Protože byl v předchozí části *Spring* zvolen jako centrální framework jádra nástroje, bylo by vhodné pro realizaci webové služby využít framework, který se s ním snadno integruje.

Nejpřímočařejší volbou je v tomto případě využití další knihovny z rodiny *Spring*, a to *Spring Web Services*.

4.6.1 Spring Web Services

Framework ([35]) je dalším z projektů z rodiny *Spring* a pro nasazení vyžaduje nejméně *Spring* 3. Skládá se z modulu pro mapování mezi XML a objekty, modulu pro zpracování XML, samotného jádra frameworku a dalších podpůrných modulů, viz 4.2.

Nejdůležitější důvody pro volbu tohoto frameworku jsou:

Integrace se *Spring core* Knihovna využívá pro veškerou konfiguraci stejný mechanismus aplikačních kontextů, jako jádro *Springu*. To pro standardní implementaci jádra nástroje umožní snadno registrovat služby pro realizaci požadavků na parametrizaci rozhraní (2.3.2).

Komplexní možnosti konfigurace Knihovna podporuje široké spektrum API pro zpracování XML, jeho *marshallování* nebo mapování požadavků na konkrétní *endpointy*. Z tohoto pohledu knihovna ničím nesvazuje volbu ostatních technologií.

Podpora standardů V rámci požadavků (2.3) bylo stanoveno, že SOAP webové služby umožní využití standardů jako je například WS-Security. *Spring-WS* obsahuje podporu pro tyto standardy a umožňuje jejich snadnou integraci.

Implementaci webové služby popisuje sekce 6.4.

Kapitola 5

Návrh jádra nástroje

Tato kapitola se věnuje popisu architekturního řešení jádra nástroje. Nejprve jsou popsány klíčové komponenty nástroje (5.1), potom mechanismus rozhodování o stavu podpisu (5.2), způsob, jakým nástroj pracuje s výjimkami (5.3) a možnosti konfigurace nástroje (5.4).

5.1 Komponenty jádra nástroje

Následující část práce popisuje nejdůležitější celky, které musí být implementovány pro korektní funkčnost ověření XAdES.

5.1.1 Reprezentace podpisu

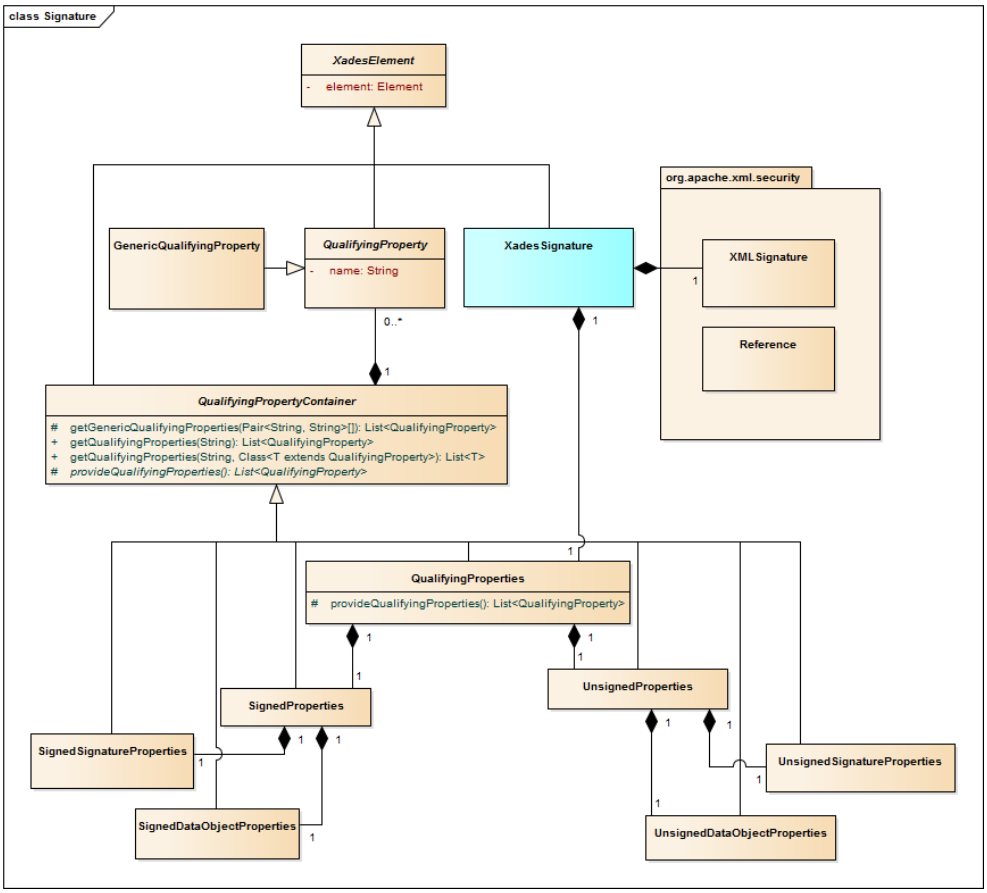
Pro potřeby ověření podpisu bylo zapotřebí obohatit strukturu XML-DSig, kterou poskytuje *Apache Santuario* (4.2.2) o nové elementy XAdES, tzn. *qualifying properties* (1.2.1). Stejně jako *Santuario* využívá jádro nástroje pro reprezentaci podpisu DOM. Procesu převodu z XML do objektové reprezentace se věnuje 6.1.

5.1.1.1 Reprezentace QP

Pro reprezentaci QP byly zavedeny dvě základní abstraktní třídy reflektující jejich strukturu uvnitř podpisu – `QualifyingProperty` a `QualifyingPropertyContainer`. Potomci první z nich reprezentují vždy jednu instanci jedné konkrétní QP, potomci druhé z nich potom element obsahující více QP, jak je patrné z diagramu 5.1. Kontejner umožňuje získávat různé QP, které obsahuje, na základě jejich typu nebo názvu.

Protože dle [5] může podpis obsahovat i další QP z různých jmenných prostorů, které tím pádem nemohou být pokryty stávajícím schématem XAdES, umožňuje kontejner i získání těchto generických QP ve formě DOM. Každý objekt reprezentující součást podpisu (včetně podpisu samotného) je totiž

Obrázek 5.1: Diagram tříd použitých pro reprezentaci podpisu. Konkrétní třídy dědící od `QualifyingProperty` jsou pro přehlednost vynechány.



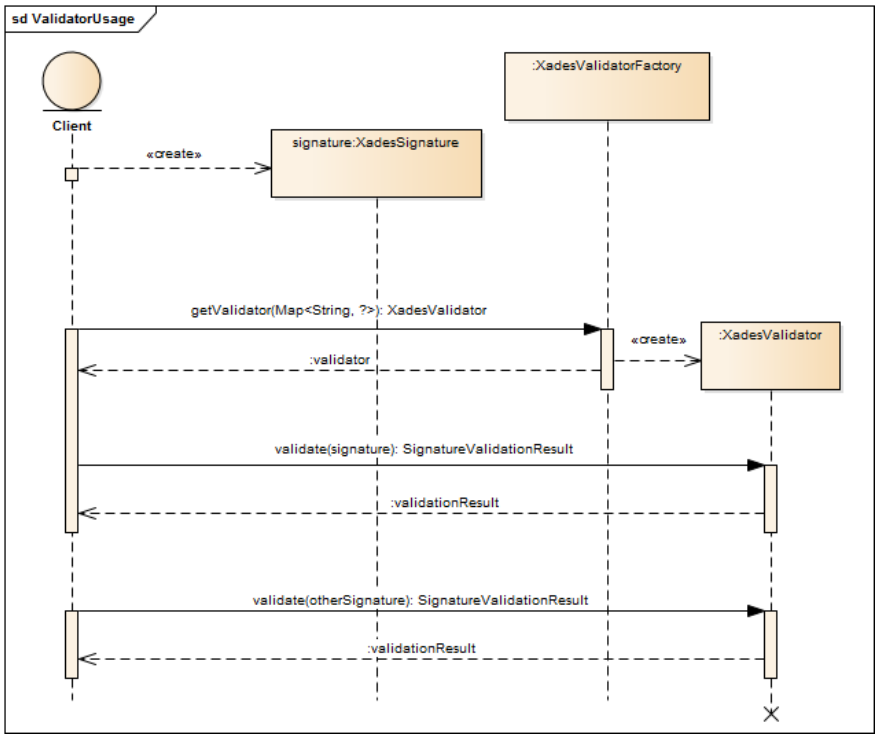
potomkem třídy `XadesElement`, která přístup k DOM zprostředkovává, a tak se k němu v případě potřeby může dostat i klient jádra nástroje.

5.1.1.2 Třída `XadesSignature`

Samotný podpis, reprezentovaný třídou `XadesSignature`, je složen ze dvou hlavních částí – základního XML-DSig tvořeného třídou `XMLSignature` a třídy `QualifyingProperties`, která obsahuje všechny QP ve struktuře popsané v části 5.1.1.1.

Třídy reprezentující podpis by dále neměly obsahovat žádnou logiku související s ověřením, pouze by měly elementům ověřovacího procesu pohodlně zprostředkovat potřebné informace. Výjimku tvoří konstrukce objektu podpisu, kdy je potřeba zjistit, zda se skutečně jedná o XAdES a zda je jeho formát nástrojem podporován (např. nástroj nebude podporovat nepřímé vkládání QP, viz 1.2.1.1). Model struktury podpisu je potom zcela nezávislý na pro-

Obrázek 5.2: Základní schéma získání a opakovaného použití validátoru



cesu ověření a jednu instanci `XadesSignature` lze opakovaně ověřovat různými validátory (5.1.2).

5.1.2 Validátor

Komponenta systému, která má na starosti samotné ověření podpisu, se nazývá *validátor*. V jádru nástroje pro ni existuje velmi jednoduché rozhraní `XadesValidator`; to obsahuje jedinou metodu `validate`, které je předán podpis a která vrací výsledek ověření. Jediná implementace, kterou jádro obsahuje (5.1.2.1), provádí ověření dle [1].

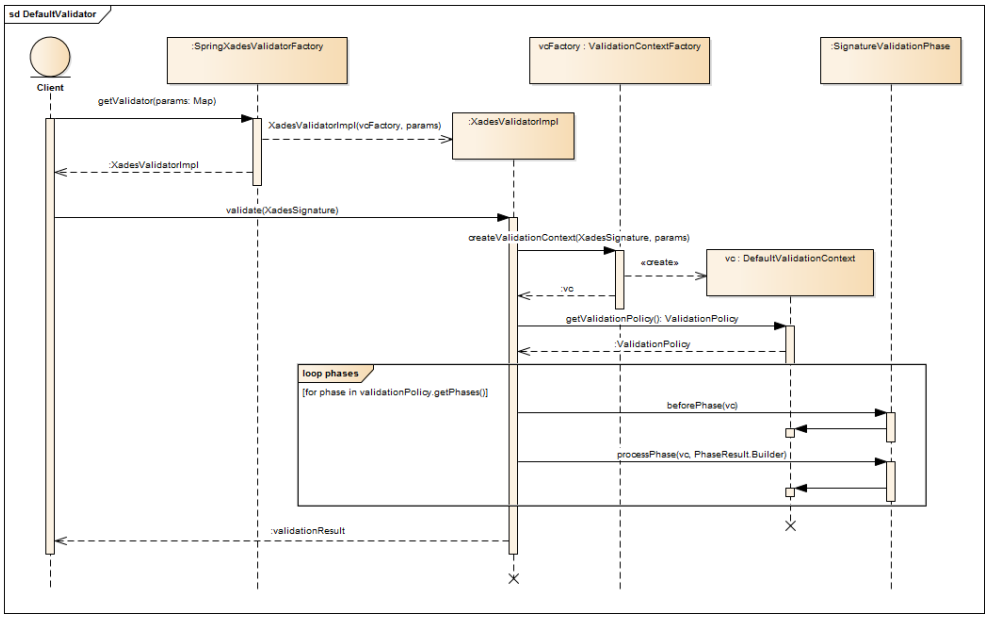
Stejně jako lze jednu instanci podpisu (5.1.1) opakovaně nezávisle ověřit několika validátory, lze i jeden validátor opakovaně použít k nezávislému ověření více podpisů, jak ilustruje diagram 5.2.

5.1.2.1 Výchozí implementace validátoru

Diagram 5.3 ukazuje základní průběh ověření podpisu výchozí implementací validátoru. Ta je pouze orchestrátorem celého procesu a postupně provádí následující kroky:

1. analyzuje podpis a získá informace o podepsaných datových objektech;

Obrázek 5.3: Sekvenční diagram ověření podpisu výchozí implementací validátoru.



2. získá z továrny instanci validačního kontextu (5.1.5);

3. z validačního kontextu získá validační politiku (5.1.4);

4. z validační politiky získá jednotlivé fáze validačního procesu (5.1.3.1) a ty postupně provádí a buduje výsledek ověření.

Celý proces je tedy závislý především na použitém validačním kontextu, respektive na tovární třídě `ValidationContextFactory`, která jej poskytuje a která je validátoru předána v konstruktoru.

5.1.2.2 Získání validátoru

Pro získání validátoru by měl klient využívat továrních tříd, konkrétně některé implementace rozhraní `XadesValidatorFactory`. Jí poskytnutý validátor by se měl chovat v souladu s nastavením této továrny, viz 5.4.

Vytváření nových validátorů skrze tovární třídu je kromě centralizace jejich konfigurace preferováno zejména z důvodu odstínění uživatele validátoru od samotného validačního kontextu (5.1.5). Ten je sestavován interně a využívá nastavení validátoru.

5.1.3 Komponenty procesu ověření

Proces ověření se skládá z komponent tří typů – fází ověření (5.1.3.1), definic kontrol (5.1.3.2) a jejich instancí (5.1.3.3). Hierarchii tříd komponent procesu ověření ukazuje diagram 5.4.

Každý element procesu ověření implementuje rozhraní `ValidationProcessElement`, které poskytuje identifikátor tohoto elementu. Ten slouží k jednoznačné identifikaci konkrétního elementu (fáze, kontroly, instance) a musí proto být mezi všemi elementy stejné úrovně unikátní.

Jednou z hlavních myšlenek celého procesu je provést ověření v co nejširším možném rozsahu, a to i v případě chyby některé z jeho komponent. Cílem validačního procesu je podchytit takovou chybu na co nejnižší možné urovni, aby nebyly zasaženy další komponenty, které jsou na sobě víceméně nezávislé. Tento princip je aplikován na různých úrovních procesu:

- Pokud selže provádění některé instance kontroly a příčina selhání je očekávatelná, je výsledek provedení této instance závislý na nastalé chybě (5.3).
- Pokud selže provádění instance kontroly z neočekávatelných důvodů nebo vytváření instancí, je výsledkem celé kontroly INDETERMINATE.
- Pokud selže provádění celé fáze ověřovacího procesu nebo selže kontrola podmínek na jejím počátku (metoda `beforePhase`), je výsledkem celé fáze INDETERMINATE.
- Pokud selže některá jiná komponeta validátoru, nemá smysl pokoušet se v procesu pokračovat a výjimka je propagována dále.

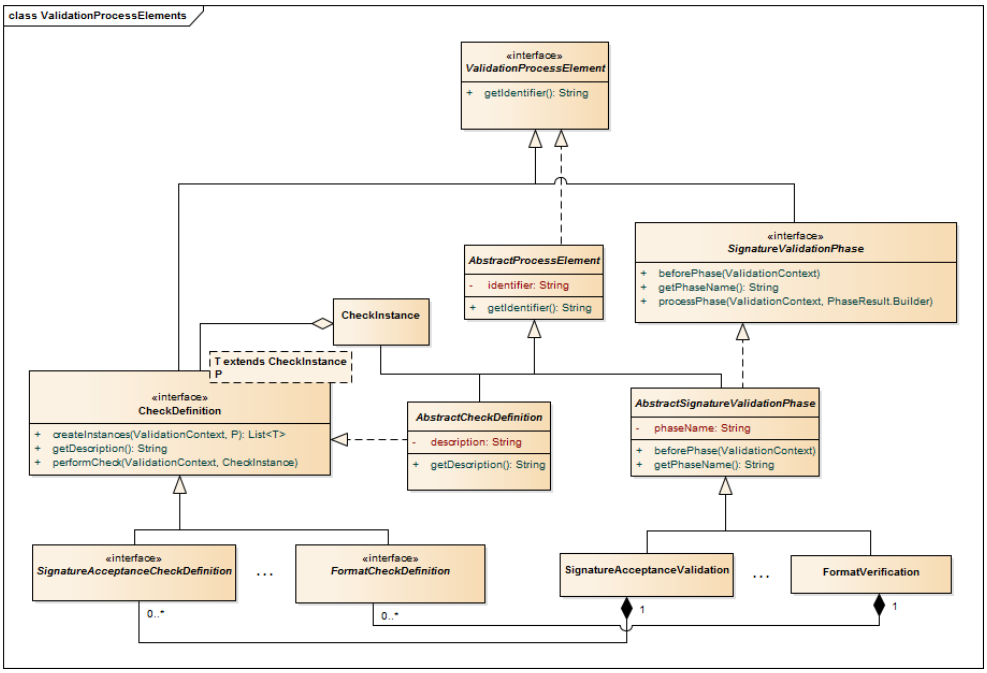
5.1.3.1 Fáze ověření

Rozhraní `SignatureValidationPhase` reprezentuje jednu fázi ověření, která zpracovává jeden konkrétní aspekt elektronického podpisu. V jádru nástroje jsou implementovány fáze definované v [1].

Konkrétní implementace fáze ověření by vždy měla být pouze orchestrátor a obsahovat tedy pouze řídící logiku. Ta může být závislá na výsledcích kontrol, které jsou prováděny v rámci fáze, jiném nastavení fáze nebo validačním kontextu. Způsob, jakým fáze definuje a provádí kontroly, není záměrně jednotný – může se jednat o prosté postupné provedení všech (1.4.3.6), nebo může být vyžadován komplexnější postup (1.4.3.4).

Fáze ověření, které mají být provedeny, určuje v jednoznačném pořadí použitá validační politika (5.1.4).

Obrázek 5.4: Hierarchie tříd použitých ve validačním procesu. Konkrétní implementace rozhraní jsou pro přehlednost vynechány.



5.1.3.2 Definice kontroly

Rozhraní `CheckDefinition` je základním rozhraním pro definici kontroly jedné určité vlastnosti podpisu. V jádru nástroje jsou implementovány kontroly požadovaných vlastností popsaných v [5] a [2] pro podpisy do úrovně T (1.3.1).

Třídy implementující toto rozhraní by měly obsahovat samotnou logiku kontroly, tedy například ověření hodnoty podpisu, platnosti certifikátu apod. Protože obecně pro každou kontrolu může existovat n jejích výskytů (např. n kontrolovaných otisků referencí), fáze by měla pro každou kontrolu, kterou chce provést, učinit následující dvojici kroků:

1. Metodou `createInstances` získat sadu instancí kontroly (5.1.3.3). Parametr třídy `P`, který je jedním ze dvou generických parametrů rozhraní, zde může být použit k předání dálších informací pro vytvoření instancí, které nejsou součástí validačního kontextu (5.1.5) – může se jednat například o výsledky některé z předchozích kontrol. Metoda vrací seznam instancí třídy `T`, konkrétního typu instance pro danou kontrolu.

2. Pro každou vytvořenou instanci zavolat metodu `performCheck`, které je instance předána. Použitím generiky pro typ instance je zaručena statická typová bezpečnost. Tato metoda může vyhazovat výjimku v případě neúspěchu kontroly, viz 5.3.

Většina kontrol je nad jedním podpisem ale provedena pouze jednou, proto vznikla pomocná třída `SingletonCheckDefinition`, která vždy produkuje jedinou instanci.

5.1.3.3 Instance kontroly

Nejmenším článkem validačního procesu je instance kontroly. Jedná se pouze o nosič informací pro provedení konkrétní kontroly společně s jednoznačným identifikátorem instance.

5.1.4 Validační politika

Použitá validační politika je jednou z nejdůležitějších charakteristik ověření podpisu, protože definuje složení validačních fází a tím i samotný proces ověření.

Validační politika je reprezentována rozhraním `ValidationPolicy`. To kromě metod `getName` a `getDescription`, které poskytují člověkem čitelný název a popis validační politiky, obsahuje především metodu `getValidationPhases`, která vrací seznam fází ověření (5.1.3.1) v pořadí, ve kterém mají být vykonány.

5.1.4.1 Určení validační politiky

V základu je validační politika zvolena podle použité implementace rozhraní `ValidationPolicyResolver`. Jeho metoda `resolvePolicy` se na základě předaného validačního kontextu pokusí politiku určit; pokud rozpoznání politiky selže, vyhazuje metoda výjimku (5.3).

Jádro nástroje obsahuje jedinou implementaci tohoto rozhraní – ta se pokusí na základě QP (1.2.1), které podpis obsahuje, určit formu základního XAdES (1.2.5). Pokud forma není rozpoznána, nebo je vyšší než T, není takový podpis v aktuální verzi nástroje podporován a je vyhozena výjimka. Jinak je zvolena validační politika obsahující kontroly pro příslušnou formu. Pokud je navíc pro validační kontext aktivován *baseline* profil, jsou do fáze přijetí podpisu (1.4.3.6) přidány další kontroly definované v [2].

V aktuální verzi nástroje také chybí podpora pro politiky explicitně definované přímo v podpisu (1.2.2.1). Pro zavedení jejich podpory je možné využít stávající mechanismus, stačí doplnit příslušnou implementaci `ValidationPolicyResolver`.

5.1.5 Validační kontext

Hlavní nosičem informací pro celý proces ověření je instance třídy `ValidationContext`. Kromě samotného podpisu poskytuje také metadata podpisu a další informace potřebné pro ověření – jedná se například o:

- identifikovaný certifikát podepisující strany,

- sadu datových objektů, které byly nástrojem v rámci analýzy podpisu identifikovány jako podepsané (6.2),
- sadu kryptografických omezení a omezení pro certifikační cestu podepisujícího certifikátu,
- validační politiku, která je použita pro ověření podpisu (5.1.4),
- identifikátor použitého profilu podpisu apod.

Validační kontext je vytvářen validátorem a pro každé ověření podpisu musí vzniknout jeho nová instance. K tomu je využito rozhraní `ValidationContextFactory` (viz 5.3).

Kontext odpovídá množině dat, která mají být získaná ve třetí fázi ověření (1.4.3.3). Ve výchozí implementaci ale vstupuje do procesu již v první fázi a data získává a poskytuje až ve chvíli, kdy si o ně některá část nástroje zažádá. Tím odpadá přítomnost této fáze v ověřovacím procesu, zároveň ale nejsou porušena pravidla předepisovaná [1], protože tato změna nemá vliv na výsledek kontroly.

5.2 Rozhodnutí o výsledku ověření

Diagram 5.5 ukazuje hierarchii tříd použitých pro reprezentaci výsledku ověření. Každá komponenta (5.1.3) poskytuje informaci o výsledku svého provedení. Pro instance kontrol je tato informace získána implicitně (VALID pokud proběhla v pořádku) nebo pomocí systému výjimek (5.3). Pro nadřazené komponenty se pak jedná vždy o agregovanou informaci z komponent podřazených – souhrnným výsledkem nadřazené komponenty je vždy nejméně příznivý výsledek ze všech podřazených, zároveň ale musí obsahovat i ostatní podřazené výsledky.

Pokud v průběhu ověřování dojde k vnitřní chybě nástroje, musí být obvykle jako výsledek vrácen stav INDETERMINATE, jak popisuje 5.1.3.

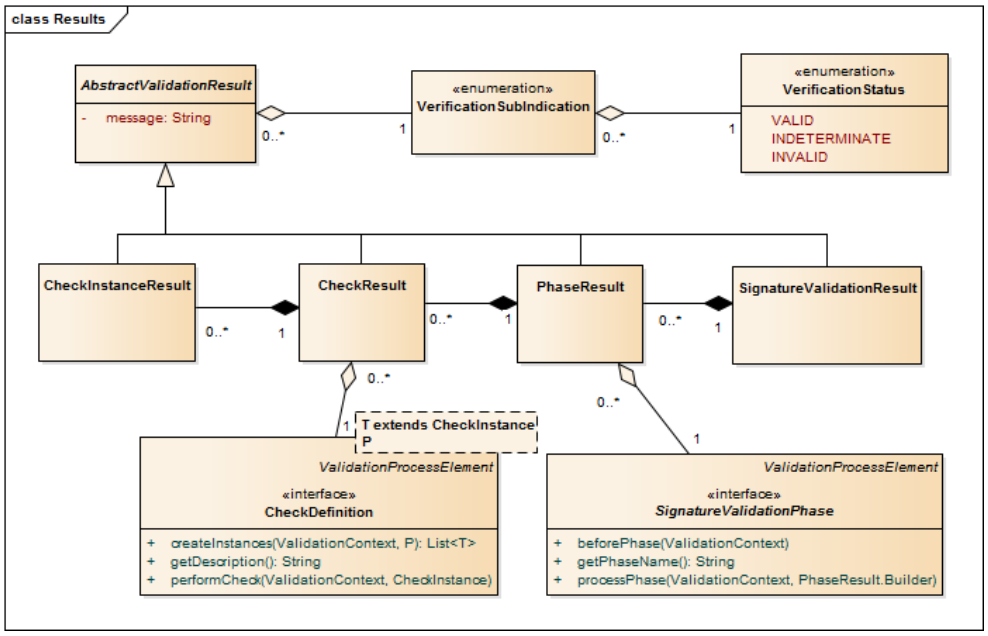
Podle [1] je součástí každého takového výsledku komponenty informace o stavu (1.4.1), volitelně pak i člověkem čitelná zpráva a případná další metadata.

5.3 Využití výjimek

Kromě specifické hierarchie výjimek pro výsledky ověření (5.3.1) používá jádro nástroje následující přístup k vyhazování a ošetřování výjimek.

`ValidatorException` Tato výjimka a její potomci je použita k signalizaci vnitřní chyby nástroje. Může se jednat například o zabalující výjimku pro chybu některé z použitých knihoven, nepodporovanou validační politiku, chybu při vytváření objektu podpisu nebo chybnou konfiguraci prostředí nástroje. Stále

Obrázek 5.5: Třídy použité k reprezentaci výsledku ověření. Hodnoty výčtového typu `VerificationSubIndication` jsou pro přehlednost vynechány.



se však ale jedná o očekávané chyby, které nástroj rozpoznává a adekvátním způsobem na ně reaguje.

`UncheckedValidatorException` Tato *runtime* výjimka je vyhozena v případě fatálního selhání nástroje v případech, které nemají jasnou příčinu a ze kterých nemá smysl se pokoušet zotavovat.

5.3.1 Výjimky pro výsledky kontrol

Pro získání výsledků provádění kontroly byly uvažovány dvě možnosti – pomocí návratové hodnoty metody `performCheck` rozhraní `CheckDefinition` (5.1.3.2), nebo pomocí vyhození výjimky v případě neúspěchu kontroly. Z následujích důvodů byl zvolen druhý způsob:

- Vyhazování výjimky v případě chyby podpisu je sémanticky správnější. Pokud kontrola neproběhne v pořádku (ať už má být výsledkem INDETERMINATE nebo INVALID), jedná se o výjimečný stav a ten by měl být ošetřen výjimkou.
- Specializovaná výjimka může obsahovat informace o výsledku kontroly stejně jako libovolný návratový typ.

```
1 // CheckDefinition
2 void performCheck(...) throws ResultAwareException;
3 // SignatureAcceptanceCheckDefinition - rozhraní pro fázi 1.4.3.6
4 void performCheck(...) throws SigConstraintsFailureException;
5 // ReferenceVerificationCheck - kontrola podepsaných referencí
6 void performCheck(...) throws HashFailureException,
    SignedDataNotFoundException;
```

Ukázka 5.3-1: Příklady hlaviček metody `performCheck` pro rozhraní `CheckDefinition` a implementující třídy

- Pokud se nejedná o *runtime* výjimku, musí být deklarována v hlavičce metody. Tyto výjimky (a tedy i stavy) musí být explicitně ošetřeny a zároveň je tím jasně dáno, jakými stavy může kontrola skončit (5.3.2).
- Vyhození výjimky ihned po zjištění chyby je programátorsky přívětivější, zvyšuje názornost a pochopitelnost kódu.

Pokud výjimka není vyhozena, předpokládá se, že kontrola proběhla v pořádku a výsledným stavem má být VALID.

5.3.2 Hierarchie výjimek pro výsledky kontrol

Základní třídou pro ohlášení neúspěšného ukončení kontroly je `ResultAwareException`, jejíž metoda `getSubIndication` vrací výsledek kontroly (jakýkoliv stav INVALID nebo INDETERMINATE) a standardní `getMessage` případnou doplňující zprávu. Tuto výjimku deklaruje rozhraní `CheckDefinition` pro metodu `performCheck`.

Od tříd implementujících toto rozhraní se pak očekává, že tuto metodu překryjí a omezí co nejkonkrétnějším způsobem deklarované výjimky na potomky `ResultAwareException`. Ukázka 5.3-1 ilustruje názornost takového kódu.

5.4 Konfigurace procesu ověření

Hlavním konfigurovatelným elementem procesu ověření je tovární třída `ValidationContextFactory` poskytující validační kontext (5.1.5). Ta umožňuje specifikovat hodnotu jakéhokoliv parametru jednoznačně identifikovaného řetězcem. Použití záleží vždy na konkrétní implementaci, ty obecně nemají povinnost použít nerozpoznané parametry. Výchozí implementace využívá pouze parametr specifikující profil podpisu.

Parametry je možné specifikovat na několika úrovních, v následujícím seznamu jsou seřazeny podle priority od nejnižší po nejvyšší (hodnota s vyšší prioritou bude použita):

- implicitní hodnoty definované přímo v tovární třídě,

- hodnoty získané z Java systémových vlastností (nastavených např. při startu JVM),
- hodnoty nastavené pro konkrétní instanci tovární třídy,
- hodnoty předané jako parametr metodě `createValidationContext`.

5.4.1 Rozhraní `ValidatorServiceProvider`

V zájmu zvýšení modularity prostředí poskytuje validační kontext přístup k rozhraní `ValidatorServiceProvider`. To umožňuje získat instanci, resp. instance tříd předaných v parametru metody `getService`, resp. `getServices`. Tím je snížena závislost mezi komponentami procesu – jakákoliv třída s přístupem k validačnímu kontextu se může pokusit získat jinou komponentu – a to nezávisle na použitém *dependency injection* mechansimu.

Poskytovatel služeb je svázán s validačním kontextem záměrně – např. pro XAdES s různými validačními politikami může být potřeba využití různých důvěryhodných kotev (rozhraní `TrustAnchorsProvider`). Jakýkoliv *dependency injection* framework by v takové situaci musel pro každé ověření provádět nové sestavení komponent; zde je odpovědnost přenesena na tovární třídu validačního kontextu, která musí dodat odpovídajícího poskytovatele služeb (a který bude pravděpodobně interně využívat některý *dependency injection* framework, viz základní implementace 6.3.2).

Celý mechanismus je v režii validačního kontextu, resp. jeho tovární třídy, takže může být případně konfigurovatelný uživatelem nástroje.

Kapitola 6

Realizace

Kapitola popisuje některé ze zajímavějších technických problémů, které musely být v rámci implementace nástroje řešeny. Dále se věnuje popisu integrace nástroje s frameworkem Spring (6.3) a implementaci rozhraní webových služeb (6.4).

6.1 Práce s XML

Zpracování elektronického podpisu vyžaduje opatrné zacházení s XML při jeho konverzi do DOM. XML je totiž v takovém případě nejen nosičem informací, ale jeho části jsou také používány jako vstupy kryptografických operací (6.1.1). XML-DSig a XAdES také využívají pokročilých možností XML, které nástroj musí umět zprostředkovat (6.1.2)

6.1.1 Zachování struktury XML

Při konverzi musí být zachována původní struktura dokumentu včetně bílých znaků, komentářů, duplicitních deklarací jmenných prostorů apod. Minimálně element `ds:SignedInfo` pro XML-DSig a `xades:QualifyingProperties` tvoří vstupy pro algoritmy výpočtu otisku – první z nich pro výpočet samotné hodnoty podpisu, pro druhý musí existovat `ds:Reference` obsahující hodnotu jeho otisku.

U podpisů, které využívají kanonikalizační transformace, není tato komplikace tak zřejmá – tyto transformace totiž obvykle provádějí stejné úpravy, jako základní implementace `DocumentBuilder` ([36]), tedy třídy parseru, která konverzi z XML do DOM provádí. Problém nastává v případě, kdy podpisy takové transformace nevyužívají (takový je případ podpisů 7.1.1); jakákoliv změna provedená parserem má za následek neúspěch kryptografického ověření podpisu.

```
1  DocumentBuilderFactory dbf =
       DocumentBuilderFactory.newInstance(DOC_BUILDER_CLASS,
       getClass().getClassLoader());
2  dbf.setNamespaceAware(true);
3  // nastavení schémat
4  dbf.setSchema(schema);
5  // kontrola XML dle schématu
6  dbf.setFeature(XmlConstants.DOC_BUILDER_F_FULL_CHECKING, true);
7  // zakázání doplnění implicitních hodnot
8  dbf.setFeature(XmlConstants.DOC_BUILDER_F_DEFAULT_VALUE, false);
9  // zakázání normalizace
10 dbf.setFeature(XmlConstants.DOC_BUILDER_F_NORMALIZE, false);
```

Ukázka 6.1-1: Nastavení `DocumentBuilderFactory`

Kód 6.1-1 ukazuje, jak lze nastavit parser tak, aby žádné takové změny neprováděl:

- je explicitně použit standardní *Xerces* parser ([36]), který rozpoznává dále specifikované parametry (řádek 1),
- protože je parseru dodáno schéma, které může specifikovat implicitní hodnoty chybějících atributů nebo elementů, musí být na řádku 8 doplnění takových hodnot zakázáno,
- musí být zakázána normalizace ([37]) na řádku 10 – v opačném případě provede změny podobné kanonikalizaci popsané výše.

6.1.2 Pokročilé vlastnosti XML

Nástroj také využívá vlastnosti vyšších úrovní DOM, především při práci s XML schématy. Mimo to je potřeba, aby parser zpracovával jmenné prostory – toho je docíleno nastavením na řádku 2 v ukázce 6.1-1

6.1.2.1 Identifikace elementů

Další z důležitých vlastností využívaných *Apache Santuario* a Java DOM API je identifikace elementů pomocí jejich ID. To není dle specifikace DOM úrovně 3 ([38]) dáno jménem atributu, ale jeho typem, který musí být `xsd:ID` – samotný atribut se pak může jmenovat jakkoliv. Aby mohl parser rozhodnout o typu elementu, potřebuje schémata definující typy elementů a jejich atributů.

Proto musí být všechna schémata, která jsou potřeba, spojena a předána tovární třídě vytvářející parser. Minimálně se musí jednat o schémata XMLDSig, XAdES a XAdES verze 1.4.1, která jsou distribuována společně s jádrem nástroje. V ukázce 6.1-1 je tak učiněno na řádku 4.

Protože řada podpisů vytvořených v rámci 7.1.2 obsahovala vlastní podepsané struktury odkazované z `ds:Reference`, bylo potřeba dodat i schémata těchto vlastních struktur, jinak by nebylo tyto struktury možné identifikovat. Nástroj samozřejmě nemůže obsahovat schémata třetích stran, proto byla implementována možnost v průběhu vytváření parseru specifikovat další schémata a do rozhraní WS (6.4) byla přidána možnost tato schémata předat.

6.1.2.2 Kontrola XSD

Specifikace schémat přináší další i výhodu – celé XML podpisu může pomocí nich být zvalidováno; tato kontrola je aktivována v ukázce 6.1-1 na řádku 6. Provedeny jsou pak nejen elementární kontroly, jako jsou přítomnost elementů nebo atributů, ale i kontroly datových typů včetně kontroly duplicitních ID, korektních kódování Base64 apod.

Nástroj může potom detekovat chyby podpisu ještě předtím, než je rozbalen do objektové struktury; v případě úspěchu potom může naopak bezpečně předpokládat, že všechny povinné hodnoty jsou přítomny a jsou správného typu.

6.2 Zpracování `Manifestů`

Zpracování vnořených manifestů (1.1.1.2) se přímo nevěnuje žádná z implementací srovnávaných v kapitole 3 – *XAdES4j* i *Digital Signature Service* ale interně využívají *Apache Santuario*, které umožňuje rekurzivní průchod manifesty.

Tento přístup má pro využití v XAdES několik nedostatků:

Algoritmus pouze ověřuje otisk Už ale neposkytuje strukturu, která je v podpisu použita – kde se jaký manifest nachází, jaké reference obsahuje a odkud vede reference na něj.

Algoritmus nelze ovlivnit V praxi jsou manifesty používány z různých důvodů – datový objekt je podepsán více podpisy (pak stačí spočítat jeho otisk pouze jednou a pro další použít právě manifest, z kterého je v případě objemných datových souborů otisk snazší vypočítat), jindy může podpis být považován za platný, i když datové objekty odkazované manifestem nejsou dostupné. V různých situacích tak může být potřeba procházet manifesty do různé hloubky nebo dle jiných parametrů.

Algoritmus neuvažuje QP `DataObjectFormat` To je pochopitelné, protože se jedná o rozšíření XAdES. Podepsané datové objekty je tedy potřeba získat pro spárování s těmito QP jiným způsobem.

Problém správného rozpoznání a zpracování manifestů lze zobecnit na rozpoznání a zpracování podepsaných datových objektů, resp. referencí na ně. Třída `SignedReferencesInfo` realizuje tuto potřebu – skrz `ds:SignedInfo` rekurzivně zpracovává všechny odkazované manifesty a buduje strukturu provázaných referencí. Její podtřídy pak mohou rozhodovat o tom, které z nich mají být ověřovány (metoda `getReferencesToBeVerified`) a pro které má existovat QP `DataObjectFormat`.

Základní implementace umožňuje pouze omezit maximální hloubku ověřovaných referencí – v rámci testování nevznikla potřeba využít složitější mechanismus.

6.3 Integrace s frameworkem Spring

Tato část práce krátce popisuje způsob integrace jádra nástroje s frameworkem *Spring*, zejména způsob využití jeho mechanismu vkládání závislostí.

6.3.1 Konfigurace závislostí

Spring poskytuje dvě základní cesty, jak nakonfigurovat závislosti mezi komponentami (*beans*) – pomocí XML konfigurace a pomocí anotací. Pro jeho využití v jádru nástroje byla z následujících důvodů zvolena první z možností:

- Při použití anotací vzniká silná vazba mezi zdrojovým kódem vyvíjeného nástroje a knihovnami, které využívá. Protože velká část komponent – implementace kontrol, fází, či politik, viz 5.1.3 – by měla být použitelná *různými* frameworky pro vkládání závislostí nebo dokonce zcela samostatně, je použití anotací konkrétních knihoven nepřípustné.
- XML odstiňuje konfiguraci nástroje od samotného kódu nástroje. Ten v důsledku není svazován závislostí na dalších komponentách. Tím je naplněna architekturní myšlenka vytváření elementů ověřovacího procesu – samostatných uzavřených částí kódu, které jsou podle aktuální konfigurace zapojitelné do libovolné přípustné části procesu.

Protože XML konfigurace vyžaduje pouze *settery* pro třídní proměnné [3], které mají být v průběhu vkládání závislostí inicializovány, jsou komponenty ověřovacího procesu od *Springu* kompletně odstíněny.

6.3.2 Třídy přímo využívající *Spring*

Následující tři třídy poskytují základní implementaci důležitých rozhraní využívaných jádrem nástroje. K naplnění své funkcionality přitom využívají aplikačního kontextu.

[3] *Spring* umožňuje použít i vkládání závislostí skrz konstruktory, tuto možnost ale nástroj nebude využívat.

`SpringXadesValidatorFactory` je tovární třídou pro vytvoření validátoru (5.1.2). Jedná se o jedinou třídu v jádru nástroje implementující rozhraní `ApplicationContextAware`, a proto jako jediná obdrží aplikační kontext při startu kontextu přímo od *Springu* – ostatním třídám využívajícím *Spring* kontext sama předává.

Pokud klientská aplikace nepoužívá *Spring* nebo chce z nějakých důvodů zůstat od interní implementace odstíněna, je možné instanci továrny vytvořit přímo přes konstruktor – tomu lze volitelně předat aplikační kontext, se kterým bude nadále pracovat, jinak je automaticky vytvořen interní kontext podle konfigurace jádra, `spring-validator.xml`.

`SpringValidationContextFactory` je tovární třídou pro vytvoření validačního kontextu (5.1.5). Je využívána `SpringXadesValidatorFactory` v případě, že se klientská aplikace nerozhodne použít jinou implementaci. Využívá aplikačního kontextu pro sesbírání všech *bean* implementujících ta rozhraní, která nástroj rozeznává, jako jsou poskytovatelé důvěryhodných kotev, časových značek a dalších služeb, kterými validační kontext inicializuje. Zároveň provádí inicializaci použitého profilu.

`SpringValidatorServiceProvider` je výchozí implementací poskytovatele služeb (5.4.1). Z aplikačního kontextu vybírá vhodné komponenty pro realizaci služeb, zároveň se stará o inicializaci validačního kontextu ve třídách těchto služeb, pokud implementují rozhraní `ValidationContextAware` (6.3.4).

6.3.3 *Scope* komponent

Některé komponenty validačního procesu (v rámci *Springu* se jedná o *beans*) může být potřeba inicializovat různým způsobem pro různá ověření, typicky pokud má být některá kontrola vynechána. Taková konfigurace na globální úrovni by jinak zasahovala i do průběhu ostatních ověření, což je nežádoucí.

Spring umožňuje pro každý *bean* specifikovat parametr *scope*, který ovlivňuje, zda v konkrétní situaci bude pro požadovaný *bean* vrácena stávající instance nebo bude vytvořena nová. Pokud není zadán, je jeho hodnota *singleton*, což znamená, že *bean* bude vytvořen právě jednou a bude používán po celý běh kontextu. Další přípustnou hodnotou je *prototype*; pro takový *bean* bude vytvořena pokaždé nová instance.

Pro použití nástrojem by bylo vhodné implementovat *scope* tak, aby byly instance vytvářeny zvlášť pro každý validační kontext – podobný *scope* existuje například ve *Spring MVC*, kdy je *bean* vytvářen znovu pro každý HTTP požadavek. Implementace takového *scope* je tedy možná, ale poměrně komplikovaná a vzhledem k tomu, že je aktuálně každý *bean* v průběhu validace vyžadován nejvýše jednou, je možné bezpečně použít *prototype*.

6.3.4 Rozhraní `ValidationContextAware`

Některé komponenty, kterými může být v budoucnu funkcionalita nástroje rozšířena, mohou vyžadovat přístup k validačnímu kontextu, i když jim nebude v rámci procesu explicitně předán.

Pro takové komponenty vzniklo rozhraní `ValidationContextAware` s jedinou metodou `setValidationContext`, kterou je dané komponentě validační kontext nastaven. Nastavení provádí poskytovatel služeb (6.3.2) a je aktivováno pouze pro *prototype beany* (6.3.3), u kterých je možné bezpečně předpokládat, že jejich instance bude skutečně použita pouze pro aktuální ověření.

6.4 Implementace webových služeb

Pro realizaci webové služby byl zvolen framework *Spring Web Services* (4.6), a to především kvůli snadné integraci s jádrem nástroje.

Centrálním místem konfigurace je opět konfigurační XML, `spring-ws--servlet.xml`, kde jsou definovány dodatečné komponenty rozšiřující jádro. Samotná konfigurace jádra musí být nejprve do tohoto souboru importována, aby byly načteny všechny základní komponenty (řádek 2 v ukázce 6.4-3).

Důležitým konfiguračním tagem je také ten na řádku 27, který umožňuje zpracování anotací v třídě endpointu (6.4.1). Také je použit *Axiom*, StAX implementace webových služeb, viz řádek 13. Ten mimo jiné umožňuje odkládání velkých příloh požadavků na disk, čímž snižuje riziko zahlcení paměti. Pro zvýšení efektivity přenosu dat je navíc využit standard MTOM ([39]).

Další konfiguraci je ještě potřeba provést ve `web.xml` – zde je nutné zaregistrovat `MessageDispatcherServlet`, *Spring* komponentu zpracovávající příchozí HTTP požadavky a transformující je na požadaky na webovou službu. Na základě jména servletu (*spring-ws*) je potom automaticky nalezen výše zmíněný XML konfigurační soubor.

6.4.1 Realizace endpointu

V ukázce 6.4-2 je zachycena veškerá konfigurace provedená v kódu třídy `ValidatorEndpoint`, která umožní zpracování požadavku:

- anotace `Endpoint` na řádku 1 umožní zpracování následujících anotací na úrovni metody;
- anotace `PayloadRoot` na řádku 3 značí, že metoda obsluhuje požadavky; její parametry specifikují kořenový element požadavku, který metoda zpracovává;
- anotace `ResponsePayload` na řádku 6 signalizuje, že návratová hodnota metody má být použita pro sestavení odpovědi;

```
1  @Endpoint
2  public class ValidatorEndpoint implements ApplicationContextAware {
3    @PayloadRoot(
4             localPart = "ValidatorRequest",
5             namespace = "http://fit.cvut.cz/svorcmar/validator")
6    @ResponsePayload
7    public ValidatorResponse handleValidatorRequest(
8        @RequestPayload ValidatorRequestType validatorRequestType)
9        throws ValidatorServiceException {
10   ...
11   }
12 }
```

Ukázka 6.4-2: Třída `ValidatorEndpoint` a hlavička její metody pro obsluhu požadavku na webovou službu

- anotace `@RequestPayload` na řádku 8 signalizuje, že parametr metody se má konvertovat některým z podporovaných mechanismů z těla požadavku.

Pokud je v průběhu zpracování požadavku vyhozena výjimka, je automaticky převedena na SOAP *Fault* odpověď.

Endpoint je potřeba zaregistrovat v aplikačním kontextu (řádek 8 v ukázce 6.4-3), protože aplikace nevyužívá automatické skenování komponent tagem `<ctx:component-scan/>`.

Třída `ValidatorRequestType`, jejíž instance je jediným parametrem metody, a třída `ValidatorResponse`, která je návratovým typem metody, jsou automaticky generovány na základě schématu vytvořeného pro webovou službu. Pro generování je použit JAXB2 *Maven* plugin, to probíhá při každé kompilaci aplikace.

Pro takto vygenerované třídy obsahující JAXB2 anotace poskytuje *Spring-WS* nativní podporu v rámci OXM modulu (viz 4.2). Programátor proces konverze mezi objektem a XML nemusí řešit.

6.4.2 Registrace nových služeb

Webová služba umožňuje na vstupu předat množinu důvěryhodných kotev nebo mezilehlých certifikátů, které by měly být zaregistrovány v aktuálním validačním kontextu. Výchozí implementace `ValidationContextFactory` (6.3.2) využívá aplikačního kontextu *Springu* k nalezení vhodných služeb, proto stačí pro tyto služby zaregistrovat *bean* implementující vhodné rozhraní (řádek 6 v ukázce 6.4-3).

Zaregistrování nových komponent tímto způsobem je také dostačující pro jejich nalezení výchozím poskytovatelem služeb (6.3.2).

Protože data předaná společně s požadavkem (certifikáty, datové objekty) mají platnost pouze pro aktuální požadavek, je potřeba, aby byla skutečně použita jenom pro něj. K tomu je využito `ThreadLocal` proměnných navázaných na obsluhující vlákno, které je po ukončení požadavku potřeba explicitně vyčistit (6.4.3).

6.4.3 Vyčištění požadavku

Po úspěšném nebo neúspěšném vyřízení požadavku je obvykle potřeba uvolnit zdroje, které byly pro vyřízení alokovány. Pro takové účely poskytuje *Spring-WS* rozhraní `EndpointInterceptor`, jehož metoda *afterCompletion* se po zaregistrování interceptoru (v ukázce 6.4-3 na řádku 23) vykoná po ukončení každého požadavku.

Konkrétní implementace `RequestCleanupInterceptor` pak zavolá nad všemi *beany* v aktuálním aplikačním kontextu, které implementují rozhraní `RequestCleanup`, metodu `cleanup`. Toto rozhraní by měly implementovat všechny komponenty, které pracují s *ThreadLocal* proměnnými (aby bylo zabráněno únikům paměti) a také komponenty pracující s proudy, které je po dokončení požadavku potřeba uzavřít.

```
<!-- import nastavení jádra nástroje -->
<import resource="classpath:spring-validator.xml"/>
<!-- služba pro dereferencování URI na lokální objekty v~požadavku -->
<bean id="uriResolverRequest" class="cz.cvut...URIResolverRequest"/>
<!-- služba pro získání důvěryhodných kotev z~požadavku -->
<bean id="requestTrustAnchorsProvider"
    class="cz.cvut...RequestTrustAnchorsProvider"/>
<!-- registrace endpointu pro zpracování požadavku -->
<bean id="endpoint"
    class="cz.cvut.fit.svorcmar.validator.service.ValidatorEndpoint">
  <property name="requestTrustAnchorsProvider"
      ref="requestTrustAnchorsProvider"/>
  <property name="requestOtherCertificatesProvider"
      ref="requestOtherCertificatesProvider"/>
</bean>
<!-- Axiom implementace zpráv -->
<bean id="messageFactory"
    class="org.springframework.ws.soap.axiom.AxiomSoapMessageFactory">
  <property name="attachmentCaching" value="true"/>
</bean>
<!-- publikace XSD s~typy elementů používanými službou -->
<bean id="validator"
    class="org.springframework.xml.xsd.SimpleXsdSchema">
  <property name="xsd" value="classpath:validator.xsd"/>
</bean>
<!-- statická publikace WSDL -->
<sws:static-wsdl location="classpath:validatorService.wsdl"
    id="validatorService"/>
<!-- interceptor pro vyčištění zdrojů po obsloužení požadavku -->
<sws:interceptors>
  <bean class="cz.cvut...RequestCleanupInterceptor"/>
</sws:interceptors>
<!-- anotacemi řízené sestavení kontextu -->
<sws:annotation-driven/>
```

Ukázka 6.4-3: *Spring* konfigurační soubor pro webovou službu (zkráceno)

Kapitola 7

Testování

Kapitola představuje testování prováděné v rámci vývoje nástroje. Soustředí se zejména na popis testovacích scénářů pro jádro nástroje (7.1) a testování webové služby (7.2).

V průběhu vývoje nástroje byly také prováděny manuální testy a automatizované unit testy; ty pomohly v rané části vývoje odhalit některé chyby, ale jejich popis ale není příliš zajímavý a proto se jimi práce nezabývá.

7.1 ETSI Plugtesty

Pro účely testování implementací jednotlivých norem, především pak jejich interoperability, pořádá ETSI každoročně tzv. *Plugtesty*. Jedná se o události vázané vždy k určitému standardu, v rámci kterých účastníci implementující tento standard zasílají výstupy svých implementací a ostatní účastníci tyto výstupy verifikují.

Protože ETSI normy jsou obvykle základem pro legislativní nařízení EU, účastní se těchto událostí především oficiální zastoupení různých institucí členských států, ale také soukromé nebo akademické subjekty. Plugtesty jsou organizovány vzdáleně (veškerá interakce s ostatními účastníky probíhá přes internet) a účast na nich je bezplatná.

Pro testování implementací norem týkajících se pokročilých podpisů účastníci nejprve podle zadání vygenerují elektronický podpis. Zadání obvykle předepisuje formu podpisu, podpisový certifikát a další vlastnosti výsledného podpisu. Ostatní potom tyto podpisy ověřují podle platných norem a podle těchto předepsaných pravidel. Účastníci nemají povinnost podpisy vytvářet ani ověřovat – některé subjekty je pouze generují, některé pouze ověřují, což je případ této práce.

Pro otestování funkčnosti jádra vytvořeného nástroje byly vedoucím práce doporučeny podpisy vygenerované v rámci Plugtestu 2015 (7.1.1) a 2016 (7.1.2), který proběhl v dubnu ([40]).

7.1.1 Plugtest 2015

První skupinu testovacích dat tvoří podpisy z loňského běhu události. Narozdíl od letošního ročníku (7.1.2) nebyla platnost podpisů vázána na TSL ([26]).

Účastníkům byla na počátku události dána k dispozici sada kryptografických dat pro podepisování (privátní klíče, datové objekty, politiky) a ověřování (certifikáty CA, TSA a atributové certifikáty). Součástí vstupních dat byla i sada negativních testovacích scénářů (tedy scénářů, ve kterých podpisy nejsou vytvořeny zcela dle standardu a cílem validátoru je detekovat záměrně obsažený nedostatek) pro rozšířené formy XAdES a *baseline* profil – druhá z nich byla použita k otestování správné implementace profilu nástrojem.

7.1.1.1 Negativní testy *baseline* profilu

Pro *baseline* profil balíček obsahoval celkem dvacet testovacích scénářů, z toho devět pro formu B, sedm pro formu T a čtyři pro dlouhodobé formy (1.3.1). Nástroj byl testován všemi scénáři pro formy B a T.

Pro formu B se jednalo a následující testy:

- vstupní XAdES neobsahuje vždy právě jednu z povinných QP `SigningTime`, `SigningCertificate` a `DataObjectFormat`,
- vstupní XAdES obsahuje nesprávnou hodnotu podpisu (dešifrovaná hodnota neodpovídá hodnotě otisku vypočteného z transformovaného `ds:SignedInfo`),
- k vytvoření XAdES byl použit nedůvěryhodný certifikát (nebyla nalezena certifikační cesta, protože certifikát nebyl vydán CA obsaženou v množině důvěryhodných kotev),
- k vytvoření XAdES byl použit certifikát, jehož doba platnosti již vypršela,
- k vytvoření XAdES byl použit certifikát, který byl odvolán,
- k vytvoření XAdES byl použit certifikát vydaný CA, jejíž certifikát byl odvolán,
- vstupní XAdES v QP `SigningCertificate` obsahuje nesprávný otisk certifikátu podepisující strany.

Pro formu T se jednalo a následující testy, k jejich provedení bylo zapotřebí extrahovat časovou značku obsaženou v QP `SignatureTimeStamp`:

- k vytvoření XAdES byl použit certifikát, jehož doba platnosti vypršela již k datu uvedenému v časové značce,
- k vytvoření XAdES byl použit certifikát, který byl již k datu uvedenému v časové značce odvolán,

- otisk obsažený v časové značce neodpovídá otisku elementu `ds:SignatureValue`,
- k vytvoření časové značky byl použit certifikát, jehož doba platnosti k datu uvedenému v časové značce již vypršela,
- k vytvoření časové značky byl použit certifikát, který byl již k datu uvedenému v časové značce odvolán,
- certifikát použitý k vytvoření časové značky nebyl vydán důveryhodnou CA,
- certifikát použitý k vytvoření časové značky byl vydán důveryhodnou CA, jejíž certifikát byl ale odvolán.

7.1.1.2 Výsledky testování

Nástroj správně vyhodnotil stav podpisů ze všech testovacích scénářů s výjimkou těch, ve kterých byla potřeba revokační data – seznamy CRL se totiž v průběhu Plugtestu stahovaly z URL uvedených v certifikátech CA, které ale již při provádění testů nebyly dostupné. Proto byla pro všechny scénáře vypnuta kontrola revokačních dat (viz 4.4.2).

Tyto testy pomohly ověřit elementární funcionalitu nástroje, zejména byla prokázána funkčnost těchto jeho částí:

- rozpoznání formy podpisu,
- identifikace certifikátu podepisující strany,
- aplikace specifikovaného profilu,
- sestavení certifikační cesty (bez použití revokačních dat) obsahující mezilehlé certifikáty,
- kryptografické ověření, tedy kontrola otisků podepsaných datových objektů a kontrola hodnoty podpisu,
- zpracování časového razítka (správné načtení hodnoty, kontrola otisku, kontrola podpisu).

7.1.2 Plugtest 2016

Poslední z uspořádaných Plugtestů si kladl za cíl otestovat implementace nového standardu [41]. Tento dokument je nástupcem [1] a klade si za cíl specifikovat průběh ověření tak, aby mělo právní platnost na území EU. Také všechny podpisy vytvořené účastníky měly být vytvořeny v souladu s [26], tedy podepsány certifikáty vydanými CA na seznamech TSL. Zároveň měly být podpisy vytvářeny pokud možno dle *baseline* profilu.

7.1.2.1 Výsledky testování

Hlavním nedostatkem nástroje v tomto Plugtestu bylo, že není specializovaný na ověřování kvalifikovaných podpisů dle [26], a proto v současném stavu není schopen získat seznam důvěryhodných autorit ze seznamů TSL. To lze částečně obejít předáním certifikátů příslušných CA společně s požadavkem na ověření; stále však nebude možné ověřit zda se jedná o kvalifikované certifikáty.

To by bylo možné vyřešit implementací nové validační politiky a příslušnou konfigurací nástroje; nová politika by důvěřovala pouze certifikátům vydaným autoritami na seznamech TSL. Nástroj je pro umožnění takových rozšíření navržen, ale jejich implementace je mimo rozsah této práce.

Díky velmi variabilní podobě podpisů bylo nalezeno a opraveno i několik chyb nebo nedostatků na straně jádra nástroje i webové služby:

- Některé subjekty podepisovaly své vlastní XML struktury, které bez schématu nebylo možné identifikovat. Řešení problému je popsáno v 6.1.2, testy potvrdily funkčnost tohoto řešení.
- Formát některých podpisů nebyl v souladu s [5], nástroj takové podpisy ale neodmítl ihned, což způsobovalo chybu v pozdějších fázích ověření.
- Nástroj nebyl schopen načíst XML, jehož kořenovým elementem nebyl `ds:Signature`. Někteří účastníci totiž podpisy vnořovali do struktur obsahujících společně s podpisem další vlastní metadata.
- Při načítání některých z podpisů se ztrácely redundantní atributy s deklaracemi jmenných prostorů, což mělo za následek nesprávně vypočtené hodnoty otisků. Řešení problému je popsáno v 6.1.1.
- U některých podpisů byla nesprávně rozpoznávána časová razítka.

7.2 Testování webových služeb

K testování rozhraní byl využit klient SoapUI ([42]). Ten na základě poskytnutého WSDL umožňuje generovat požadavky, odesílat je na server a uživateli zobrazit odpověď. Také obsahuje podporu pro řadu standardů, mimo jiné také MTOM.

Testování bylo zaměřené na odhalení chyb ve webové službě, nikoliv v jádru nástroje.

Aplikace byla pro účely testování nasazena na Jetty server ([43]).

7.2.1 Testovací scénáře

- Požadavek neobsahuje XML dokument s podpisem, nástroj musí ohlásit chybu.

- XML dokument s podpisy obsahuje více podpisů, všechny musí být ověřeny.
- Požadavek neobsahuje důvěryhodnou kotvu a není možné sestavit certifikační cestu.
- Podepsaná reference odkazuje na webový zdroj, ten ale není dostupný; podepsaný datový objekt je předán společně s URI této reference jako příloha a použije se místo původního zdroje.
- XML dokument s podpisy je přiložen k požadavku jako příloha podle standardu MTOM, ověření musí proběhnout v pořádku.

7.2.2 Výsledky testování

Testování pomohlo najít a odstranit tyto chyby:

- Při dlouhém běhu programu a velkém množství zpracovaných požadavků docházela virtuálnímu stroji paměť. To bylo způsobeno nekompletním uvolněním zdrojů po zpracování požadavku (6.4.3).
- Z podobného důvodu nebyly uvolňovány dočasné soubory použité pro odkládání příloh požadavku, což způsobovalo jejich hromadění na disku.
- Některé procesní fáze neměly nastavený *prototype scope* (6.3.3), což způsobovalo jejich nesprávné vyhodnocení.

Naopak byla ověřena správná funkčnost rozhraní v následujících směrech:

- Je využíván aplikační kontext jádra nástroje, rozhraní do něj korektně integruje nové služby.
- Rozhraní správně přijímá parametry předané společně s požadavkem a ty správně ovlivňují proces ověření.
- Odpověď služby poskytuje všechny informace, které má.
- Velké přílohy jsou odkládány na disk a jsou správně přijaty i v případě, že jsou ve formátu MTOM.

Závěr

V této práci bylo po seznámení se s důležitými podklady, analýze požadavků a rešerši stávajících řešení navrženo jádro nástroje pro ověřování elektronických podpisů ve formátu XAdES. Návrh byl následně realizován a společně s ním byla implementována i jednoduchá webová služba, která jádro nástroje využívá.

Nejprve byly představeny normy a standardy, které definují formát a předepisují způsob práce s ním. Na základě zadání práce a informací sesbíraných z těchto a dalších dokumentů byla potom provedena analýza a specifikace požadavků. Následně byla zvolena implementační platforma a konkrétní technologie, které realizují dílčí potřeby plynoucí z těchto požadavků a které umožnily se soustředit na samotný ověřovací proces a zpracování XAdES.

Rešerše stávajících řešení pomohla najít některé nedostatky těchto řešení, kterým se vyvíjený nástroj snažil vyhnout. Zároveň umožnila poznat postupy, kterými jejich autoři přistupují k různým aspektům ověření XAdES a inspirovat se jimi pro vlastní implementaci.

Jádro nástroje bylo navrženo tak, aby umožňovalo realizovat všechny vytyčené požadavky; jeho hlavní myšlenkou bylo umožnit definovat, konfigurovat a provádět komplexní proces ověření, který bude poskytovat co nejúplnější komplexní výstupy. Návrh počítá s rozšiřitelností nástroje, pro řadu mechanismů (jako jsou explicitně definované podpisové politiky) obsahuje podporu, kterou ale prototyp nástroje neimplementuje.

Podle vzniklého návrhu byla implementována klíčová funkcionalita jádra nástroje. Společně s ní byla implementována sada kontrol pro *baseline* profil a SOAP webová služba, která umožňuje pohodlnou práci s nástrojem.

Jádro nástroje i webová služba byly důkladně otestovány a nalezené chyby opraveny. Také nalezené nedostaky byly dodatečně implementovány. Výsledek splňuje všechny body zadání práce a realizuje všechny požadavky, které si práce vytyčila.

Výhled do budoucna

Ačkoliv prototyp nástroje je plně funkční s ohledem na vydefinované požadavky, XAdES a přidružené standardy jsou velmi obsáhlé, a proto exisuje stále mnoho dalších funkcí, které by nástroj mohl realizovat. Mezi nejdůležitější z nich patří:

Podpora aktualizovaného standardu [41] V průběhu psaní práce vydalo ETSI aktualizovanou podobu [1], klíčového dokumentu specifikujícího algoritmus ověřování pokročilých elektronických podpisů. Tento dokument v porovnání s původním (který byl určen zadáním práce) obsahuje řadu drobných změn, které by bylo vhodné do budoucna implementovat.

Podpora dalších formem XAdES Nástroj nyní podporuje ověřování podpisů pouze formy T a nižších. Společně s implementací vyšších forem by bylo vhodné zahrnout i podporu pro vyšší úrovně *baseline* profilu.

Implementace dalších AdES formátů V průběhu seznamování se s podklady pro implementaci nástroje, jeho návrhu i samotné realizace se čím dále tím jasněji ukazovalo, že ověřovací proces je do značné míry nezávislý na formátu pokročilého elektronického podpisu. Pokud by byla jeho současná implementace zobecněna (validační kontext je nyní silně vázán na XAdES), mohl by proces být využit i pro další formáty pokročilého podpisu ([15], [11]).

Podpora podpisových politik dle [20] Nástroj v současnosti nepodporuje explicitně specifikované podpisové politiky, zejména proto, že jejich podoba není důsledně standardizována. Dokument [20] se tento nedostatek snaží řešit definováním XML schématu pro strojově zpracovatelný popis obecné podpisové politiky.

Literatura

[1] European Telecommunications Standards Institute: *Signature validation procedures and policies v1.2.1* [online]. TS 102 853, ETSI, Prosinec 2014, [cit. 27. 2. 2016]. Dostupné z: http://www.etsi.org/deliver/etsi_ts/102800_102899/102853/01.02.01_60/ts_102853v010201p.pdf

[2] European Telecommunications Standards Institute: *XAdES Baseline Profile V2.1.1* [online]. TS 103 171, ETSI, Březen 2012, [cit. 27. 2. 2016]. Dostupné z: http://www.etsi.org/deliver/etsi_ts/103100_103199/103171/02.01.01_60/ts_103171v020101p.pdf

[3] Bartel, M.; Boyer, J.; Fox, B.; aj.: *XML Signature Syntax and Processing (Second Edition)* [online]. Technická zpráva, W3C Recommendation, Poslední revize 10. 6. 2008, [cit. 14. 3. 2016]. Dostupné z: https://www.w3.org/TR/xmldsig-core/

[4] WWW Consorcium: *W3C Standards FAQ* [online]. 2012, [cit. 14. 3. 2016]. Dostupné z: https://www.w3.org/standards/faq#std

[5] European Telecommunications Standards Institute: *XML Advanced Electronic Signatures (XAdES) v1.2.1* [online]. TS 102 853, ETSI, Prosinec 2012, [cit. 27. 2. 2016]. Dostupné z: http://www.etsi.org/deliver/etsi_ts/101900_101999/101903/01.04.02_60/ts_101903v010402p.pdf

[6] Berners-Lee, T.; Fielding, R. T.; Masinter, L.: *Uniform Resource Identifier (URI): Generic Syntax* [online]. RFC 3986, RFC Editor, Leden 2005, [cit. 14. 3. 2016]. Dostupné z: http://www.rfc-editor.org/rfc/rfc3986.txt

[7] Evropský parlament a Rada: *Směrnice Evropského parlamentu a Rady 1999/93/ES*. 1999. Dostupné z: http://eur-lex.europa.eu/legal-content/CS/TXT/PDF/?uri=CELEX:31999L0093&from=CS

[8] Peterka, J.: *Báječný svět elektronického podpisu*. CZ. NIC, 2011, ISBN 978-80-904248-3-8. Dostupné z: https://secure.nic.cz/files/bajecny_svet/peterka_bs_cznic.pdf

[9] Farrell, S.; Housley, R.: *An Internet Attribute Certificate Profile for Authorization* [online]. RFC 3281, RFC Editor, Duben 2002, [cit. 23. 3. 2016]. Dostupné z: http://www.rfc-editor.org/rfc/rfc3281.txt

[10] Freed, N.; Borenstein, N.: *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types* [online]. RFC 2046, RFC Editor, Listopad 1996, [cit. 14. 3. 2016]. Dostupné z: http://www.rfc-editor.org/rfc/rfc2046.txt

[11] European Telecommunications Standards Institute: *CMS Advanced Electronic Signatures (CAdES) v2.1.1* [online]. TS 101 733, ETSI, Březen 2012, [cit. 27. 2. 2016]. Dostupné z: http://www.etsi.org/deliver/etsi_ts/101700_101799/101733/02.01.01_60/ts_101733v020101p.pdf

[12] Adams, C.; Cain, P.; Pinkas, D.; aj.: *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)* [online]. RFC 3161, RFC Editor, Srpen 2001, [cit. 23. 3. 2016]. Dostupné z: http://www.rfc-editor.org/rfc/rfc3161.txt

[13] Rivest, R.: *The MD5 Message-Digest Algorithm* [online]. RFC 1321, RFC Editor, Duben 1992, [cit. 14. 3. 2016]. Dostupné z: http://www.rfc-editor.org/rfc/rfc1321.txt

[14] European Telecommunications Standards Institute: *ETSI Drafting Rules* [online]. Leden 2012, [cit. 27. 2. 2016]. Dostupné z: http://www.etsi.org/WebSite/document/Legal/ETSI%20Drafting%20Rules%20January%202012.pdf

[15] European Telecommunications Standards Institute: *PDF Advanced Electronic Signature (PAdES) Profiles; Part 1, v1.1.1* [online]. TS 102 778-1, ETSI, Červenec 2009, [cit. 27. 2. 2016]. Dostupné z: http://www.etsi.org/deliver/etsi_ts/102700_102799/10277801/01.01.01_60/ts_10277801v010101p.pdf

[16] Nadalin, A.; Kaler, C.; Monzillo, R. M.; aj.: *Web Services Security: SOAP Message Security 1.1* [online]. Technická zpráva, OASIS, Poslední revize 1. 2. 2006, [cit. 14. 3. 2016]. Dostupné z: https://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

[17] Box, D.; Christensen, E.; Curbera, F.; aj.: *Web Services Addressing* [online]. Technická zpráva, W3C Member Submission, Poslední revize 10. 8. 2004, [cit. 14. 3. 2016]. Dostupné z: https://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

[18] Ministerio de Industria, Energía y Turismo: *Aplicación eCo-Firma 1.4* [online]. [přístup 10. 3. 2016]. Dostupné z: https://sedeaplicaciones.minetur.gob.es/ecofirma/

[19] Republic of Estonia: *Information System Authority* [online]. [přístup 10. 3. 2016]. Dostupné z: https://www.ria.ee/en/

[20] European Telecommunications Standards Institute: *XML format for signature policies v1.1.1* [online]. TR 102 038, ETSI, Duben 2002, [cit. 27. 2. 2016]. Dostupné z: http://www.etsi.org/deliver/etsi_tr/102000_102099/102038/01.01.01_60/tr_102038v010101p.pdf

[21] DigiDoc Format Specification [online]. Poslední revize 12. 5. 2004, [přístup 10. 3. 2016]. Dostupné z: http://www.id.ee/public/DigiDoc_format_1.3.pdf

[22] Gonçalves, L.: *XAdES4j* [software]. 2014, [přístup 14. 3. 2016]. Dostupné z: https://github.com/luisgoncalves/xades4j

[23] Boyer, J.; Marcy, G.: *Canonical XML Version 1.1* [online]. Technická zpráva, W3C Recommendation, Poslední revize 2. 5. 2008, [cit. 14. 3. 2016]. Dostupné z: https://www.w3.org/TR/xml-c14n11/

[24] Joinup: *Digital Signature Service* [online]. Poslední revize 18. 2. 2016, [přístup 14. 3. 2016]. Dostupné z: https://joinup.ec.europa.eu/software/sd-dss/release/all

[25] Evropská komise: *Rozhodnutí Evropské komise 2011/130/EU*. 2011. Dostupné z: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2011:053:0066:0072:CS:PDF

[26] Evropská komise: *Rozhodnutí Evropské komise 2009/767/ES*. 2009. Dostupné z: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2009:299:0018:0054:CS:PDF

[27] Oracle Corporation: *Java Platform Standard Edition 8 Documentation* [online]. 2016, [cit. 20. 4. 2016]. Dostupné z: http://docs.oracle.com/javase/8/docs/

[28] Catania, N.; Eastlake, D. E.; Geuer-Pollmann, C.; aj.: *Java XML Digital Signature API Specification* [online]. JSR 105, Poslední revize 24. 6. 2005, [cit. 10. 3. 2016]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/security/xmldsig/overview.html

[29] The Apache Software Foundation: *Apache Santuario* [software]. 2016, [přístup 14. 3. 2016]. Dostupné z: http://santuario.apache.org/

[30] Imamura, T.; Dillaway, B.; Simon, E.: *XML Encryption Syntax and Processing* [online]. Technická zpráva, W3C Recommendation, Poslední revize 10. 12. 2002, [cit. 14. 3. 2016]. Dostupné z: https://www.w3.org/TR/xmlenc-core/

[31] Legion of the Bouncy Castle Inc.: *Legion of the Bouncy Castle* [online]. 2013, [cit. 15. 4. 2016]. Dostupné z: https://www.bouncycastle.org/

[32] Myers, M.; Ankney, R.; Malpani, A.; aj.: *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP* [online]. RFC 2560, RFC Editor, Červen 1999, [cit. 14. 3. 2016]. Dostupné z: http://www.rfc-editor.org/rfc/rfc2560.txt

[33] Cooper, D.; Santesson, S.; Farrell, S.; aj.: *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile* [online]. RFC 5280, RFC Editor, Květen 2008, [cit. 23. 3. 2016]. Dostupné z: http://www.rfc-editor.org/rfc/rfc5280.txt

[34] Pivotal Software, Inc.: *Spring Framework* [software]. 2016, [přístup 1. 5. 2016]. Dostupné z: https://projects.spring.io/spring-framework/

[35] Pivotal Software, Inc.: *Spring Web Services* [software]. 2016, [přístup 1. 5. 2016]. Dostupné z: http://projects.spring.io/spring-ws/

[36] The Apache Software Foundation: *Apache Xerces* [software]. 2016, [přístup 14. 3. 2016]. Dostupné z: http://xerces.apache.org/

[37] Boyer, J.; Marcy, G.; Datta, P.; aj.: *XML Normalization* [online]. Technická zpráva, W3C Editor's Draft, Poslední revize 15. 3. 2013, [cit. 20. 4. 2016]. Dostupné z: https://www.w3.org/2008/xmlsec/Drafts/xml-norm/Overview.html

[38] Le Hors, A.; Le Hégaret, P.; Wood, L.; aj.: *Document Object Model (DOM) Level 3 Core Specification* [online]. Technická zpráva, W3C Recommendation, Poslední revize 7. 4. 2004, [cit. 20. 4. 2016]. Dostupné z: https://www.w3.org/TR/DOM-Level-3-Core/

[39] Gudgin, M.; Mendelsohn, N.; Nottingham, M.; aj.: *SOAP Message Transmission Optimization Mechanism* [online]. Technická zpráva, W3C Recommendation, Poslední revize 25. 1. 2005, [cit. 20. 4. 2016]. Dostupné z: https://www.w3.org/TR/soap12-mtom/

[40] European Telecommunications Standards Institute: *e-Signature Validation Remote Plugtests 2016* [online]. 2016, [cit. 28. 4. 2016]. Dostupné z: http://www.etsi.org/news-events/events/1060-e-signature-validation-2016

[41] European Telecommunications Standards Institute: *Procedures for Creation and Validation of AdES Digital Signatures; Part 1, v1.1.0* [online]. EN 319 102-1, ETSI, Únor 2016, [cit. 22. 4. 2016]. Dostupné z: `http://www.etsi.org/deliver/etsi_en/319100_319199/31910201/01.01.00_30/en_31910201v010100v.pdf`

[42] SmartBear Software: *SoapUI 5* [software]. 2015, [přístup 1. 5. 2016]. Dostupné z: `https://www.soapui.org/`

[43] The Eclipse Foundation: *Jetty* [software]. 2016, [přístup 1. 5. 2016]. Dostupné z: `http://www.eclipse.org/jetty/`

Příloha A

Seznam použitých zkratek

API Application programming interface

ASN.1 Abstract syntax notation one

ASiC Associated signature container

CA Certifikační autorita

CAdES CMS advanced electronic signature

CMS Cryptographic message syntax

CRL Certificate revocation list

DSA Digital signature algorithm

ETSI European telecommunications standards institute

GUI Graphical user interface

JAXB Java architecture for XML binding

JCA Java cryptography architecture

JDK Java development kit

JEE Java Enterprise Edition

JRE Java runtime environment

JVM Java virtual machine

JSR Java specification request

MIME Multipurpose internet mail extensions

OCSP Online certificate status protocol

OXM Object/XML mapping

PAdES PDF advanced electronic signature

PKCS Public key cryptographic standards

PDF Portable document format

QP Qualifying properties

RSA Rivest, Shamir, Adleman

SDK Software development kit

SOAP Simple object access protocol

StAX Streaming API for XML

TSA Time stamping authority

TSP Time service provider

URI Uniform resource identifier

W3C World wide web consortium

XAdES XML advanced electronic signature

XML Extensible markup language

XSD XML schema definition

WSDL Web services description language

Příloha B

Instalační příručka

B.1 Potřebné SW vybavení

Pro použití jádra nástroje je potřeba JRE verze 8 nebo vyšší. Pro spuštění webové služby je pak potřeba navíc libovolný servletový kontejner.

Pro zkompilování obou modulů je potřeba mít JDK verze 8 nebo vyšší. K sestavení aplikace a vykonávání přidružených úloh (generování tříd pro webovou službu, obsluha jetty serveru) musí být nainstalován *Maven2*.

B.2 Kompilace

Oba moduly je možné sestavit spuštěním následujícího příkazu z kořenového adresáře projektu, `xades-validator`:

```
mvn package
```

Při prvním spuštění příkazu bude pravděpodobně potřeba mít aktivní internetové připojení pro stažení potřebných závislostí.

Po úspěšném dokončení příkazu budou vytvořeny následující tři Java archivy:

```
./xades-validator-core/target/xades-validator-core-1.0.jar
./xades-validator-core/target/xades-validator-core-1.0-full.jar
./xades-validator-service/target/xades-validator-service-1.0.war
```

První z nich obsahuje zkompilované jádro nástroje bez závislostí a pro použití vyžaduje jejich dodání, druhý pak jádro společně se všemi závislostmi a je samostatně použitelný jako knihovna. Třetí z nich je webový archiv obsahující webovou službu, který lze nasadit do libovolného servletového kontejneru.

B.3 Spuštění webové služby

Nejjednodušší cestou pro spuštění webové služby je využití *Maven* pluginu. Z adresáře `xades-validator-service` je po provedení kompilace potřeba spustit následující příkaz:

```
mvn jetty:run
```

Tento příkaz spustí jetty server s aplikací na portu 8080. Pro použití jiného portu je možné spustit příkaz s parametrem:

```
mvn -Djetty.http.port=<číslo portu> jetty:run
```

Při použití původního portu je webová služba dostupná na této adrese:

```
http://localhost:8080/xades-validator-service/validatorService/
```

WSDL je pak publikováno na následující adrese:

```
http://localhost:8080/xades-validator-service/validatorService.wsdl
```

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `bin` ..... adresář se zkompilovanými výstupy implementace
- `src`
  - `xades-validator` ..... zdrojové kódy implementace
    - `xades-validator-core` ..... zdrojové kódy jádra nástroje
    - `xades-validator-service` ..... zdrojové kódy webové služby
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `DP_Švorc_Martin_2016.pdf` ..... text práce ve formátu PDF