ASN1spect: Uncovering ASN.1 Compiler-Generated Vulnerabilities in Critical Infrastructure

Seaver Thorn
North Carolina State University
Raleigh, USA
swthorn@ncsu.edu

Nathaniel Bennett
University of Florida
Gainesville, USA
bennett.n@ufl.edu

Kevin Butler
University of Florida
Gainesville, USA
butler@ufl.edu

Patrick Traynor
University of Florida
Gainesville, USA
traynor@ufl.edu

William Enck
North Carolina State University
Raleigh, USA
whenck@ncsu.edu

Abstract

ASN.1 is widely used for communication protocols in critical infrastructure. Many projects avoid parsing vulnerabilities by using ASN.1 compilers to automatically generate parsing code directly from complex protocol specifications. However, ASN.1 compilers can themselves have vulnerabilities, propagating vulnerable parsing routines to projects that use them. This paper proposes ASN1spect, a novel approach to identifying known vulnerable code generated by vulnerable ASN.1 compilers. Our analysis of vulnerabilities related to ASN.1 type constraints show that known and silently fixed vulnerabilities propagate to actively-maintained downstream projects and remain undetected for years. While our primary focus is on silently fixed supply-chain vulnerabilities, our analysis also uncovered two previously unknown vulnerabilities in asn1c. We apply ASN1spect to 93 open-source projects containing asn1c-generated parsing code and detect vulnerable parsing code in 40 (43%). We further demonstrate proof-of-concept payloads that can cause denial-of-service and logic vulnerabilities in electrical grid and satellite communication projects. These results motivate managing code generators such as ASN.1 compilers as versioned components in the software supply chain.

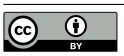
CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

ASN.1 code generation, vulnerability discovery, software supply chain

ACM Reference Format:
Seaver Thorn, Nathaniel Bennett, Kevin Butler, Patrick Traynor, and William Enck. 2026. ASN1spect: Uncovering ASN.1 Compiler-Generated Vulnerabilities in Critical Infrastructure. In *Proceedings of the 2026 ACM Secure Development Conference (SecDev '26), July 5–6, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3805773.3805995



SecDev '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2602-6/2026/07
https://doi.org/10.1145/3805773.3805995

1 Introduction

Abstract Syntax Notation One (ASN.1) is used throughout wireless networks and critical infrastructure. Initially developed for telecommunications [48], ASN.1 is now widely used for X.509 certificates [49], network management [46], electric and gas utilities [22], and airplane and satellite communication [20, 59]. Packet parsers are a natural target for attackers [16, 43, 56, 64], and ASN.1 is no exception. Fortunately, many projects use ASN.1 compilers to generate parsing code directly from complex protocol specifications, minimizing human error in parsing routines.

Unfortunately, ASN.1 compilers can themselves have vulnerabilities, propagating vulnerable parsing routines to projects that use them. Finding uses of vulnerable ASN.1 compilers is non-trivial. In contrast to traditional vulnerable components in software supply chain security [11], ASN.1 compilers are not managed via package registries, nor do they leave easily identifiable code clones [23, 24] in projects. The generated code is unique to each ASN.1 protocol specification, and it is often committed directly to the project's software repository. We identified multiple active projects where the ASN.1 parsing code was not modified for over a decade [18, 30].

The goals of this paper are two-fold. First, we seek to understand vulnerabilities in ASN.1 compilers, in particular those that have been silently fixed [10, 65, 66], i.e., without providing any public notification. Second, we seek to scale the discovery of vulnerable ASN.1 parsing code across open-source projects and facilitate vulnerability management. We target the most popular and feature-complete open-source ASN.1 compiler: asn1c [63]. There are several widely used forks of asn1c, and the forks often fix vulnerabilities in code generation without clear releases.

In this paper, we focus on *ASN.1 type constraints*. Projects rely on ASN.1 parsing code to properly validate input, and failure to do so can result in remote code execution, memory corruption, and denial of service [1]. Specifically, we propose the use of *differential constraint analysis* to discover constraint-related vulnerability fixes in ASN.1 compilers. We present the ASN1spect static analysis tool that semi-automatically analyzes a project P to determine if it used a vulnerable version of asn1c. With a small amount of manual effort to retrieve the existing ASN.1 protocol specification, ASN1spect automatically (1) identifies the ASN.1 parsing code, (2) creates a variant P', replacing ASN.1 parsing code with code generated by an up-to-date version of asn1c, and (3) symbolically extracts and compares the constraints for all ASN.1 types between P and P'.

ASN1SPECT also includes several other analysis modules inspired by our exploration of ASN.1 compiler vulnerabilities. These modules include (a) a differential analysis between the decoding functions for different ASN.1 encoding types, (b) the use of recursive ASN.1 types, which can lead to stack overflows if not managed properly, and (c) inconsistent data structures for managing information object sets, which was motivated by a recently discovered vulnerability [1].

We have two broad types of findings. First, we identify four asn1c vulnerabilities, two were silently fixed, and two of which are new. The first silently fixed vulnerability was discovered by comparing constraints between P and P'. Specifically, (1) we identified an old fork / version that did not enforce constraints. The second silently fixed vulnerability relates to a recently discovered NextEPC vulnerability [1]. Notably, (2) we identified an inconsistency with the data structures used to manage information object sets. The differential constraint analysis between the decoding functions of different ASN.1 encoding types found inconsistencies in the latest forks / versions. In particular, (3) we identified that constraints for `ENUMERATED` types are not enforced for BER and DER ASN.1 encoding types. Finally, (4) we identified that recursive ASN.1 types can cause stack overflows if limits are not defined, which we found when some ASN.1 protocol specifications caused ASN1SPECT to crash. We performed responsible disclosure of all four asn1c vulnerabilities, receiving four CVEs from MITRE to help project maintainers understand risks.

Second, we used ASN1SPECT to analyze 93 open-source projects containing asn1c-generated parsing code. Simply using a vulnerable version of asn1c is not enough; whether the issue manifests depends on how the project uses the affected features in the generated code, not just which features are present or which compiler version was used. ASN1SPECT identified vulnerable parsing code in 40 projects, affecting 704 types and 19 ASN.1 specifications. The vulnerabilities allow attackers to (a) perform denial of service against critical technologies and (b) send data that can poison applications that rely on ASN.1 validation. We further created proof-of-concept exploits for two projects. The first allows an attacker to send a maliciously crafted packet that could take down power monitoring utilities, impacting an operator's ability to monitor the substation or individual components. The second proof-of-concept allows an attacker to cause a program to accept invalid data as though it is valid for airplane communication. This invalid data can lead to logic errors or fool air traffic control operators on important data fields of an airplane, such as its location, speed, or altitude.

Contributions: Our approach does not require knowing which version of asn1c was used to generate the code. We provide an implementation of our approach ASN1SPECT, which targets the most popular open-source ASN.1 compiler (asn1c). We identify four previously undisclosed vulnerabilities in asn1c by integrating different versions of the compiler, and we capture these findings as analysis modules within ASN1SPECT. We apply ASN1SPECT to 93 open-source projects, discovering that 40 (43%) have vulnerable parsing code. We will release the source code of ASN1SPECT and data upon acceptance of this paper. The source code for ASN1SPECT is available at https://github.com/wspr-ncsu/ASN1spect/.

2 Background

ASN.1 is an interface description language for defining data structures in a programming language-agnostic way. Initially standardized in 1984 as ITU X.208, ASN.1 underwent significant revisions in the 1990s, resulting in ITU X.680 [47, 50]. We refer to the ITU X.680 series of specifications as the *ASN.1 standard* and individual ASN.1 protocols as an *ASN.1 specification*.

An ASN.1 compiler translates specifications into code that parses the data structures in a programming language. As such, an ASN.1 compiler acts as a specialized parser-generator for ASN.1 specifications, where implementations are rarely formally verified. Implementations that are formally verified only verify a subset of ASN.1 features [33, 40]. Unlike competing formats such as Protobuf [17], ASN.1 offers a diverse set of encoding types (BER, DER, PER, ...), offering flexibility where each encoding type is optimized for different use cases and has unique parsing requirements. However, capturing this complexity requires a context-sensitive grammar, especially for arbitrary ASN.1 specifications. As such, it is difficult to use a standard parser-generator like ANTLR [37].

2.1 ASN.1 Specification Example

Figure 1 is an example of using ASN.1 to define data structures and the code generated by asn1c. This example uses data types defined by multiple ASN.1 specifications including aviation, LDAP, and cellular networks. Each new type starts with the name of the type, followed by an ASN.1 data type such as `INTEGER`, `ENUMERATED`, `CHOICE`, or `SEQUENCE`. An ASN.1 specification consists of new types, fields, and constraints. The constraints validate incoming data and can effectively prevent attacks such as buffer overflows and denial of service by ensuring only syntactically and semantically valid data is processed by the application.

Integer Constraints: In Figure 1, PlaneAltitude is a new `INTEGER` type that accepts values between 0 and 40,000 inclusive. The compiler generates code for each new type, including a way to verify that a given value satisfies the type's constraints. When encoding or decoding this type, the generated code validates the decoded value using the constraints defined in the specification after parsing.

Enum Constraints: An `ENUMERATED` type defines a named set of values. Each named item is represented by a keyword and a numeric index. Unlike `INTEGER` types, which can hold any numeric value within constraints, `ENUMERATED` types are restricted to only the listed named items. As such, `ENUMERATED` types can be conceptualized as a mapping between integers and keywords.

Recursive Type Decoding: A recursive type is a structured data type that includes itself. Lines 31 to 36 in Figure 1 show the recursive type `Filter` that references itself. ASN.1 allows recursive types, but the application must take precautions to prevent decoding a deeply nested value from causing a stack overflow [50].

Information Object Set: Information Object Classes (IOC) and Sets (IOS) are conceptually analogous to classes and objects in Object-Oriented programming. The IOC defines a template where the information object (IO) provides values for the fields in the class. An IOS is a collection of information objects that conform to a single IOC and are implemented as a series of data structures.

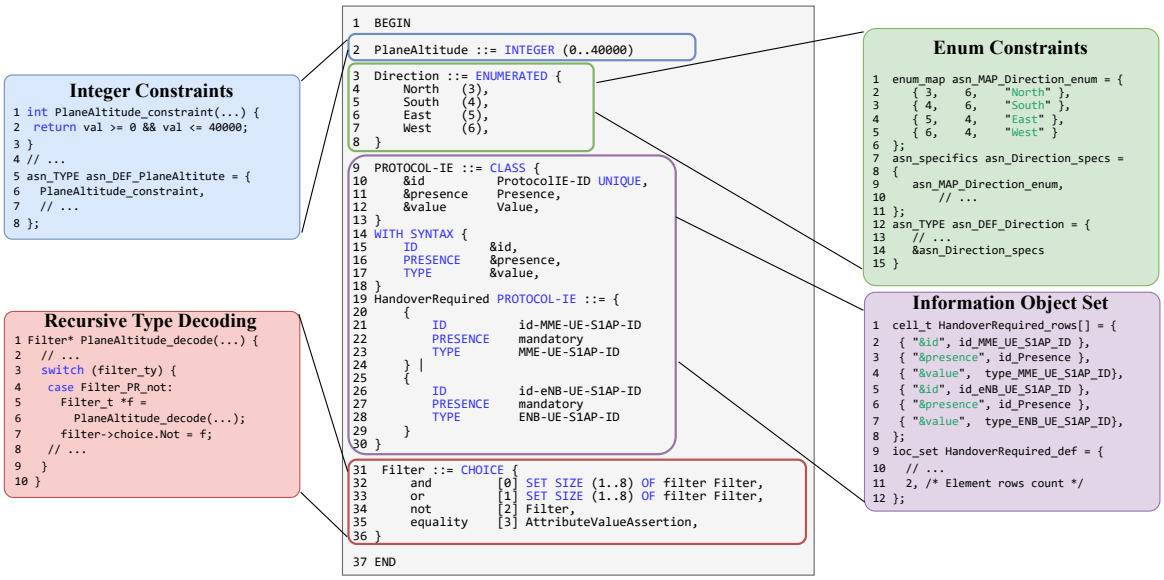


Figure 1: A fictitious ASN.1 specification with real definitions from multiple ASN.1 specifications (airplane, LDAP, and cellular network). On both sides of the ASN.1 definition, there are corresponding generated code snippets. Code snippets are significantly simplified for brevity; generated code involves several layers of indirection spanning multiple files.

ASN.1 Encoding Types: ASN.1 supports multiple methods to encode and decode messages. The different methods allow the protocol designer to prioritize compactness, speed, human-readability, or another property. ASN.1 Basic Encoding Rules (BER) and Distinguished Encoding Rules (DER) encode elements as Tag-Length-Value sequences. In other encoding types, such as Packed Encoding Rules (PER) and Octet Encoding Rules (OER), the ASN.1 specification is assumed to be known by both sides of the communication and therefore the tag is ommitted. These encoding types are used by different specifications and have different parsing requirements. For example, X.509 digital certificates uses DER ASN.1 encoding, which due to DER's deterministic nature avoids ambuiguities in certificate parsing. PER is common in bandwidth-constrained protocols, such as telecom standards (S1AP, NGAP) and aviation/satellite systems (CPDLC). Finally, Lightweight Directory Access Protocol (LDAP) and Manufacturing Message Specification (MMS) use BER encoding for historical reasons.

2.2 ASN.1 Compiler Study

We selected asn1c based on a detailed investigation of ASN.1 parsing code on GitHub. We focused on open-source ASN.1 compilers because closed-source compilers have strict licensing agreements and high costs [21, 34]. Moreover, many developers and companies rely on open-source compilers. The remainder of this subsection describes why we chose asn1c as our target compiler.

Compilers: We considered three open-source ASN.1 compilers: asn1c [63], asn1scc [13], and eSNACC-ng [12]. asn1c was created in 2003 by Lev Walkin to provide a generic ASN.1 parser for many encoding types and was last committed to in 2021 [63]. Many forks of asn1c add new features and continue the development with the latest commits in 2025. asn1scc was designed by the European Space Agency in 2008 and is used to support customized protocols in satellite systems, flight, and ground software [13, 26]. Finally, eSNACC-ng is a fork of the snacc ASN.1 compiler and provides a generalized compiler that can generate code in C and C++ [12].

Dataset: To identify repositories containing ASN.1 parsers, we used GitHub's API. We identified C code snippets in the generated code for the three compilers to use as the search criteria. These code snippets consistently appear in the output of the compiler, regardless of the specification. For example, "`define ASN1SCC_ASN1CRT_H_`" is unique to the asn1scc compiler's output. From the search, we clone and collect every repository. Forked repositories were excluded by filtering them out during the search. We found that forked repositories tend not to include any newly generated code or new features from the compiler.

Open source use of ASN.1 compilers: We identified 336, 52, and 30 projects on GitHub using asn1c, eSNACC-ng, and asn1scc. asn1c is the most popular C language ASN.1 compiler by almost an order of magnitude. Projects that use asn1c also have significantly more user engagement, with 17,067 GitHub stars collectively compared to 602 and 627 for asn1scc and eSNACC-ng, respectively. While stars for individual repositories can be manipulated [19], it is unlikely there is a coordinated effort for projects in favor of asn1c compared to the other ASN.1 compilers.

Industry use of asn1c: asn1c is also used by a wide range of companies. asn1c's BSD license requires proprietary software to include an acknowledgment. We found licenses from prominent technology companies using asn1c including Apple [60], Bosch [2],

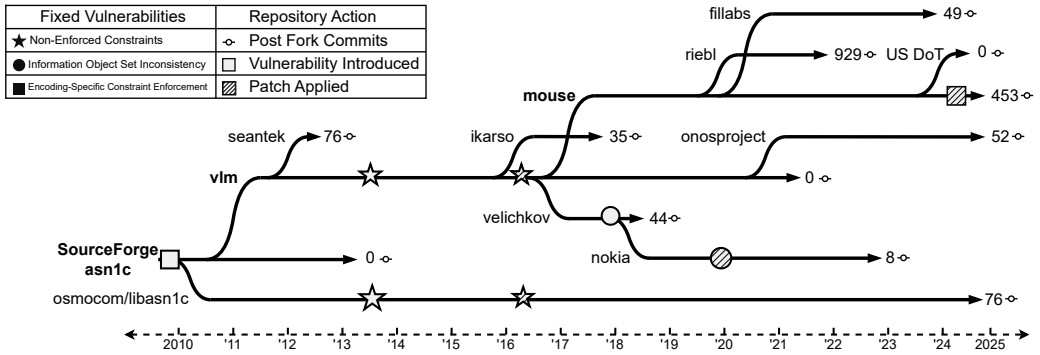


Figure 2: Maintenance timeline of asn1c and its prominent forks. Bolded titles (i.e., SourceForge, vlm and mouse) denote primary sources of active development/release for asn1c. Other prominent forks and their silent vulnerability fixes are shown alongside. The patch for the Encoding-Specific Constraint Enforcement is a direct result of our work.

Cisco [8], Juniper [31], Mercedes-Benz [62], NASA [44], Oracle [36], Samsung [7], Siemens [54], Telecom Behnke [55], and Viasat [58].

asn1c version complexity: Figure 2 shows the fragmented use of asn1c versions. The upstream repository for asn1c is outdated, and as a result, many forks are used in practice. Different asn1c releases produce nearly identical output–including version numbers and documentation links–making it challenging to determine which version a repository relies on [15]. Our analysis indicates many projects unknowingly rely on either older insecure versions of asn1c or forked asn1c compilers that do not include all upstream patches. Additionally, of the major Linux distributions that distribute asn1c, they provide only the base repository version (vlm) [39, 52, 63].

3 Overview

Projects often use ASN.1 compilers to create parsing code directly from ASN.1 specifications. However, ASN.1 compilers can themselves have vulnerabilities, which produce vulnerable code. Identifying ASN.1 type constraint vulnerabilities in ASN.1 parsing code generated by compilers presents the following research challenges:

- **C1** *The ASN.1 standard is complex.* Manually verifying the semantic correctness of generated code against a complex ASN.1 standard and specification is time-consuming and error-prone. Automating this comparison would require re-implementing the logic already in ASN.1 compilers, which itself would require verification. Retrofitting formal verification is impractical; asn1c lacks the design foundations, and proving its non-compliance provides little utility since maintainers are unlikely to re-architect the system.
- **C2** *Application developers often do not reference the compiler version used.* While symbols can identify the broad family of compiler used (e.g., asn1c), the specific version of that compiler is difficult to determine.[1] Hence, traditional vulnerability management approaches that track version numbers cannot be used to identify vulnerable programs.
- **C3** *Using a vulnerable version of a compiler does not mean the generated code is vulnerable.* A vulnerability in the compiler's logic, constraint enforcement, or its code generation routines might only be triggered under circumstances that are determined by the ASN.1 specification.

[1]Ironically, all versions of asn1c from the past 8 years insert the same version code: "v0.9.29" [15].

We overcome these challenges by developing ASN1SPECT, a tool that uses differential constraint analysis, a methodology that employs two distinct forms to uncover ASN.1 vulnerabilities. First, we perform differential constraint analysis against program variants. ASN1SPECT compares the constraints in program P and a variant program P' created using the same specification but a different compiler version. Second, ASN1SPECT performs differential constraint analysis on a single type across all supported encoding types. For each ASN.1 type, ASN1SPECT compares the constraints extracted from parsers for encoding types when they differ, thereby detecting encoding-specific constraint enforcement issues.

ASN1SPECT's differential constraint analysis approach addresses the three challenges in different ways. Rather than re-implementing ASN.1 semantics (**C1**), ASN1SPECT leverages the diversity of ASN.1 compiler versions and encoding types. Recall from Figure 2 that asn1c has different forks, each with different versions. If a program P has a type constraint parsing vulnerability that was fixed by the latest version of one of the forks, differential analysis will detect it. This approach also has the benefit of being agnostic to specific version numbers (**C2**). It does not matter which version of the compiler was used to create P, as long as the compiler used to create P' has fixed the vulnerability. Finally, differential analysis avoids falsely reporting a problem when the vulnerable feature does not exist (**C3**). If the vulnerable feature is not in the code, the constraints for P and P' will be equivalent.

Differential Constraint Analysis: Figure 3 shows ASN1SPECT's differential analysis process. Initially, ASN1SPECT takes a project P that has an ASN.1 parser (①) in the code base and the ASN.1 specification (②). ASN1SPECT then regenerates the ASN.1 parser (③) using the latest version (or any chosen version) of asn1c with the ASN.1 specification and replaces the parser in P to create P' (④). ASN1SPECT additionally analyzes all supported encoding types within a single P to ensure that the parser handles constraints on ASN.1 types consistently regardless of the encoding type used.

Conceptually, ASN1SPECT's differential constraint analysis modules operate in two phases: (1) *constraint extraction* and (2) *constraint*

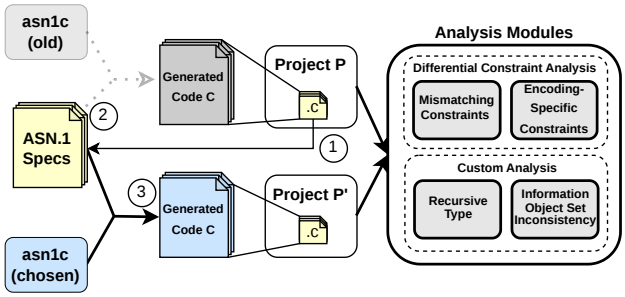


Figure 3: Projects include asn1c's generated code (1). We locate which ASN.1 specification (2) was used in P to regenerate the code (3) for P'. P and P' are input to all Analysis Modules.

comparison. ASN1SPECT identifies the data structure symbols that represent ASN.1 types. Each type is then modeled as an independent data structure to abstract compiler-specific details. asn1c implements a type's constraint enforcement as a new function for each type. These constraint enforcement functions are simple and only check the constraints for a single type. ASN1SPECT uses both static analysis to extract function pointers and symbolic execution of the function to extract the type constraints from the generated code. ASN1SPECT uses angr [53] to extract the ASN.1 constraints.

Additional Analysis Modules: OpenSSL and Wireshark recently patched a vulnerability caused by deeply nested recursive ASN.1 types [14, 28]. Similarly, we discovered that asn1c may also cause a stack overflow when presented with a deeply nested structure. We also generalized the IOS vulnerability recently identified in a cellular core implementation [1]. Information Object Sets are represented by multiple data structures across the code base, but the structures may do so with conflicting elements. We describe these additional analysis modules in Appendix A.

4 ASN1SPECT

ASN1SPECT is primarily designed to identify ASN.1 type constraint problems. Two of ASN1SPECT's analysis modules use differential constraint analysis on program variants and encoding types. ASN1SPECT also includes two additional analysis modules that demonstrate the utility of its abstractions.

4.1 Differential Constraint Analysis

An ASN.1 parser is responsible for enforcing constraints on types, as discussed in Section 2. This subsection begins by defining primitives used by the constraint analysis. We then discuss the analysis algorithm itself. We conclude by discussing module-specific topics.

4.1.1 Definitions. ASN.1 type definitions can define the type based on collections of elements or a primitive type such as `INTEGER` and `UTF8String`. The collections have semantics (e.g., `CHOICE` or `SEQUENCE`), modeling them is not necessary. It is sufficient for ASN1SPECT to extract the constraints and capture all elements within the type.

Definition 1 (ASN.1 Types)**.** ASN.1 recursively defines types using a context-free grammar. Let t be a single ASN.1 type. We model each type t as a tuple (T, c) consisting of a set of types T and a constraint c (Def. 2) that applies to t. The set of types T is collected from the elements defined in the specification (e.g., in the `SET`). For many types, T is $\emptyset$. We refer to the set of all types as $\mathcal{T}$.

Conceptually, Constraints are subsets or filters on a type. ITU X.680 [50] states that a constraint is "A notation which can be used in association with a type, to define a subtype of that type." [50, Def. 3.8.16] Hence, (1) a constraint evaluates to a subset of possible values for a type, and (2) the definition of a constraint directly relates to the corresponding type.

For example, if constraint c corresponds to an `INTEGER` primitive type, the constraint may consist of ranges (e.g., `1..10`) and specific values (e.g., `20`). Multiple constraints can be combined with a `|` operator that unions the constraints (e.g., `1..10 | 20`). Constraints can also be combined with an `^` operator for intersection.

Not all types are based on numbers. For example, `UTF8String` is based on alphanumeric characters. ASN.1 specifications commonly use regular expressions to define constraints on strings. They also commonly restrict the length of strings using the `SIZE` operator. OSS Nokalva maintains a comprehensive list of ASN.1 constraint types as well as examples for each constraint [35].

As with ASN.1 types, ASN1SPECT does not need to model the exact set of values defined by a constraint. Its analysis algorithm simply compares the sets of constraints between the two programs to determine if they are equivalent (see Def. 4).

Definition 2 (Constraints)**.** Let c be a constraint for a type t. Without c, type t has an associated value set (e.g., all integers). The constraint c limits which values in the value set can be assigned to t. We recursively model a constraint c as a set of constraints C.

A constraint can also reference another type, creating a parent-subtype relationship. ITU X.680 [50] states that (1) a parent type is "the type that is being constrained when defining a subtype, and which governs the subtype notation," [50, Def. 3.8.58] and (2) a subtype is "a type whose values are a subset (or the complete set) of the values of some other type (the parent type)" [50, Def. 3.8.76].

Definition 3 (Parent and Subtypes)**.** Let $t_p = (T, c_p)$ be a ASN.1 type. Let t_s be a subtype defined with respect to t_p with a constraint c_s Then the effective constraint on t_s is $c_p \wedge c_s$. However, as in Def. 2, we only need to model the effective constraint as the set $\{c_p, c_s\}$.

Finally, ASN1SPECT's constraint analysis compares the constraint encoded in the parsers of two program variants. Let P represent the existing project and P' represent the project using ASN.1 parsing code generated by another version of asn1c. For each type t in the corresponding ASN.1 specification, our analysis compares the constraints c and c' we extract from P and P', respectively. Therefore, a key goal is to determine if c and c' are *equivalent*.

Definition 4 (Equivalent Constraints $\equiv_c$)**.** Let c and c' be two constraints. The individual equivalence comparisons are done based on their corresponding types. For example, an `INTEGER` range `1..10` is equivalent to an `INTEGER` range `1..10`. If c and c' are sets of constraints C and C', then all constraints must match. Formally, $c \equiv_c c'$ if and only if $\forall c_i \in C, \exists c'_j \in C'$ such that $c_i \equiv c'_j$ and $\forall c'_j \in C', \exists c'_i \in C$ such that $c'_j \equiv c_i$.

Algorithm 1: AnalyzeConstraints

```
Input: Projects P, P'
Output: Set Γ of types with different constraints
Procedure AnalyzeConstraints(P, P')
    Γ ← ∅;
    T ← ExtractTypes(P);
    T' ← ExtractTypes(P');
    for t_i ∈ T do
        for t_j ∈ T' do
            if EquivTypeNames(t_i, t_j) then
                if t_i.c ≠_c t_j.c then
                    Γ.append((t_i, t_j));
    return Γ;
Procedure EquivTypeNames(t, t')
    if t.name = t'.name then
        for t_i ∈ t.T do
            if ContainsTypeName(t'.T, t_i) = false then
                return false;
        for t_i ∈ t'.T do
            if ContainsTypeName(t.T, t_i) = false then
                return false;
        return true;
    return false;
Procedure ContainsTypeName(T, t)
    for t_i ∈ T do
        if t.name = t_i.name then
            return true;
    return false;
```

4.1.2 Analysis. Algorithm 1 shows how ASN1SPECT compares constraints of types in different programs P and P'. A key challenge is mapping each type t in P to its corresponding type t' in P'. The *AnalyzeConstraints* algorithm begins by calling $ExtractTypes(\cdot)$ on P and P'. The $ExtractTypes(\cdot)$ function is not shown. At a high level, it loops through the symbols in the binary and checks if the name matches a regular expression that captures how asn1c generates type definitions (i.e., "`asn_DEF_`"). The corresponding memory address points to a data structure containing the type definition. These structures vary based on the version of asn1c.

Next, ASN1SPECT identifies matching types based on their names and the elements defined in the data structure (i.e., T from Def. 1). ASN1SPECT goes through all symbols using angr in both program binaries until the symbol names match between two candidate types. Then, it checks if all elements in those types match. If there are elements that do not match, it discards that candidate type and continues looking for a matching type. If no matching type is found, an exception is thrown. In all such cases, it was the result of selecting the wrong version of the ASN.1 specification for generating P'. Using the correct version allowed the analysis to proceed.

When the types match, ASN1SPECT compares their constraints. In the case where two matching types have differing constraints, ASN1SPECT adds the two types to the output set for manual review. Note that ASN1SPECT does not recursively check the constraints of the type's elements. This is not needed, because each element is a type and the analysis covers all types in the program.

4.1.3 Analysis Module Specifics. Two analysis modules rely on the differential constraint analysis. While the constraint enforcement modules use Algorithm 1, each module identifies different problems with the code. The design of each module helps ASN1SPECT detect when constraints between P and P' are mismatched or are enforced only for a specific encoding type.

Mismatching Constraints Module: The mismatching constraints module uses the types extracted from Algorithm 1 and flags types that do not have matching constraints between P and P'. It identifies when constraints are implemented in one parser but not the other by comparing the path constraints between the two functions.

Encoding-Specific Constraint Enforcement Module: Each decoding rule (e.g., BER, UPER) uses its own decoder function for each ASN.1 type (e.g., `INTEGER_decode_ber`, `INTEGER_decode_uper`). The analysis module detects when these functions are reused between distinct types. For example, if BER uses `INTEGER_decode_ber` for an `ENUMERATED` type and UPER uses `ENUMERATED_decode_uper` for an `ENUMERATED` type, then there is a potential decoding vulnerability. Note that since some types have semantically equivalent decodings (e.g., `BIT STRING` and `OCTET STRING`), ASN1SPECT groups them to avoid unnecessary alerts.

4.2 Additional Analysis Modules

The abstractions created by ASN1SPECT allow the creation of custom analysis modules that do not necessarily require differential constraint analysis. We created two additional analysis modules to demonstrate this utility. We provide the complete algorithmic details and implementation specifics in Appendix A.

Recursive Type Analysis Module: Recursive Type Analysis can be performed on a single program and does not require differential analysis. While ASN.1 allows recursive data types, the application developer needs to ensure each ASN.1 parsing method does not receive an untrusted user input directly. Not all instances of a recursive type are vulnerable. For example, if the developer limits the stack depth during parsing, then it is not vulnerable. ASN1SPECT only reports if a type is recursively defined.

Information Object Set (IOS) Analysis Module: IOSes are generated as arrays of structures in the code. They are represented by multiple data structures spread across the code base. The structures may have a conflicting number of elements. During the parsing of an information object, if it contains a field that exists in one of the data structures but not another, it leads to a program crash. This problem was identified by prior work using a fuzzer [1]. Our methodology can detect this problem statically.

4.3 Implementation

We implemented ASN1SPECT in 2900 lines of Python code, which includes all modules (including the two modules in the Appendix). The implementation of the differential constraint analysis modules is 672 lines of code. With all the underlying code, individual analysis modules can be added to ASN1SPECT with minimal effort. The implementation of the recursive type analysis module (appendix) is 39 lines of code. The implementation of the information object set inconsistency module (appendix) is 138 lines of code.

ASN1SPECT uses angr [53] to extract the types from the programs. The types are defined in a data structure and each begins with a prefix "`asn_DEF_`". asn1c has always used this prefix, but the underlying data structures have changed over time. ASN1SPECT supports

different versions of asn1c by analyzing all different data structure variations that asn1c can or has used in the past by determining the structure that exists in the compiled binary. ASN1SPECT identifies ASN.1 type definitions from the symbol table, which contains a pointer to the function that enforces ASN.1 constraints. Symbolic execution is used to extract path constraints from the constraint enforcement function referenced by the type's data structure. The symbolic variable is the value that is checked against the constraint, the return value of the function indicates if the input value satisfies the constraint. These functions are typically small, and there is minimal path explosion using symbolic execution.

Once the constraints are extracted, the value-set describing the constraint is compared. We used angr's z3-based `solver.simplify` function to normalize the extracted constraints. For example, the `INTEGER` constraints 1..5 and 1 | 2 .. 5 both are simplified to 'x >= 1, x <= 5'. Since `solver.simplify` is used in both cases. Characters are handled similarly. We did not encounter any regular expression constraints that were more complex than character ranges.

Analysis Modules: Each analysis is implemented as a separate Python class, which receives the type structure as input. Analysis modules are separated into differential and non-differential analysis. Each analysis module is independent of another, which ensures that they can be added or removed without affecting other modules. Analysis modules implement an `analyze` function that takes a type and returns the result of the analysis as a JSON-serializable type.

To perform differential analysis on program variants, ASN1SPECT loads P types and then checks that P' is also loaded. When both programs are loaded, ASN1SPECT runs each differential analysis module sequentially. ASN1SPECT runs each non-differential analysis module of each type after extracting all types from the binary. Finally, when all analysis modules have finished running, results are saved into separate JSON files for each analysis module.

5 Results

ASN1SPECT aided the identification of four previously undisclosed vulnerabilities, two of which were silently fixed and two of which are new. We use ASN1SPECT to analyze 93 repositories that contain asn1c-generated code and detect vulnerable code in 40 (43%).

5.1 Experimental Setup

Dataset: Using GitHub's API, we identified 336 repositories containing asn1c-generated code written in C by searching for them in Janurary 2025. We consider a repository to be actively maintained if it is at least two years old and has had at least 10 commits in each of the past two years, as established by Miller et al. [27]. This filtered our dataset to 107 repositories. None of the repositories in our final dataset appeared to be test repositories or toy examples; rather, they showed signs of ongoing maintenance and real-world use (e.g., regular commits, issue activity, and integration into larger systems). Of these 107 repositories, we successfully analyzed 93; we discuss why the remaining could not be analyzed in Section 7.2.

Table 1 overviews these 107 projects, grouped by area and listing the total number of stars per area. We also report asn1c code age which is the average time between the latest commit in the repository and the last time asn1c-generated code was updated. We attempted to automatically compile each project. Build errors were resolved by iteratively refining the build process, such as installing missing dependencies or adding compiler flags. We automatically compiled all identified ASN.1 specifications with the latest version of asn1c to create P'. Note that any version of asn1c can be used for this; however, we used the most recent version by mouse07410 as of Janurary 2025 to benefit from the most up-to-date patches [29]. Since asn1c-generated code includes all relevant files from a folder, replacing the parser involves locating the folder and substituting it.

Table 1: Overview of analyzed projects using asn1c.

Area	Active Projects	Stars	Avg asn1c Code Age*
Mobile Networks	47	3,694	483 days
Transportation	16	491	495 days
Electric Grid	7	1,491	1,262 days
GPS/GNSS	6	2,099	519 days
Financial	5	125	810 days
Security & Authentication Protocols	4	93	837 days
Aviation (Airplanes, Satellites)	3	2,230	1,563 days
RFID & NFC	2	541	361 days
LDAP	1	1,180	3,722 days
Apple App Purchase verification	2	1,384	2,375 days
Total	93	13,328	692.5 days

* The average time between the last update to the code generated by asn1c and the most recent commit.

We ran ASN1SPECT on all 93 projects. The experiments were conducted on a virtual machine with 4 vCPUs and 64 GB of RAM, on a server with 2 AMD EPYC 9124 CPUs and 500 GB of RAM. Cloning repositories and compiling projects took 2.5 hours, which was primarily constrained by GitHub rate limits.

5.2 asn1c Vulnerabilities

A combination of developing and running ASN1SPECT led to the identification of four unique vulnerabilities in asn1c.

(1) *Non-Enforced Constraints:* In some versions of asn1c, the constraint function pointer is overwritten by the parent type's constraint function under all circumstances. It was introduced in 2013 [61] and silently fixed in 2017 [3].
(2) *Encoding Specific Constraint Enforcement:* We found that XER, BER and DER encoding types do not enforce constraints on `ENUMERATED` types. asn1c treats `ENUMERATED` values as `INTEGER` and incorrectly accepts any `INTEGER` value instead of restricting the inputs to the named enum values.
(3) *Recursive Types:* Recursively-defined types may cause a stack overflow in asn1c's parsing routines if the application using the generated code does not specifically account for it.
(4) *Information Object Set Inconsistency:* ASN1SPECT generalizes a recently discovered vulnerability in NextEPC [1]. We identified the root cause as an inconsistency in related data structures, which originated in the velichkov fork [57].

5.3 ASN.1 Project Study Results

Non-Enforced Constraints: As shown in Table 2, ASN1SPECT identified 454 automatically generated types across 12 projects. In Section 6.1 we discuss how this can be exploited in the SatDump [45] project, which relies on the vulnerable library libacars [25].

Table 2: ASN1SPECT's findings for each analysis module on our dataset.

Analysis Module	True Positives			False Positives		Timeouts	
	Projects	ASN.1 Types	Specifications	Projects	ASN.1 Types	Projects	ASN.1 Types
Non-Enforced Constraints	12	454	7	0	0	4	28
Encoding Specific Constraint Enforcement	10	225	5	33 *	943 *	0	0
Recursive Types	27	12	6	0	0	0	0
Information Object Set Inconsistency	8	13	2	0	0	0	0
Total Unique	40	704	19	33	943	4	28

* These projects and alerts were classified as false positives because the project does not use the affected encoding types. However, if it used the affected encoding types, it would be vulnerable.

ASN1SPECT had 28 types that timed out during analysis due to path explosion using symbolic execution. For example, when the constraint function performed a loop on a field of a symbolic variable, the path constraints could not be reliably extracted. When this occurred, ASN1SPECT marked each type for manual review and we list them as timeouts in Table 2. We manually reviewed them and found that they were not vulnerable.

Encoding-Specific Constraint Enforcement: As shown in Table 2, ASN1SPECT identified 43 projects with a problem in the XER's, BER's, and DER's constraint enforcement function for `ENUMERATED` types. All of these were for `ENUMERATED` types. Upon reviewing, we found that 10 are vulnerable since they use XER, BER, or DER. The other 33 contain vulnerable code, but that code will never be executed since they use an encoding type that correctly enforces the constraints. However, we note that if the projects were to use XER, BER, or DER then they would be vulnerable. While ASN1SPECT could be modified to include reachability analysis, imprecision in call graph construction will lead to false negatives. Given the limited manual effort required to determine what encoding type a project uses, we determined to not to include reachability analysis.

Recursive Types: As shown in Table 2, ASN1SPECT found 27 projects that have a recursive type. Recursive types are frequently used in ASN.1 protocols to define data structures, such as LDAP and IEC 61850. While not inherently problematic, recursion introduces potential risks when encoded types become excessively nested. An encoded type that is deeply nested will cause the decoding procedure in asn1c to reach the maximum function call depth. Given the relatively small number of findings, we manually validate whether or not these projects limited stack depth or provided some mitigation on the recursive types. No projects that we investigated had any mitigation for recursive types except for libIEC61850 [30]. Since their mitigation did not cover all cases, we created a proof-of-concept for how a recursive type can lead to a denial-of-service attack in electric grids using libiIEC61850 in Section 6.2.

Information Object Set Inconsistency: As shown in Table 2, ASN1SPECT identified the issue in 13 unique types across 8 projects on GitHub. These findings include vulnerabilities in NextEPC that persisted despite prior work identifying some of them [1]. We manually verified each to be a true positive.

6 Proof of Concept Exploits

We now discuss how we used ASN1SPECT's findings to create proof-of-concept on satellite ground receivers and electrical substations. We describe our responsible disclosure process in Section 7.1.

6.1 Satellite & Plane Communication

The mismatching constraints module discovered that the SatDump SDR processing does not enforce constraints for Inmarsat satellite data [25, 45]. SatDump is an active project and was last committed in Janurary 2026 [27, 45]. This flaw allows the transmission of invalid data to ground units by bypassing the input validation of constraints on data types. Depending on how the data is processed, these vulnerabilities can lead to denial of service or data integrity violations. SatDump is an SDR software processor, and other applications consume its output. libacars is a library that processes Aircraft Communications Addressing and Reporting System (ACARS) messages included in SatDump. libacars is actively maintained, with the newest release made in November 2025 [25, 27].

Listing 1 is a code sample from libacars for the `PlaneAltitude` type. The listing shows how libacars processes values for this type and shows that it does not enforce constraints. While the listing shows just one type, libacars has many types that are defined in ASN.1 and have similar issues. First, the asn1c defines the function that will enforce constraints at ①, which is determined at compile time. This function checks if the value is between 0 and 40,000 feet (②). During runtime, the `PlaneAltitude_decode` function is called to decode the value, which overwrites the statically defined constraint function. The function pointer is set to `asn_DEF_NativeInteger.check_constraints` which does not enforce any constraints (③). We created an example ACARS packet that violates the constraint on the `PlaneAltitude` type and when sent to libacars produces a result that should be rejected.

Listing 2 shows the example input into one of libacars' sample applications distributed with the library. This listing uses an example in the libacars repository to take a CPDLC position report message and extract the longitude, latitude, and altitude. We created an ACARS input for this program with nonsense values for latitude, longitude, altitude, and time in the message. Since the constraints on these data types are not enforced, the ACARS message is accepted and the invalid values are printed. These values should be rejected since the ASN.1 specification specifically forbids them. In the replaced parser for this program, these packets are rejected.

```
    int PlaneAltitude_constraint(...) {
        ...           ②
        return (value >= 0 && value <= 40000)
    }
    asn_dec_rval_t PlaneAltitude_decode(...) {
        ...                ③
        td->check_constraints = asn_DEF_NativeInteger.
             check_constraints;
        return td->ber_decoder(...);
    }
    asn_TYPE_descriptor_t asn_DEF_PlaneAltitude = {
        ...
        PlaneAltitude_constraint ①
    };
```

Listing 1: Constraint enforcement code exists in the generated code base. However, before decoding, the constraint enforcement is overwritten by the parent type's constraint enforcement function.

```
./cpdlc_get_position /MELCAYA.AT1.ZK-\
OKC253B21CC3D903BFFFFFFFFFFFFFF7F025B83128CD7886\
A12E9D85266B927584A9169C1A8EEB3800EEA7
 Latitude: -128.705000
 Longitude: -256.705000
 Altitude: 104600 ft
 Time: 31:63
```

Listing 2: Linux shell output when sending a malformed packet into the libacars library for processing. This packet should return a failure to decode the message since it violates the constraints on longitude, latitude, altitude, and time.

```

asn_MBR_DataSequence[] = {
    { ..., asn_DEF_Data, ... } ②
};

asn_DEF_DataSequence = {
    "DataSequence",
    ...,
    asn_MBR_DataSequence ①
};

asn_MBR_Data[] = {
    { ..., asn_DEF_DataSequence, ..., }
};

asn_DEF_Data = {
    "Data",
    ...,
    asn_MBR_Data
};
```

Listing 3: Code included in libIEC61850 for a `Data` type that is recursive. A recursive type can cause a stack overflow during encoding or decoding if not handled carefully.

6.2 Electrical Substations

The IEC61850 ASN.1 specification has a recursively defined data type. The recursive type analysis module detected this by analyzing libIEC61850. libIEC61850 is an actively-maintained open-source implementation of the IEC 61850 standard used in commercial products, with the latest commit made in December 2025 [27, 30]. IEC 61850 defines the design of electrical substation automation [30].

Listing 3 shows code in libIEC61850 that defines a recursive type. The `Data` type contains a member of type `DataSequence` ((①)), which in turn contains a member of type `Data` (②). This structure enables the creation of deeply nested hierarchies of `Data` objects.

Listing 4 demonstrates how to construct a deeply nested type that exploits the decoding process using asn1c. As the loop iterates,

```
create_nested_mms_pdu(depth):
    for i = 1 to depth-1:
        curr.list.count = 1
        curr.list.array = alloc(1 Data_t*)
        curr.list.array[0] = alloc(Data_t) ①
        curr.list.array[0].present = Data_PR_array
        next = alloc(DataSequence_t) ②
        curr.list.array[0].choice.array = next
        curr = next
    return m
```

Listing 4: This proof of concept creates a large nested PDU in MMS. When sending a packet with high depth, a stack overflow can occur for the decoder.

we are operating on a single instance of the `Data` type. This type contains a `DataSequence` type as one of its elements which contains a `Data` type. At ①, we fill an array with a `Data` type. Starting at ②, we perform the setup for the next loop iteration. The decode operation provided by asn1c causes a stack overflow and segmentation fault on a malicious packet containing roughly 15,000 nested types. By default, libIEC61850 allows packets of size up to 65 KB, and the malicious packet is 59 KB. A centralized SCADA system or an individual IED receiving the packet will cause denial of service causing significant outages in critical infrastructure.

7 Discussion

7.1 Responsible Disclosure

We reported both the recursive type and `ENUMERATED` types lacking constraints to the developers of both the base repository and mouse07410 fork since they are actively maintained and other forks are abandoned [27]. We created a pull request to resolve the `ENUMERATED` types issue in the mouse07410 fork, which was merged. Neither version is vulnerable to non-enforced constraints or the IOC inconsistency. Neither has released a new version addressing the recursive type problem. We also reported the problems to the package maintainers of Debian and OpenSUSE.

We submitted reports of the asn1c problems to MITRE, who assigned four CVEs. Each CVE corresponds to an underlying problem discussed in Section 5.3 that ASN1SPECT's analysis modules detect. We received CVE-2025-32891 for non-enforced constraints, CVE-2025-32895 for Encoding Specific Constraint Enforcement on `ENUMERATED` types, CVE-2025-32894 for Recursive Types, and CVE-2025-32892 for Information Object Set Inconsistency.

We reported the findings to all 40 affected projects. For projects for which we created proof-of-concepts, we also disclosed them to the developers. NextEPC's developers fixed the vulnerabilities in the repository [32]. One repository stated that while `ENUMERATED` types had no constraints, it did not have any security impact because they do not use those types in a way that would lead to an exploit. While the security impact might be mitigated by a project's current usage, determining if a specific `ENUMERATED` value is used in a security-sensitive setting is beyond the scope of ASN1SPECT. We have not received a response from the other affected projects.

7.2 Limitations

Specification Identification and Validation: Our analysis assumes access to the ASN.1 specification corresponding to each

parser. In most cases, ASN1SPECT extracts the ASN.1 specification files when they are in the same repository as the parser code. In 29 of the 107 projects, the ASN.1 specification was not present in the repository. In these cases, we manually identified the specifications and validated this choice using ASN1SPECT. Discrepancies in type matching during the differential analysis signaled an incorrect specification, prompting manual correction and re-analysis. This iterative feedback loop was crucial for confirming specification accuracy and preventing subsequent errors.

Manual Effort in Compiling Projects: Our analysis relies on symbolic execution (angr), and therefore requires producing a build artifact for each project. Other symbolic execution frameworks (e.g., KLEE [4]) also require project compilation. ASN1SPECT automatically clones and compiles GitHub projects, but a subset of those projects required manual intervention to compile. Common issues included missing dependencies, specific library versions, missing required compilation arguments, version mismatches, or missing project-specific build flags. When such fixes were required, we encoded them in ASN1SPECT to ensure reproducibility. Six projects required packages that were not available in the package repositories (e.g., python2), and five were Windows-only projects incompatible with our environment. This reduced our analysis from 107 identified projects to 93 successfully compiled projects.

8 Related Work

Formal Verification of ASN.1 parsing: A few works have proposed formally verifying ASN.1 parsing code [40]. ASN1* [33] provides a verified implementation of a non-malleable parser for ASN.1 DER. The importance of non-malleable DER parsing can manifest itself in subtle vulnerabilities. They provide an example of Microsoft's CryptoAPI incorrectly parsing a null character in the domain name of an X.509 TLS certificate to carry out a MITM attack. Recently, ARMOR [9] formally models ASN.1 DER. They use this model to formally prove an implementation correctly parses X.509 certificates. Unfortunately, both of these works focus solely on ASN.1 DER and do not consider other encoding rules.

The European Space Agency developed its own ASN.1 compiler (asn1scc) for embedded systems [26]. They claim a few advantages to their approach, with the first being that they automatically generate unit tests that exercise 100% of the code. asn1scc does not support the use of any dynamic memory functions such as `malloc`. Instead, all memory requirements are calculated when the ASN.1 specification is compiled, which ensures that all required memory is statically allocated. This prevents the use of some ASN.1 features as it would not be possible for the compiler to determine the amount of memory needed statically. Therefore, this compiler supports only a subset of available ASN.1 features.

Fuzzing ASN.1 & Cellular Networks: Several recent works have focused on fuzzing cellular network protocols that use ASN.1 decoding [1, 41]. However, they mainly focus on memory-related issues and not logic flaws. ASNFuzzGen [1] is a tool to mutate ASN.1 structures for fuzzing and requires a deep understanding of ASN.1 encoded values and where problems may occur. They fuzz interfaces between the RAN and core cellular network and identify 93 CVEs using this approach, including some in asn1c. Berserker is more focused on fuzzing the RRC protocol, which is a protocol between the UE and RAN. While we do find memory-related issues using static analysis, we also identify issues that a fuzzer could not easily by using differential analysis.

X.509 Certificate Validation: TLS certificate parsing is one of the most used applications of ASN.1 decoders. SSL certificate validation has been a hot topic for at least 10 years [16]. More recently, SymCerts [5] analyzed multiple implementations of X.509 certificate parsing libraries and compared the validation logic between the libraries. In addition to verifying ASN.1 DER, ARMOR [9] also formally models TLS certificate parsing and verifies implementations are identical to the model. Other works have analyzed SSL/TLS implementations using differential testing [6, 38, 42]. However, they are limited to certificate parsing and not other ASN.1 protocols.

9 Conclusion

ASN.1 compilers play a crucial role in ensuring the secure and reliable operation of critical applications that rely on ASN.1. This paper presents ASN1SPECT, a tool designed to statically detect parsing vulnerabilities in code generated by asn1c, a widely used open-source ASN.1 compiler. We applied ASN1SPECT to a dataset of 93 applications and found that nearly half (43%) have at least one of the four vulnerabilities. We demonstrated these vulnerabilities have real-world consequences by creating proof-of-concept attacks in electrical grid and satellite communication projects that are widely deployed on real infrastructure. Many, but not all, of the vulnerabilities can be resolved by generating parsing code with newer versions of the compiler. Continued use of ASN.1 in critical services necessitates advancement in analysis tools to ensure the security and resilience of ASN.1 parsing.

Acknowledgments

This work is supported in part by NSF grants CNS-2054911 and CNS-2055014. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

A Additional Analysis Modules

As mentioned in Section 4.2, ASN1SPECT includes two additional analysis modules, which identify recursive types and a vulnerability in specific uses of information object sets. This appendix describes these additional analysis modules in detail.

A.1 Recursive Type Analysis

Recursive Type Analysis is run on a single program and does not require differential analysis. Recall the recursive definition of the `Filter` type from Figure 1. ITU X.680 [50, Section 3.8.61] says "Recursive definitions are allowed in ASN.1: the user of the notation has the responsibility for ensuring that those values (of the resulting types) which are used have a finite representation and that the value set associated with the type contains at least one value."

Specifically, the application developer needs to ensure each ASN.1 parsing method does not receive an untrusted user input directly. Not all instances of a recursive type are vulnerable. For example, if the developer limits the stack depth during parsing, then it is not vulnerable. ASN1SPECT only reports if a type is recursive.

Algorithm 2: FindIOSMismatch

Input: Set of types T.
Output: Set Γ of IOC primary types inconsistent with the IOS.

```
Procedure FindIOSMismatch(T)
    Γ ← ∅;
    for t ∈ T do
        func ← GetTypeSelectorFunction(t);
        if func ≠ null then
            IOSArray ← FindIOSArray(func);
            (row, col) ← IOSArrayExtract(IOSArray);
            if |t.T| > 0 & |t.T| ≠ row then
                Γ ← Γ ∪ t;
    return Γ
```

A.1.1 Definitions. The recursive type analysis extends Section 4.1.1 with a definition for recursive types.

Definition 5 (Recursive Types)**.** Let $t = (T, c)$ be an ASN.1 type (Def. 1). A type t is *recursive* if its elements either directly or indirectly reference t. A direct reference exists if $\exists t_i \in t.T$ such that $t_i = t$. An indirect reference exists if $\exists t_i \in t.T$ such that t_i has an element that directly or indirectly references t.

A.1.2 Analysis. The Recursive Type Analysis algorithm detects if a type t is recursive. It goes through all ASN.1 types T extracted from the binary program. For each type $t_i \in T$, it checks if a cycle exists to determine if it is a recursively defined type. Conceptually, it is an optimized depth-first search traversal that stops as soon as a cycle is identified. The type and element structure conceptually creates a directed graph.

A.1.3 Analysis Module Specifics. ASN1SPECT internally uses the Recursive Type Analysis algorithm to detect recursive types while extracting types from a program, and flags when types are recursive. The recursive type analysis module reads this flag and outputs all recursive types. ASN1SPECT's analysis modules cannot mark types as recursive until the type extraction process is completed. Otherwise, the analysis module may try to access elements of a type that have not already been processed by ASN1SPECT.

A.2 Information Object Set (IOS) Analysis

IOSes are generated as arrays of structures in the code. They are represented by multiple data structures spread across the code base. The structures may have a conflicting number of elements. During the parsing of an information object, if it contains a field that exists in one of the data structures but not another, it leads to a program crash. This problem was identified by prior work using a fuzzer [1]. Our methodology can detect this problem statically and therefore find problems that are hard to identify using a fuzzer.

A.2.1 Definitions. ITU X.681 [51, Def. 3.4.10] states an information object class is "A set of fields, forming a template for the definition of a potentially unbounded collection of information objects, the instances of the class."

Definition 6 (Information Object Class)**.** Let $\mathbb{C}$ be a single ASN.1 information object class. Each class $\mathbb{C}$ consists of a set of fields F, which form a template for the definition. Each field $f \in F$ is a type t (Def. 1).

Conceptually, an information object is a group of types where each type in the group is the value of the fields defined by the information object class. ITU X.681 [51, Def 3.4.9] states an information object is "An instance of some information object class, being composed of a set of fields which conform to the field specifications of the class."

Definition 7 (Information Object)**.** Let $O_{\mathbb{C}}$ be an information object that conforms to an information object class $\mathbb{C}$. Each $O_{\mathbb{C}}$ is a set of value-type tuples (v, t), which are instances of the fields F in $\mathbb{C}$.

ITU X.681 [51, Def. 3.4.12] states that an information object set is "A non-empty set of information objects, all defined using the same information object class reference name." We found that asn1c flattens all objects in an information object set into a single array such that the value-type tuples are concatenated. For example, let O_1, O_2, and O_3 be information objects for class $\mathbb{C}$, where $\mathbb{C}$ has fields t_1 and t_2. asn1c generates a single array containing (v_{11}, t_1), (v_{12}, t_2), (v_{21}, t_1), (v_{22}, t_2), (v_{31}, t_1), (v_{32}, t_2).

Conventionally,[2] an IOC contains one value. The other fields contain a unique identifier and other properties about that one value. ASN1SPECT must identify which field is that one value from the program binary.

A.2.2 Analysis. The *FindIOSMismatch* algorithm (Algorithm 2) identifies when the number of elements in an information object set is inconsistent with the possible set of values defined for the type corresponding to the primary value for the information object class. *GetTypeSelectorFunction*$(\cdot)$ checks each type for a type selector function, which only exists if it is a field in an IOC. If the type has a type selector function, *FindIOSArray*$(\cdot)$ performs symbolic execution on the function to identify the name of the corresponding IOS array. *IOSArrayExtract*$(\cdot)$ then extracts the number of rows and columns in the IOS array. Then, the number of rows is compared against the number of elements in t. If these are not the same, the algorithm adds t to the set of types for manual review.

A.2.3 Analysis Module Specifics. The implementation of *GetType SelectorFunction*$(\cdot)$ involves extracting the type selector function pointer from the data structure that defines an ASN.1 type. *Find IOSArray*$(\cdot)$ is implemented using symbolic execution from the beginning of the function until a particular `lea` assembly instruction is reached. The address that is loaded with `lea` is the Information Object Set array, which ASN1SPECT loads. Finally, it compares the number of elements in the array to the type definition.

References

[1] Nathaniel Bennett, Weidong Zhu, Benjamin Simon, Ryon Kennedy, William Enck, Patrick Traynor, and Kevin R. B. Butler. 2024. RANsacked: a domain-informed approach for fuzzing LTE and 5G RAN-core interfaces. https://doi.org/10.1145/3658644.3670320. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery, Salt Lake City, UT, USA, 2027–2041. https://doi.org/10.1145/3658644.3670320

[2] Bosch. 2022. Bosch - Open-Source Software Attributions for XC_CT_RN_AIVI_Scope 3.x. https://oss.bosch-cm.com/download/Mitsubishi/5128_210730/OSS_Disclosure_Document_5128_210730.pdf.

[3] brchiu. 2017. Simplify the logic of accessing codec function for specific TYPE · vlm/asn1c@1fa31c9. https://github.com/vlm/asn1c/commit/1fa31c9e3e4182251439e8d5795fe4abf73f3152.

[2] ITU X.861 Annex B [51, Annex B] shows this convention. All of the information object classes in our dataset follow this convention and only have one primary value.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, San Diego, CA, 209–224. https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar.pdf

[5] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. http://ieeexplore.ieee.org/document/7958595/. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 503–520. https://doi.org/10.1109/SP.2017.40

[6] Chu Chen, Pinghong Ren, Zhenhua Duan, Cong Tian, Xu Lu, and Bin Yu. 2023. SBDT: Search-Based Differential Testing of Certificate Parsers in SSL/TLS Implementations. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 967–979. https://doi.org/10.1145/3597926.3598110

[7] Inc. Cisco Systems. 2017. Open Source Used In Samsung 7252S v1. https://linqcdn.avbportal.com/documents/2b290775-0cca-4d76-8844-2f5f77b59867.pdf.

[8] Inc. Cisco Systems. 2025. Open Source Used In Cisco Secure Endpoint Connector (Mac) 1.24.0. https://www.cisco.com/c/dam/en_us/about/doing_business/open_source/docs/CiscoSecureEndpointConnectorMac-1270-1747841284.pdf.

[9] Joyanta Debnath, Christa Jenkins, Yuteng Sun, Sze Yiu Chau, and Omar Chowdhury. 2024. ARMOR: A Formally Verified Implementation of X.509 Certificate Chain Validation. In *2024 IEEE Symposium on Security and Privacy (SP)*. 1462–1480. https://doi.org/10.1109/SP54263.2024.00220 ISSN: 2375-1207.

[10] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. 2023. Finding Fixed Vulnerabilities with Off-the-Shelf Static Analysis. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*.

[11] William Enck and Laurie Williams. 2022. Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations. *IEEE Security and Privacy Magazine* 20, 2 (March/April 2022).

[12] esnacc. esnacc/esnacc-ng: A continuation of the Enhanced SNACC ASN.1 compiler. https://github.com/esnacc/esnacc-ng.

[13] European Space Agency. 2024. ASN1SCC: An open source ASN.1 compiler for embedded systems. https://github.com/esa/asn1scc.

[14] OpenSSL Software Foundation. 2018. CVE - CVE-2018-0739. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0739.

[15] gatopeich. 2021. Version string is always the same even across forks! · Issue #75 · mouse07410/asn1c. https://github.com/mouse07410/asn1c/issues/75.

[16] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/2382196.2382204

[17] Google. 2024. protocolbuffers/protobuf. https://github.com/protocolbuffers/protobuf.

[18] gssapi. 2025. gssapi/mod_auth_gssapi. https://github.com/gssapi/mod_auth_gssapi.

[19] Hao He, Haoqin Yang, Philipp Burckhardt, Alexandros Kapravelos, Bogdan Vasilescu, and Christian Kästner. 2026. Six Million (Suspected) Fake Stars on GitHub: A Growing Spiral of Popularity Contests, Spams, and Malware. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[20] ICAO. 2016. Procedures for Air Navigation Services (PANS) - Air Traffic Management (Doc 4444). https://store.icao.int/en/procedures-for-air-navigation-services-air-traffic-management-doc-4444.

[21] Objective Systems Inc. 2024. License Management FAQ. https://obj-sys.com/support/license-faq.php.

[22] ISO. [n. d.]. Industrial automation systems — Manufacturing Message Specification. https://www.iso.org/standard/37079.html.

[23] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[24] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[25] Tomasz Lemiech. 2024. szpajder/libacars. https://github.com/szpajder/libacars.

[26] George Mamais, Thanassis Tsiodras, David Lesens, and Maxime Perrotin. 2012. An ASN.1 compiler for embedded/space systems. *Embedded Real Time Software and Systems (ERTS)* (Feb. 2012).

[27] Courtney Miller, Mahmoud Jahanshahi, Audris Mockus, Bogdan Vasilescu, and Christian Kastner. 2025. Understanding the Response to Open-Source Dependency Abandonment in the Npm Ecosystem. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2355–2367. https://doi.org/10.1109/ICSE55347.2025.00004

[28] MITRE. [n. d.]. NVD - CVE-2016-4421. https://nvd.nist.gov/vuln/detail/CVE-2016-4421.

[29] mouse07410. 2024. mouse07410/asn1c. https://github.com/mouse07410/asn1c.

[30] mz automation. 2024. mz-automation/libiec61850. https://github.com/mz-automation/libiec61850.

[31] Juniper Networks. 2015. Juniper Networks® Steel-Belted Radius® Carrier Administration and Configuration Guide. https://www.juniper.net/documentation/en_US/sbr-carrier8.1.0/information-products/topic-collections/sbr-admin-guide-10/book-sw-sbrc-admin.pdf.

[32] NextEPC. 2025. nextepc/nextepc. https://github.com/nextepc/nextepc.

[33] Haobin Ni, Antoine Delignat-Lavaud, Cédric Fournet, Tahina Ramananandro, and Nikhil Swamy. 2023. ASN1*: Provably Correct, Non-malleable Parsing for ASN.1 DER. https://dl.acm.org/doi/10.1145/3573105.3575684. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, Boston MA USA, 275–289. https://doi.org/10.1145/3573105.3575684

[34] OSS Nokalva. 2024. Product Licensing. https://www.oss.com/products/purchase.html.

[35] OSS Nokalva. 2025. ASN.1 Made Simple - Constraints. https://www.oss.com/asn1/resources/asn1-made-simple/advanced-constraints.html.

[36] Oracle. 2018. Oracle Communications Control Plane Monitor Licensing Information User Manual. https://docs.oracle.com/communications/E89194_01/doc.40/om_40_licensing_information.pdf.

[37] Terence Parr. 2025. ANTLR. https://www.antlr.org/.

[38] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 615–632. https://doi.org/10.1109/SP.2017.27

[39] Martin Pluskal. 2026. OpenSUSE:Factory Build Status of asn1c. https://build.opensuse.org/package/show/devel:tools:compiler/asn1c.

[40] Nika Pona and Vadim Zaliva. 2020. Research Report: Formally-Verified ASN.1 Protocol C-language Stack. https://ieeexplore.ieee.org/document/9283873/. In *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE, San Francisco, CA, USA, 308–317. https://doi.org/10.1109/SPW50608.2020.00065

[41] Srinath Potnuru and Prajwol Kumar Nakarmi. 2021. Berserker: ASN.1-based Fuzzing of Radio Resource Control Protocol for 4G and 5G. In *2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 295–300. https://doi.org/10.1109/WiMob52687.2021.9606317 ISSN: 2160-4894.

[42] Lili Quan, Qianyu Guo, Hongxu Chen, Xiaofei Xie, Xiaohong Li, Yang Liu, and Jing Hu. 2020. SADT: Syntax-Aware Differential Testing of Certificate Validation in SSL/TLS Implementations. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 524–535. https://ieeexplore.ieee.org/document/9286011/?arnumber=9286011 ISSN: 2643-1572.

[43] Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified secure Zero-Copy parsers for authenticated message formats. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1465–1482. https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

[44] Joseph L Rios and Moffett Field. 2015. A Formal Messaging Notation for Alaskan Aviation Data. https://ntrs.nasa.gov/api/citations/20150022356/downloads/20150022356.pdf.

[45] SatDump. 2024. SatDump/SatDump. https://github.com/SatDump/SatDump.

[46] Jürgen Schönwälder, David T. Perkins, and Keith McCloghrie. 1999. *Structure of Management Information Version 2 (SMIv2)*. Request for Comments RFC 2578. Internet Engineering Task Force. https://doi.org/10.17487/RFC2578 Num Pages: 42.

[47] ITU Telecommunication Standardization Sector. 1988. Specification of Abstract Syntax Notation One (ASN.1). https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.208-198811-W!!PDF-E&type=items.

[48] ITU Telecommunication Standardization Sector. 1988. X.400 : Message handling system and service overview. https://www.itu.int/rec/T-REC-F.400-198811-S/en.

[49] ITU Telecommunication Standardization Sector. 2019. X.509: Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks. https://www.itu.int/rec/T-REC-X.509.

[50] ITU Telecommunication Standardization Sector. 2021. X.680: Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. https://www.itu.int/rec/t-rec-x.680/en.

[51] ITU Telecommunication Standardization Sector. 2021. X.681: Information technology - Abstract Syntax Notation One (ASN.1): Information object specification. https://www.itu.int/rec/T-REC-X.681/en.

[52] Eugene Seliverstov. 2026. Debian – Details of package asn1c in trixie. https://packages.debian.org/trixie/asn1c.

[53] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, 138–157. https://doi.org/10.1109/SP.2016.17

[54] Siemens. 2023. Siemens - License Conditions. https://cache.industry.siemens.com/dl/files/710/109825710/att_1158963/v1/OSS_SINEC-INS_99.pdf.

[55] Telecom Behnke. 2019. Telecom Behnke - Wichtige Lizenzinformation. https://www.behnke-online.de/de/downloads/lizenzinformation/488-aktuelle-

lizenzinformationen/file.
[56] Willy R. Vasquez, Stephen Checkoway, and Hovav Shacham. 2023. The most dangerous codec in the world: Finding and exploiting vulnerabilities in H.264 decoders. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6647–6664. https://www.usenix.org/conference/usenixsecurity23/presentation/vasquez
[57] Vasil Velichkov. 2024. velichkov/asn1c. https://github.com/velichkov/asn1c.
[58] Viasat. 2022. Viasat - Open Source Software Notice. https://www.viasat.com/content/dam/us-site/corporate/documents/MobileDynamicDefense_open_source_notices_for_web_page.pdf.
[59] Viasat. 2024. Classic Aero. https://www.viasat.com/aviation/flight-operations-and-safety/classic-aero/.
[60] Lev Walkin. 2009. asn1c in iPhone. https://lionet.livejournal.com/43412.html.
[61] Lev Walkin. 2013. fix default constraint checking · vlm/asn1c@6169b8d. https://github.com/vlm/asn1c/commit/6169b8d5654ca53692ea50ca573408a1321a50f7.
[62] Lev Walkin. 2016. Where is my code? https://lionet.livejournal.com/138575.html.
[63] Lev Walkin. 2017. Open Source ASN.1 Compiler: asn1c 0.9.28. https://lionet.info/asn1c/compiler.html.
[64] Zichao Zhang, Limin Jia, and Corina Păsăreanu. 2024. ProInspector: Uncovering Logical Bugs in Protocol Implementations. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. 617–632. https://doi.org/10.1109/EuroSP60621.2024.00040 ISSN: 2995-1356.
[65] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E. Hassan. 2023. CoLeFunDa: Explainable Silent Vulnerability Fix Identification. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.
[66] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E. Hassan. 2021. Finding A Needle in a Haystack: Automated Mining of Silent Vulnerability Fixes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Received 2026-02-02; accepted 2026-03-19