Go or No Go: Differential Fuzzing of Native and C Libraries

Alessandro Sorniotti
IBM Research Europe – Zürich
Switzerland
aso@zurich.ibm.com

Michael Weissbacher
Block, Inc.
USA
mweissbacher@squareup.com

Anil Kurmus
IBM Research Europe – Zürich
Switzerland
kur@zurich.ibm.com

***Abstract*—In little more than a decade, Go has become one of the most popular programming languages in use today. It is a statically-typed, compiled language with spatial and temporal memory safety achieved by way of strong typing, automatically inserted bounds checks, and a mark-and-sweep garbage collector. Go developers can make immediate use of a large set of native libraries, whether shipped as part of the runtime or available to be imported from community code. Alternatively Go developers can directly link to C/C++ libraries which can be called from Go sources thanks to `cgo` functionality. Factors that go into this decision are stability, performance, and availability. As a result developers have a choice between Go native libraries or non-native code. However, today there is little understanding how to consider security implications in this decision.**

Our work is the first to investigate security implications of choosing between native and non-native libraries for Go programs. We first investigate to what extent popular GitHub projects make use of `cgo`, revealing that this choice is in fact quite popular. We then design and build a differential fuzzer that can compare native and C/C++ implementations of the same functionality. We implement the fuzzer and test its effectiveness on four popular packages (`libcrypto`, `libpng`, `libssl`, and `libz`), describing the results and highlighting their security impact. Finally, we present two real-world case studies (anti-virus evasion including the anti-virus scanner included in Gmail plus Certificate Transparency case study) and discuss how our differential fuzzer discovered implementation differences with security impact. Our work has led to changes in Golang `zlib` which have since shipped.

***Index Terms*—Computer security; Application Security; Fuzzing; Language Security;**

1. Introduction

Go is a programming language that is widely celebrated for its security benefits and concurrency. The language is memory safe by default and overcomes decades of security problems by languages such as C/C++. However, when making use of libraries to leverage commonly used functionality, developers are faced with a difficult choice: Use native Go libraries or non-native C/C++ (using the `cgo` feature of the compiler/runtime). Factors that go into this decision today are availability, performance, and stability.

While Go offers a wide array of libraries, not all libraries are available in Go and developers might be required to reimplement functionality by following specifications. This is expensive and error-prone. Performance is also a consideration when making the decision between native and non-native libraries, as using C/C++ libraries can bring performance advantages to programs that make heavy use of such functionality. Stability is another factor since libraries can have different maturity levels. Go libraries are implemented from a clean slate but might be missing extensive testing, whereas C/C++ libraries might have stale code, but have gone through decades of testing.

Beyond the factors of availability, performance, stability, security implications should also be considered when making this decision, because there are security issues associated with either option. In the case where the developer reuses existing `C` code, they run the risk of introducing memory safety issues that may exist in the `C` library. In the case of a Go reimplementation, new defects that do not exist in the `C` library version may be introduced, at the specification or implementation level. More broadly, libraries designed and implemented differently will likely deviate in their outputs. From a language-theoretic point of view, we can see this as an instance of the *parse tree differential* attack [38], [30], [37]: two different parsers exist for the same protocols/languages, and the differences between them may lead to security issues. Unfortunately, formally verifying whether two parsers are equivalent is undecidable for context-free languages [29], which motivates pragmatic, albeit incomplete, approaches such as greybox fuzzing.

In this paper, we propose and implement an approach to study the problems deriving from the use of native vs. non-native libraries in the case of the Go programming language. In particular, we look for differences between these two types of libraries by making use of *differential fuzzing*, focusing on commonly used libraries where differences may lead to widespread security impact.

We take an effective approach to differential fuzzing, by making use of an existing fuzzer targeting either of the two library versions to generate an input corpus and a harness. For each input, we compare the output (and possibly other side effects) of both libraries. For instance,

2023 IEEE Security and Privacy Workshops (SPW) | 979-8-3503-1236-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/SPW59333.2023.00036

DOI 10.1109/SPW59333.2023.00036

in the case of a parsing library, if the same input results in two different parsed outputs, a potential defect may exist. Based on the parsing results alone it is not immediately clear which of the two parsers deviated from the specification, if any. Specifically, security vulnerabilities can arise from the difference alone, rather than one parser misinterpreting input.

We develop and evaluate our differential fuzzer on four popular libraries and explore differences between the two implementations (native vs. non-native). We discuss the findings for three libraries in detail, to reveal the root cause and gauge security implications. For the fourth library, `libpng`, our findings had no justifiable security impact.

We also present two case studies: in the first, we show how to corrupt the headers of libz-compressed data, leading to evasion in 19 anti-virus (AV) products and the AV scanner included in Gmail. In the second, provided in the Appendix, we show how parser mismatches can impact Certificate Transparency infrastructure. For all affected products we initiated responsible disclosure processes.

Our findings demonstrate that: (i) From the security point of view, the choice for a Go developer between a Go implemented or C/C++ library is not straightforward; (ii) Differential fuzzing is an effective approach to find parser differences between library implementations, and should be employed systematically on Go re-implementations of C libraries; (iii) Parser differences may lead to important security issues.

To summarize, the contributions of this paper are the following:

- We design a custom differential fuzzer to uncover security relevant parsing differences in libraries used in Go programs.
- We implement a prototype of our design, with support for widely used libraries: `libcrypto`, `libpng`, `libssl`, and `libz`.
- We evaluate the fuzzer extensively on unique differences and analyse root cause and potential impact.
- We present two case studies where discovered parser differences lead to security impact. One study affecting Certificate Transparency and one bypass for 19 AV systems tested on Virus Total, plus Gmail.

2. Analysis of GitHub Go code

In this section we describe our analysis of native code use in popular GitHub repositories. We crawled GitHub checking Go repositories for usage of two packages (`unsafe`, `C`) as well as for `.s` files. `unsafe` permits circumventing some of the spatial memory safety guarantees of Go such as C-style pointer arithmetic or casts that override the type system. The `C` package contains functions that are directly callable from Go and yet are written in C. Furthermore, `.s` files are source files in the Go assembly language which is usually lowered into analogous platform assembly.

We implemented the crawler using `astutil` package to walk the AST. We scan for function invocations rather than imports for side effect reasons and also ignore autogenerated code.

The analysis revealed widespread use for both `unsafe` and `C` while `.s` usage was low. Further we compare usage of `unsafe` and `C` function calls to regular Go function calls. We found that `unsafe` functions are on average shorter, while regular Go functions have on average longer comments.

We attempted to extend the automated Go source code analysis to report what the most common targets of non-native invocations are in Go programs. This requires an automated way to classify uses of `cgo`; one way is to extrapolate the target from the `cgo` headers (include directives and linker flags). However, it is often the case that `cgo` is used as glue code or to implement wrappers or specific functionality which is not immediately attributable to a specific target through metadata. We therefore leave the task of automating the target identification of `cgo` invocations as an item of future work.

2.1. Dataset

We use stars as proxy for community interest in the projects and analysed the highest rated ones. We scanned 1,001 projects total, ranging from 94,516 stars to 2,312 stars[1]. We believe this is a representative corpus to analyse, we consider repositories of widely used code such as `golang/go`, `kubernetes/kubernetes`, `prometheus/prometheus`. This experiment was performed in January 2022.

2.2. Findings

Overall we found that 298 repositories use `unsafe` (29.77%), 550 use `C` (54.95%) and 51 include `.s` files (5.09%). The popularity of `C` and `unsafe` code seems surprising, especially considering 23.68% of the top repositories combine both `C` and `unsafe`. The detailed breakdown of popularity by kind of Go code is in Table 1.

To gauge function complexity and how well documented different kinds of Go code are we collected metrics on the top GitHub repositories. We use function length as proxy for complexity and characters used to comment functions. The average `unsafe` function has the least lines as well as least characters used for comments. For `C` code the median function length one line below other kinds, and on average the functions are as long as regular Go code. The data is broken down in detail in Table 2.

2.3. Motivation

Go provides many libraries implemented natively, but developers often choose to include code written in C/C++, whether for performance, to avoid re-implementation, or otherwise. While our crawl does not allow to reason why

1. Note: we planned to analyse the top 1,000 – however, a tie between two repositories led us to consider 1,001 repositories instead

TABLE 1. BREAKDOWN OF `UNSAFE`, `C`, AND `.S` USAGE ACROSS TOP GITHUB GO REPOSITORIES, 1,001 TOTAL. WE NOTICE THAT `C` CODE IS POPULAR (54.95%) WHILE `S` IS ONLY USED IN 5.09%. OVERALL, 38.86% OF THE ANALYSED REPOSITORIES CONTAIN NONE OF THE THREE PROPERTIES UNDER ANALYSIS.

Type	Repositories
Unsafe	298 (29.77%)
C	550 (54.95%)
S	51 (5.09%)
C+Unsafe	237 (23.68%)
C+S	48 (4.80%)
S+Unsafe	47 (4.70%)
C+S+Unsafe	45 (4.50%)
No C or Unsafe	390 (38.96%)
No C, S, or Unsafe	389 (38.86%)
All	1,001 (100%)

developers use `C` and `unsafe`, it raises questions for interoperability of Go software when interacting with other systems. In this case, parsing problems that might occur in distributed systems using different languages, might occur even between systems using the same language.

The popularity of Go motivated our research in finding differences between libraries which are intended to be equivalent but often in practice are not. These discrepancies can lead to an array of security problems where a fuzzer can help in discovery.

3. Differential fuzzer

A *differential* fuzzer provides the same input to two different programs, and compares the outputs (or side effects) to detect differing behaviour. In this section, we describe the high-level design and implementation of a differential fuzzer we built with the goal of finding differing behaviour between `C` libraries invoked through `cgo` and the same libraries rewritten natively in Go.

We summarise first the main challenges involved in the design and implementation of a fuzzer with the objective to find security-relevant implementation differences between Go and `C` libraries:

- The language gap between the two worlds needs to be fixed: Go and `C` do not enjoy binary compatibility, nor do they have a common fuzzing framework. While it is possible under certain circumstances to call `C` code from Go and vice-versa, those mechanism are not fuzzer-friendly.
- Create a meaningful notion of *difference* in the behaviour of the two libraries. This is of fundamental importance since without such notion it is impossible to tell whether the two libraries under investigation behave in the same way or not. This task is rendered more complex by the aforementioned language gap.
- Since we focus on the security aspects of the Go vs. `C` choice, finding differences in the behaviour of the two libraries is not sufficient: we must also show that these differences have a security impact.

3.1. Design

Our differential fuzzer consists of three main components: input generation, input/output harness, and comparison functionality. These last two components are specific to a chosen library and require manual adjustments for each target library.

Input generation This component is responsible for creating inputs that will be provided to both libraries. We opt for a coverage-guided fuzzer design, given the success of existing fuzzers such as AFL, libFuzzer, or honggfuzz. The input generation makes use of code coverage in the library code to select inputs that are more interesting, and mutate such inputs with higher priority over other inputs that do not increase coverage. For the initial corpus, we reuse any available corpus used for fuzzing the selected library.

Input/Output harness As in existing fuzzers, we write harnesses that use the inputs from the previous phase to invoke functions from the target libraries. In contrast with fuzzers targeting single libraries, we need to format these inputs differently for each of the two targets libraries, and collect outputs from both libraries. In addition, the targeted library functionality needs to be deterministic in the input provided, and largely stateless.

Comparison functionality In this step, the two sets of collected outputs are compared. For each library, a different comparison function needs to be written to accurately identify differences in output.

3.2. Implementation

This section describes the implementation of our differential fuzzer, based on the coverage-guided fuzzer `go-fuzz` [43]. As discussed previously, we target semantically equivalent implementations in Go and C/C++. We assume that the starting point is a fuzzer native to one of those two language (e.g. AFL [46] for C/C++ or `go-fuzz` [43] for Go). A first challenge lies in the fact that neither of these fuzzers allows a harness developer to simply issue calls to functions belonging to the implementation in the other language: for example, it is not immediately possible to call a function of the Go implementation of the SSL protocol from the AFL harness of `libssl`. We evaluate both options – native C/C++ fuzzer and harness with custom Go binding, and native Go fuzzer and harness with custom C/C++ binding – and identify the latter as the most effective approach to prototype. The reason is that the Go runtime includes the `cgo` mechanism to call C/C++ shared objects; no such mechanism exists for the case of a C/C++ binary calling a Go function.

Our fuzzer builds upon a native Go fuzzer and corpus, `go-fuzz` [43] and `go-fuzz-corpus` [42]. The first project implements the fuzzing engine and the modifications required of the runtime to deliver code coverage information, whereas the second contains harnesses and corpora for a set of modules (e.g. ASN.1, png, tls...). The harness for a given module is developed by implementing the following function

TABLE 2. SUMMARY OF THE LENGTH OF FUNCTIONS AND THE EMBEDDED CODE COMMENTS. WE FOUND THAT PLAIN GO FUNCTIONS HAVE ON AVERAGE LONGER COMMENTS, WHILE `UNSAFE` FUNCTIONS ARE ON AVERAGE SHORTER. WE DO NOT CLAIM THAT THESE STATISTICS ARE RELATED TO CODE QUALITY. NOTE THAT ONLY GO/`CGO` CODE IS CONSIDERED HERE, S FILES ARE EXCLUDED AS THEY CANNOT BE DIRECTLY COMPARED TO IN TERMS OF LENGTH.

Type	Function Lines		Comment Characters per Function		Functions
	Median	Average	Median	Average	
Unsafe	7	11	61	88	67,853
C	6	16	61	150	25,122
C+Unsafe	7	12	61	105	92,975
No C or Unsafe	7	16	53	175	1,820,009
All	7	16	55	172	1,912,984

```
func Fuzz(data []byte) int
```

taking the fuzzer-generated byte slice as input and returning an integer that signal whether the considered mutation is an interesting one or not.

To implement a differential fuzzer for a specific module in Go, we first need to identify a `C` implementation of that module. Next, we need to implement a semantically equivalent Go API that calls the `C` implementation of the selected module. The existing harness can then be extended by introducing – after each call to the Go module – the equivalent call to the `C` one.

3.2.1. `C` implementation and harness. Calling the `C` implementation from Go can be achieved using `cgo`. It is a mechanism directly implemented by the Go compiler and runtime. A developer imports a pseudo-package which exposes callable functions included as `C` sources, which then become callable from the Go sources. The compiler automatically invokes `gcc` to build the `C` sources and it introduces appropriate trampoline code to transition between Go and `C` implementation.

Unfortunately `cgo` is not immediately usable since `go-fuzz` does not support it directly [13]. We solve the problem by creating a Go plugin to package the `C` dependency, and invoke the plugin from the harness, thus removing the need for `go-fuzz` to directly build a `cgo` project. This approach is still not viable since `go-fuzz` builds the instrumented binary with its own `GOROOT` and `GOPATH` environment with an appropriately instrumented runtime to extract coverage information. However, in order to ensure binary compatibility with the program that uses them, Go plugins must be built against the identical runtime; the compatibility check is executed at runtime to ensure that the link-time package hashes in the plugin are identical to those of the current runtime. We choose to modify the Go runtime to remove this check and let the fuzzer binary load the plugin and let the harness call it. While this option is not ideal from a maintainability perspective, it is sufficient to prototype the system and analyse the results it produces. Recently the Go community has added native fuzzing support to the runtime [15], which was released with Go 1.18.

The reader will notice that with this approach the fuzzer cannot use code coverage information from the `C` implementation, but does obtain coverage information from the existing native Go implementation. This is a current limitation of our fuzzer which may be improved in future releases by enabling native code coverage in the `cgo` build, extracting such information and feeding it to the fuzzer.

3.2.2. Output handling. With this approach we can develop a set of semantically equivalent API to invoke from the harness alongside the Go ones. Ideally, both implementations would behave identically: that is, they would either both successfully process correct input, or both return errors for malformed input. Errors and panics need to be handled carefully: while traditional fuzzing mainly targets abnormal program executions (e.g. a crash), our framework is mainly interested in cases where the behaviour of the two implementations diverges. For example, a scenario of interest is one where one of the two implementations successfully handles an input whereas the other returns a processing error (or crashes).

Towards this goal, we extend the harness of `go-fuzz-corpus` (for the selected targets) to pair each call to the fuzzed target with an equivalent call to the `C` implementation. Each pair of calls must behave in a *semantically similar* way. The definition of semantic similarity varies from call to call, and is best defined by example: two functions that can only return error/no error qualify for semantic similarity under our definition if (given identical input) they either always both return an error or no error. For functions that also provide a return value, semantic similarity is defined by requiring that the returned value be deeply equal. For instance, if we consider the ASN.1 component, its parsing functions returns `(error, interface{})`. We then define semantic similarity to mean that either both functions return an error, or they both return an interface that can be typecast to the same structure, and that furthermore the two structures are deeply equal.

`go-fuzz` lets harness developers encode the usefulness of the mutation in the return value of the harness, so that the fuzzer knows whether to further mutate it or whether to drop it (a positive value encodes an interest in the mutation). An initial attempt to return a positive value whenever the two implementations return conflicting results caused the fuzzer to insist on minor mutations of an input that keep exercising the same code path (and hence, the same difference between the two implementations). Among others – the previously mentioned lack of coverage from the `C` implementation might be a contributing factor to this behaviour. We have modified the harness to catch panics/errors generated by

such scenarios, report them back to the user and then return -1 to signal no further interest in the mutation. A deeper analysis of the root cause for the original behaviour might reveal more effective solutions. We leave this as an item for future work.

4. Evaluation

For our differential fuzzer prototype, we target four popular Go modules, `encoding/asn1`, `image/png`, `crypto/x509`, and `compress/zlib`. The reasons for this selection are the following: i) all usually handle security-sensitive input; ii) each component has a stable implementation in widely used C libraries (`libcrypto`, `libpng`, `libssl`, and `libz`); iii) native harnesses in `go-fuzz-corpus` exist for all four modules; and iv) it is possible to create C APIs that are semantically equivalent to the Go modules. One library that is prominent but absent from this evaluation is JSON. The reason for its exclusion, despite its popularity, is that the JSON standard is not versioned and inconsistent across implementations[2]. As such, it would make for a poor target to evaluate as inconsistencies are expected.

ASN.1 The existing harness for `encoding/asn1` attempts to unmarshal the input provided to the `Fuzz` function as a pre-defined set of data types (integers, identifiers, various types of strings, time-related object, and a custom struct), then marshals and unmarshals them again and compares the resulting output, asserting it to be equal to the initially decoded value. Panics are not filtered and a positive return value is returned whenever the initial unmarshalling was successful. We create Go API for marshalling and unmarshalling backed by `libcrypto`: in particular we use `ASN1_item_d2i` for unmarshalling and `ASN1_item_i2d` for marshalling, paired with two sets of constructors (Go and `C`) converting pointers to C types into pointers to Go types and vice versa. Then we extend the harness by adding a call to the C marshaller/unmarshaller whenever a call to the Go module is issued, and assert a semantically similar behaviour for each pair of calls.

png The existing harness for `image/png` interprets the supplied bytes as a PNG image; these bytes are initially passed to `png.DecodeConfig` which decodes the metadata header in the first bytes of the image, followed by a call to `png.Decode` to attempt to decode the full image. If this step succeeds, the image is encoded (using all available compression levels), then re-decoded and the resulting image checked for equality against the one originally decoded. We create Go API to decode PNG metadata and data using `libpng`. In particular, after constructing a read primitive from byte and setting error handling, we call `png_read_png` to decode the entire image, followed by `png_get_IHDR` to extract PNG metadata. We then check that metadata extracted with the native implementation is equal to the one extracted by the Go plugin.

X.509 The `crypto/x509` package supports parsing and using a number of artefacts from the X.509 standard, that includes cryptographic material, certificates, revocation lists and so forth. We focus on certificates and its existing harness, `FuzzCertificate`, which attempts to parse the supplied bytes as a certificate, and then performs a number of operations on the returned object (e.g. check the CA signature on subject). We use the certificate parser supplied by `libcrypto` (in the specific, by the `d2i_X509` function) to implement a certificate parser equivalent to Go's `x509.ParseCertificate`. We then extend the harness to call both parsers and assert semantic similarity by expecting equivalent error treatment and equivalent certificate metadata for a successfully-parsed certificate.

zlib The harness for `compress/zlib` interprets the supplied bytes as DEFLATE-compressed input and attempts a decompression; if it is successful it compresses it (for all available compression levels), decompresses it again and checks for bytewise equality with the outcome of the initial decompression. We provide a Go implementation of a reader/writer pair backed by `libz`, where compression and decompression are achieved by calling `deflate` and `inflate`, respectively. The harness is then extended by pairing each compression and decompression with its `C` counterpart, and asserting bytewise equality of compressed and plain data. We also exercise the Go decompressor on the `C` (re)compressed data and vice versa, always asserting bytewise equality.

4.1. Experimental setup

We fuzzed all targets (`libcrypto`, `libpng`, `libssl`, and `libz`) for 12 hours per run, `go-fuzz-corpus` configured to restart after 10 minutes, repeating the experiment 5 times with a total fuzz time of 60 hours per target. We used the standard corpus provided by github.com/dvyukov/go-fuzz-corpus with one exception. For the X.509 target we used additional corpora[3], because starting with the standard corpus yielded no results for each of the 12 hour runs.

We execute the tests 5 times each for repeatability and run over 12 hours each. To make tests independent we reset state between each run. For our experiments, we use related work on evaluation of fuzzers as a guide [31]. However, in this work we do not compare directly against other similar fuzzers.

As opposed to unique crashes which fuzzers usually consider as metric, we use unique differences in errors discovered, applying deduplication logic. We ran experiments on a commodity hardware server (Intel i7 8 Core CPU, 32GB).

4.2. Results

In this section we discuss in depth some of the findings reported by our differential fuzzer, with considerations about their possible security impact. Despite implementing the

2. http://seriot.ch/parsing_json.php

3. http://fm4dd.com/openssl/certexamples.htm

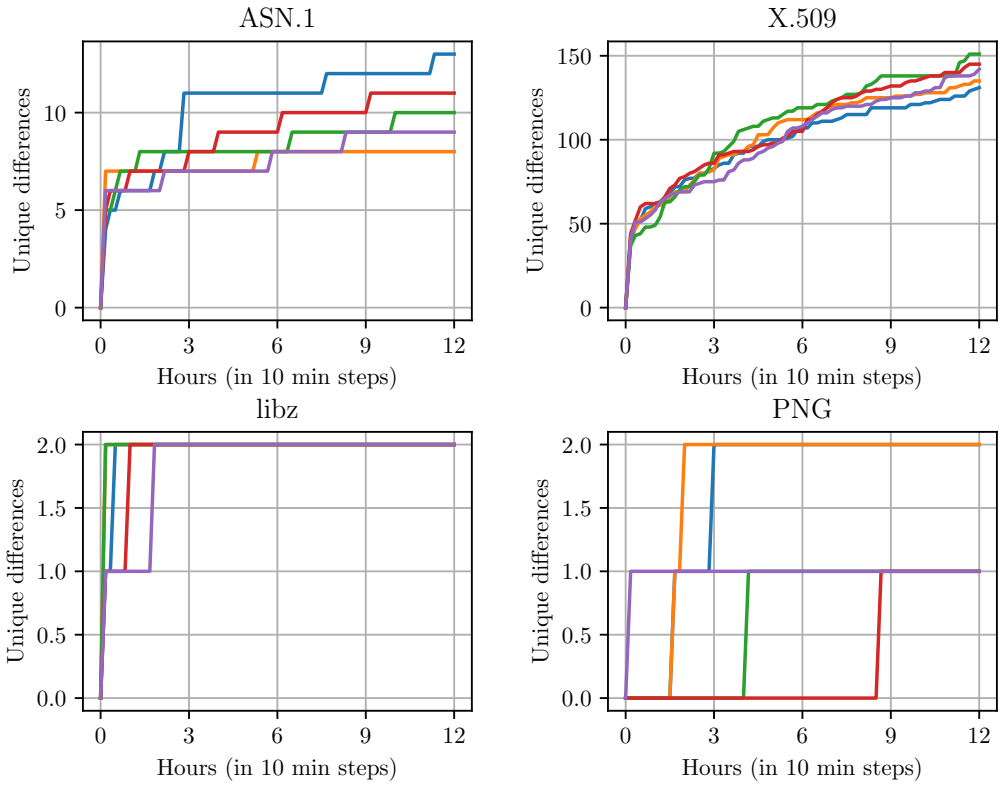

Figure 1. We evaluated each plugin for 12 hours running 5 times each, restarting every 10 minutes with 60 hours of fuzz time per target total. To ensure results are independent we reset state before each run, resetting corpora and crashers. Each line in this plot represents one such fuzzing run. Due to the random nature of fuzzing, each run can perform differently, and might not discover the same differences over 12 hours.

harness and running the fuzzing campaign, we did not find any discrepancy for which we could justify a security impact for the `libpng` case; as such, no results are reported for that target.

4.2.1. ASN.1. The ASN.1 parsers in `libcrypto` and `encoding/asn1` do not handle dates and strings the same way. To represent dates, the ASN.1 standard [9] has two tags: UTC and generalised. The two tags are associated with different string formats to represent time: where generalised time accepts four characters for the year and (optional) second fractions, UTC time only supports two digits to represent the year and an integer number of seconds. The fuzzer reveals that the Go parser will unmarshal generalised-encoded time as UTC and UTC-encoded time as generalised, without reporting an error. On the contrary, the `libcrypto` parser will correctly report an error in these cases. Furthermore, the Go parser will incorrectly report an error when second fractions are encoded as part of a generalised-encoded ASN.1 time string.

To represent strings, ASN.1 includes a number of different tags where the set of supported characters varies: for example, `NumericString` only supports spaces and digits, `PrintableString` includes the set of printable ASCII characters and so forth. The fuzzer reveals that in the Go parser mismatches occur between the expected field type supplied by way of annotations of structure fields (or through parameters directly supplied as function arguments) and the actual field type encoded in the ASN.1 tag byte. In contrast, `libcrypto` will report an error when an unmarshalling is requested as a type that conflicts with the one encoded in the tag byte of the input. As an example, it is possible to create a string that contains non-ASCII characters, encode them in an ASN.1 string payload tagged as `TagUTF8String` and have the Go parser parse it as a string of type `PrintableString`.

We argue here that this behaviour of the Go parser might have security implications. For example, in the case of ASN.1 strings, the behaviour of the Go parser might give developers a false sense of security, since they might assume that an ASN.1 string tagged as printable will only contain a subset of all printable ASCII characters. An adversary might exploit this assumption and mount an attack similar to punycode-based spoofing, where two ASN.1 encoded strings are visually similar, owing to the similarities of certain characters across alphabets, despite not being bytewise equal.

4.2.2. X.509. Our differential fuzzer highlights several differences between `crypto/x509` and `libcrypto`. All reported issues are caused by conflicts in the error model of the two libraries, namely, given an input, situations in which one of the two parsers returns successfully when

the other fails and returns an error. The analysis of the results reveals a set of false positives, stemming from the fact that full semantic similarity (as described in Section 3) was not achieved: for example, the Go parser performs several semantic validity checks (e.g. it will verify whether the modulus of an RSA public key has a length within an admissible range), whereas the `libcrypto` parser will defer such validation till when the RSA public key is used[4]. After filtering out the false positives, we report a few real cases of inconsistent error handling. One simple example is constituted by the way the two libraries handle trailing data: the `libcrypto` parser ignores trailing data whereas the Go parser always produces an error whenever the parser identifies the presence of trailing data beyond that which encodes a valid certificate. A more interesting case is represented by several inconsistencies in the way each implementation interprets the ASN.1-encoded bytes into the complex data structure defined for certificates in RFC 5280 [10]. We provide here an example that focuses on the way standard certificate extensions are encoded. In particular, we consider the Authority Key Identifier extension, used to identify the public key that signed this certificate. The standard requires that this field be encoded as an ASN.1 sequence of three optional octet strings (an identifier of the key, the issuer of the certificate, and its serial number). Our fuzzer reveals that the Go implementation abides by this requirement, whereas the `libcrypto` implementation will also accept a plain octet string for this field (notably, one not enclosed in an ASN.1 sequence) whenever only one of the three optional subfields defined in the standard is present.

In adversarial settings, these properties may be exploited in a number of ways. The inconsistencies in the way trailing bytes are handled represent a powerful primitive to build a covert channel between two malicious entities that use a system where `libcrypto` is employed to parse X.509 certificates. Another scenario of interest is represented by consensus-critical applications: this class of applications usually consists of a network of nodes running a program that processes transactions in a certain order. It is paramount that each of the nodes in the network process each of the transactions in the same way with identical side effect; otherwise the underlying network is said to have *forked*. Multiple implementations of the program might exist in different programming languages: it is for instance the case for Bitcoin and Ethereum, both consensus-critical applications, having widely-adopted implementations in different programming languages. Should X.509 parsing be required by these applications, it would be possible to exploit the inconsistencies described above to fork that network, achieving at least some form of denial of service, and at worst violating some correctness properties of the network.

X.509 stands out from other tested libraries in our work. In a relatively short amount of time operating on commodity hardware we found reproducibly over 125 unique mismatches (Figure 1). Due to the amount of differences found it appears there was no attempt made to match behaviour by maintainers using differential fuzzing.

4. The reader will notice that using a purported public key without validation might have disastrous consequences from a cryptographic standpoint, possibly leading to a full plaintext recovery attack.

4.2.3. zlib. The `libz` and `compress/zlib` decompressors are not consistent in the way they handle errors. One difference lies in the strictness with which each library accepts the input bytes constituting the `zlib` header. The `zlib` standard [8] expects a 2-byte header: the 4 LSB of the first byte encode the compression method, whereas its 4 MSB encode the compression information (the base-2 logarithm of the LZ77 window size, minus eight) – `0x78` is the value that is practically always used. The second byte encodes the compression level, whether a dictionary is present and a check field used for error detection. Our differential fuzzer reveals that the `compress/zlib` component will accept compression information that is not compliant with the constraints appearing in the standard, which states that values above 7 are not allowed in this version of the specification: indeed, we could verify that for the Go implementation – provided that the check field is computed correctly (that is, the two header bytes are congruent to zero modulo 31) – the compression information field may contain arbitrary data, since the actual value of the field does not impact the outcome of the decompression.

From a security perspective, this feature of the Go implementation may be used – for example – to build a covert channel, where a malicious sender transmits bits to a malicious receiver by encoding them in the compression information field, in a scenario where they are both users of a Go program that uses `compress/zlib` to decompress data. As another example, consider a scenario where an Intrusion Detection System scans malicious input, attempting to verify whether it may contain a zlib-compressed payload, and in that case flag it as malicious and drop it. The IDS uses `libz` for its parser, and as such, concludes that an input where the first byte of the header is not equal to `0x78` is invalid. Downstream the IDS, a Go application using the `compress/zlib` decompressor will instead successfully decompress the malicious input and process it further, thus successfully evading the IDS input check owing to the inconsistencies between the two compression libraries.

We disclosed this finding to the Go maintainers who created a patch to make the parser more strict and be in line with the specification. In detail, the patch checks whether zlibMaxWindow is larger than 7 and will emit an error if true. Details of the patch are available in the Go source repository https://go-review.googlesource.com/c/go/+/395734/.

5. Case Study

In this section we select a deviation in library behaviour reported by our differential fuzzer and construct a realistic scenario where an adversary is able to exploit them. This demonstrates that such deviations can have a security impact. We provide a second case concerning Certificate Transparency in the appendix, Section A.

5.1. Anti-Virus evasion

We consider the typical signature-based anti-virus (AV) usage scenario running on a host, where the AV scans files for known malware prior to the file being used. This AV could also run instead of files on the host, on incoming or outgoing emails, or backups. We demonstrate that some AVs can be evaded by making use of parse tree differences between the Go and `C` compression libraries.

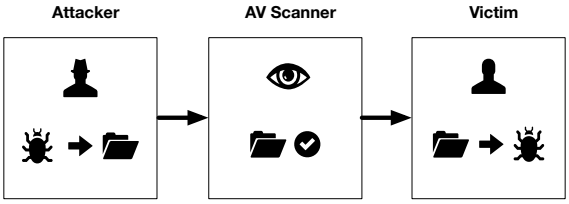

Figure 2. In this scenario, an attacker can evade an AV scanner by creating a `libz` file with corrupted header. In detail, the attacker compresses malware into a regular `libz` archive and then corrupts its headers by modifying two bytes. Next, the attacker transmits the corrupted file to their victim. When analysing the file, the AV scanner fails open as it cannot unpack the archive, due to using a `C` library. Victim unpacks the archive with software written in Go, delivering malware to the victim host. This evasion worked on 19 AV systems tested on Virus Total. We reported the vulnerability to Go maintainers and it was fixed promptly.

5.1.1. Background. A known way for attackers to bypass such AVs is via compression. If the AV does not attempt to decompress and scan incoming files, the AV may be bypassed. For these reasons, many AVs have incorporated known compression formats, and attempt to decompress potentially malicious content to scan the files therein. We focus here on the `zlib` [8] compression algorithm, because our fuzzer found parser differences in the handling of this format between the `C libz` library and the Go `compress/zlib` library for decompression.

5.1.2. Attack scenario and exploit. The precise scenario that we consider here is the following one. An AV uses the `C libz` library to attempt to decompress files, to scan them. The victim uses the Go `compress/zlib` library for decompressing the content prior to making use of it. The attacker's goal is to evade AV detection while ensuring that the victim successfully decompresses the content. The attack exploits header handling inconsistency described in Section 4.2.3. The AV is unable to decompress with the `C libz` library, and essentially *fails open*. That is, the AV can typically not afford to flag or quarantine files that are not recognized as the compression format or do not decompress properly, as that would lead to many false positives, which frustrates users. However, the Go library successfully recognizes the format and decompresses the content, exposing the user. The attack is depicted in Figure 2

5.1.3. Methodology and results. We use VirusTotal [20], a widely used AV aggregator that comprises 66 AV products as of our testing date (Feb 9th 2022). As malware, we use the EICAR test file [16], a test file specifically designed for testing AV products: 60 out of the 66 AV products on VirusTotal detect the EICAR test file as malware. We then prepare two files. The first is a standard `zlib` compressed EICAR test file, and is used to evaluate which AV products attempt to decompress `zlib` files to scan them (md5 hash: `914eec66a6a417f25c6a8188754872b1`). The second is the first file modified with the two byte header modification explained in Section 4.2.3, and is used to evaluate which AV products would be successfully evaded by the attacker (md5 hash: `8ba55a94dcb6863758e1f4619e5194c1`). Results show that 19 AVs attempt to decompress the file and successfully detect the first file, while none of the AVs are able to detect the second file. To summarize, our evasion strategy is 100% successful and is applicable to 19 AVs, including many commonly used ones.

As an additional datapoint for the practical relevance of the attack, we test both files with the AV scanner integrated in Gmail for outbound messages. Gmail successfully detects the first file and does not detect the second one, demonstrating the the evasion is also successful in this scenario.

5.1.4. Discussion. The prevention of this attack can be done in two ways. The first is to remark that the Go library is not behaving according to the specification, and should therefore be fixed. Accordingly, the developers of the `compress/libz` Go library have acknowledged this issue and quickly fixed it after our report. The other way is to update the `zlib` parser used by AV vendors to be able to decompress non-compliant payloads. The advantage of this approach is that, the attack would also be prevented in the absence of upgrades to the latest Go library release with the bugfix. If the specification had not specified the expected behaviour, this would likely have been the only option to mitigate the attack.

6. Related work

The areas of memory safety and fuzzing have received tremendous research interest in the past years. In this section we provide an overview of the body of work related to this paper to position our contributions.

6.1. Differential fuzzing

Differential fuzzing is used in many contexts prior to our work, however we are the first to use it to find parser differences in native Go and C versions of a library. JVMDiff [26], [25] uses differential fuzzing to find crashes and vulnerabilities in JVM implementations. DifFuzz [32] is a fuzzer for side channels that targets timing differences on the same program with different secrets. Frankencerts [23] performs differential testing of TLS implementations looking for disagreement between parsers. The authors crawled certificates and used them as seed input to generate mutated certificates to exercise uncommon paths in multiple TLS libraries. Since libraries should generally agree to interpret

input each library could function as an oracle, similarly to our work. Another work that uses differential fuzzing on libraries is NEZHA [34]. Rather than fuzzing for memory corruption bugs, this work focuses on semantic differences in parsers for ELF, XZ, PDF file formats, as well as X.509 certificates used in TLS. It uses the notion of δ-diversity for feedback to drive two types of input generation mechanisms targeting semantic differences. Cao et al. [24] find uninitialized kernel use vulnerabilities by using differential replay. Yang et al. [45] find bugs in code coverage tools via differential testing with random programs. DPIFuzz [36] finds elusion strategies for the QUIC protocol, to evade deep packet inspection systems.

6.2. Parse tree differential attacks

Numerous attacks exploit differences in parser behaviour in a variety of settings, a problem coined as parse tree differential attacks by Sassaman et al. [38], [37]. Kaminsky, Patterson, and Sassaman [30] exploit ASN.1 parser differences, specifically for X.509 Common Names and null terminators, allowing an attacker to claim a certificate for any site. Ptacek and Newsham [35] demonstrate that network intrusion detection systems can be evaded in this way. System-call-sequence based intrusion detection systems [28] can also be seen as parsers, and mimicry attacks [44] exploit parse tree differences. As another example, iOS implements multiple different ways to parse XML. Differences in XML parsing allowed for an exploit [14] that led to a sandbox escape.

6.3. Memory safety

Memory safety is a property of software, requiring that all memory accesses target *valid* pointers [41]. A number of protections and countermeasures have been developed and deployed over time to block specific attack vectors that relate to memory safety: for example, stack canaries target linear overflow of stack buffers, DEP/W$\oplus$X attempting to prevent the execution of attacker-injected code, ASLR to randomise the address space and increase the complexity of – for instance – the discovery of gadgets useful for an attacker, and CFI to prevent certain kinds of code-reuse attacks. Despite these protections, attacks related to memory corruption still exist and represent a serious threat [22].

Several new languages have been developed over the decades, from Java to Go and Rust, delivering native memory safety as integral part of the language. This is achieved with a combination of strong typing, compiler-inserted bounds checks (or inserted by the virtual machine in the case of Java) and various strategies to ensure temporal safety (by means of garbage collection or language enforcement – as is the case for Rust).

Go achieves memory safety by relying on strong typing, bounds checks added by the compiler and garbage collection. Developers are still able to write code that bypasses those protections. This takes place for instance when the `unsafe` package is used to perform typecasts that bypass the language's strong typing, or to perform C-style pointer arithmetic. Costa *et al.* have studied the prevalence of the `unsafe` package [27], finding that one in four popular repositories uses `unsafe`. Another scenario in which the memory-safety guarantees of Go are not guaranteed is represented by `cgo`, a mechanism to write C code as part of a Go project and call it at runtime. In this case, the compiler attempts to secure the C build by deploying ASLR, RELRO and stack canaries, which are otherwise absent from a Go binary.

A special case in terms of memory safety are mixed binaries, such as combining Rust and C/C++, or Go and C/C++. Recent work on security properties of such binaries [33], concluded that security guarantees achieved by one of the compilers can be overcome by attackers by exploiting the less secure code portion.

7. Discussion

Go is a widely used programming language that is often used in security critical applications. Popular repositories use a mix of native and non-Go libraries. Our analysis focuses on the way in which native and non-Go libraries differs, what the impact of such differences is and how to systematically find such disagreements. We discuss the main take-aways from our analysis.

7.1. Go or No Go: a difficult library choice

Generic statements such as "Go implementations are more secure than C/C++ ones" are likely to be preconceived ideas and require deeper analysis. We posit that developers face a difficult choice, at the very least from a security perspective, when opting for a Go-implemented library as opposed to a C/C++-implemented library. Memory safety guarantees are indeed better for Go native libraries, by language design. However, other issues, such as the subtle parsing differences we found, may negatively impact the security of Go native libraries. In particular, the maturity of the Go native libraries may not match that of the C/C++ libraries, leading to such issues.

7.2. Differential fuzzing of Go libraries

We found that libraries disagree on how to handle inputs, and that this can lead to security problems. While this is not surprising for the security community, our work is the first that practically demonstrates the point for the Go programming language. To address these shortcomings the Go community should invest more effort in finding differences in libraries that are security relevant. For instance, differential fuzzing of libraries could be performed as part of continuous integration/testing. Another avenue to make such testing easier is for go-fuzz to add `cgo` support [13]. Finally, appropriate staffing may be needed for triaging such issues: especially in the case of parsers, processing differences are expected, and in many cases stem from unclear

specifications in the relevant standards. Parser differences are neither unexpected nor unheard of, and development/debugging efforts should focus on differences that have an impact on correctness and security.

7.3. Security impact of parser differences

As demonstrated in our two case studies, small, seemingly inconsequential differences in library behaviour may lead to security issues in complex use cases that involve multiple components. These findings regarding the significant security impact of parser differences are consistent with previous work, and carry over multiple decades [30], [38], [35]. We posit that the fact that such vulnerabilities continue to exist may be due to the complex nature of the system and security model required for considering these parser differences as relevant to security and having them recognized as such, as well as the difficulty of finding them in the first place.

7.4. Benefits for C/C++ libraries

The primary focus of our work is how differential fuzzing may benefit Go developers, supporting their choice between Go and C/C++ libraries and aiding the discovery of inconsistencies between implementations. However, our design and implementation is immediately useable to support maintainers of C/C++ libraries in their task of i) improving compatibility of the code with the relevant standards and ii) finding memory safety violations. Our fuzzer is immediately useful for the first objective, since it will produce a list of inconsistencies whose root cause may well lie in the C/C++ implementation. As for the second objective, it would be interesting to explore how code coverage from a different implementation combined with coverage from the library may make a fuzzer more effective in finding inputs that leads to memory safety violations.

8. Conclusion

Go is a modern programming language that is widely popular with backend developers. Developers have the option of using Go-native libraries, or alternatively to use C/C++ libraries via `cgo`. Using non-native libraries in Go is popular, but choosing between the two options leads to security implications.

This is the first work to investigate security impact of the choice of native vs. non-native library use in Go. We present design and implementation of our differential fuzzer comparing these two types of libraries to find parsing mismatches that are security relevant. Namely, we analyse `libcrypto`, `libpng`, `libssl`, and `libz`. We find parser mismatches between native and non-native implementations in all of them, and argue that developers should take parser differences into account when choosing libraries for Go. Further, we describe security relevant impact on Certificate Transparency infrastructure, and evasion of 19 anti-virus products including Gmail scanning. Our work has led to a patch of Golang `libz` which has since been shipped.

Responsible Disclosure

All findings were disclosed to the affected parties. In particular we contacted Google with respect to the Go `compress/zlib` bug, and the developers quickly patched it. The corresponding fix can be found at https://go-review.googlesource.com/c/go/+/395734/.

Acknowledgements

The authors would like to thank Ange Albertini and Nigel Tao for handling our zlib disclosure and quickly providing a fix for Go, Ryan Sleevi for fruitful discussions on the Certificate Transparency use case, and anonymous reviewers and our shepherd for their comments on an earlier draft of this paper.

References

[1] Boringssl. https://github.com/google/boringssl.

[2] Cert spotter. https://github.com/SSLMate/certspotter.

[3] certificate-transparency: Auditing for tls certificates. https://github.com/google/certificate-transparency/.

[4] Certificate transparency ecosystem. https://sslmate.com/labs/ct_ecosystem/ecosystem.html.

[5] Certificate transparency: Go code. https://github.com/google/certificate-transparency-go.

[6] mkcert. https://github.com/FiloSottile/mkcert/.

[7] Network security services. https://github.com/nss-dev/nss.

[8] ZLIB Compressed Data Format Specification version 3.3. RFC 1950, 1996.

[9] Specification of Abstract Syntax Notation One (ASN.1). X.208, 1998.

[10] Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, 2008.

[11] Certificate Transparency. RFC 6962, 2013.

[12] Attack and Threat Model for Certificate Transparency. https://datatracker.ietf.org/doc/html/draft-ietf-trans-threat-analysis-16, 2018.

[13] https://github.com/dvyukov/go-fuzz/issues/171, 2019.

[14] Psychic paper. https://blog.siguza.net/psychicpaper/, 2020.

[15] https://go.dev/security/fuzz/, 2022.

[16] Anti-Malware Testfile Download. https://www.eicar.org/?page_id=3950, 2022.

[17] CA/Incident Dashboard. https://wiki.mozilla.org/CA/Incident_Dashboard, 2022.

[18] Certificate Transparency. https://developer.mozilla.org/en-US/docs/Web/Security/Certificate_Transparency, 2022.

[19] Transparent Logging: A Guide – admission control. https://github.com/google/trillian/blob/master/docs/TransparentLogging.md#admission-control, 2022.

[20] VirusTotal. https://www.virustotal.com, 2022.

[21] Andrew Ayer. WGLC started for draft-ietf-trans-threat-analysis. https://mailarchive.ietf.org/arch/msg/trans/IijSa8IPZ0oZ9fr1xmq-ETMjnss/, 2018.

[22] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 747–758, 2020.

[23] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *2014 IEEE Symposium on Security and Privacy*, pages 114–129. IEEE, 2014.

[24] Mengchen Cao, Xiantong Hou, Tao Wang, Hunter Qu, Yajin Zhou, Xiaolong Bai, and Fuwei Wang. Different is good: Detecting the use of uninitialized variables through differential replay. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1883–1897, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of jvm implementations. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1257–1268. IEEE Press, 2019.

[26] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. *SIGPLAN Not.*, 51(6):85–99, June 2016.

[27] Diego Elias Costa, Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. Breaking type-safety in go: An empirical study on the usage of the unsafe package. *IEEE Transactions on Software Engineering*, 2021.

[28] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security*, 6(3):151–180, 1998.

[29] John E. Hopcroft. On the equivalence and containment problems for context-free languages. *Mathematical systems theory*, 3(2):119–124, 1969.

[30] Dan Kaminsky, Meredith L Patterson, and Len Sassaman. Pki layer cake: New collision attacks against the global x. 509 infrastructure. In *International Conference on Financial Cryptography and Data Security*, pages 289–303. Springer, 2010.

[31] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[32] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. Diffuzz: Differential fuzzing for side-channel analysis. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 176–187. IEEE Press, 2019.

[33] Michalis Papaevripides and Elias Athanasopoulos. Exploiting mixed binaries. *ACM Transactions on Privacy and Security (TOPS)*, 24(2):1–29, 2021.

[34] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632, 2017.

[35] Thomas H Ptacek and Timothy N Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks inc., Calgary Alberta, 1998.

[36] Gaganjeet Singh Reen and Christian Rossow. Dpifuzz: A differential fuzzing framework to detect dpi elusion strategies for quic. In *Annual Computer Security Applications Conference*, ACSAC '20, pages 332–344, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Len Sassaman, Meredith L Patterson, Sergey Bratus, and Michael E Locasto. Security applications of formal language theory. *IEEE Systems Journal*, 7(3):489–500, 2013.

[38] Len Sassaman, Meredith L Patterson, Sergey Bratus, and Anna Shubina. The halting problems of network stack insecurity. *USENIX; login*, 36(6):22–32, 2011.

[39] Ryan Sleevi. WGLC started for draft-ietf-trans-threat-analysis. https://mailarchive.ietf.org/arch/msg/trans/rnoaefCotNj_sNHLDJT29J7KIj0, 2018.

[40] SSLMate. How Cert Spotter Parses 255 Million Certificates. https://sslmate.com/blog/post/how_certspotter_parses_255_million_certificates, 2017.

[41] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[42] Dmitry Vyukov. go-fuzz-corpus. https://github.com/dvyukov/go-fuzz-corpus, 2021.

[43] Dmitry Vyukov. go-fuzz: randomized testing for go. https://github.com/dvyukov/go-fuzz, 2021.

[44] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[45] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 488–499. IEEE Press, 2019.

[46] Micha Zalewski. american fuzzy lop. https://github.com/google/AFL, 2021.

Appendix A. Circumventing Certificate Transparency

We describe here a scenario where the inconsistent treatment of X.509 certificates and ASN.1 marshalled data between the Go and `libssl` implementations might be exploited by a malicious (or compromised) certificate authority to bypass the guarantees provided by the *Certificate Transparency* (CT) framework.

A.1. A CT primer

Certificate Transparency [11] is a framework originally proposed by Google to tackle some of the shortcomings of the PKI infrastructure. Specifically, the problem addressed by CT is to ensure that when a user visits a website with TLS, the server certificate indeed belongs to the owner of that website, and has not been issued by a compromised or simply sloppy certificate authority to a malicious third party. Without this guarantee, third parties in possession of rogue server certificates might be able to intercept the traffic that was meant to remain confidential between user and intended target website.

The CT ecosystem consists of legacy actors and new entities. The legacy actors in this ecosystem are end-users browsing the web, website owners and certification authorities. End-users use a possibly insecure network to connect to websites whose names they know. The DNS resolution of those names might point them to rogue endpoints. However, by requiring a secure channel (e.g. TLS) and the presentation of a certificate issued by a certification authority they trust, end-users (through their browsers) are able to verify that the name of the website they wanted to browse and the name in the presented server certificate match. If so, they are certain that they have connected to the intended endpoint.

Unfortunately, the proliferation of certificate authorities implies a wider attack surface, given that attackers have higher chances to compromise (or collude with) at least one CA, at which point the previously mentioned guarantees are entirely voided. This is where CT and its new entities step in. In order to thwart the aforementioned attack scenario, CT requires that all issued certificates be publicly auditable. This achieves two important goals: on the one hand, website owners can audit all newly issued certificate to discover whether any certificates were issued matching features (e.g. DNS name) on which they claim ownership without their consent. Public auditability thus enables prompt discovery of these attack scenarios and gives websites owners a chance to request revocation of these rogue certificates. On the other hand, by requiring that only certificates carrying a proof that they have been publicly audited be used by browsers, end users have the assurance about the integrity and correct origin of the certificate they use in their TLS session.

Public auditability is guaranteed by *CT log servers*, the entities in the CT ecosystem tasked with maintaining a public, highly available, authenticated and append-only log of all issued certificates. Website or domain owners *monitor* the logs, matching newly issued certificate against interesting features (e.g. DNS name of the `Subject` matching the own domain name) to ensure prompt revocation of all rogue certificates. Proof of inclusion in a log is achieved by way of the Signed Certificate Timestamp (SCT), a signature over the certificate by the log server that gets appended to a publicly visible log. Modern browsers such as Chrome and Safari [18] will reject certificates that lack such proof of inclusion; the latter can be embedded in the certificate or exchanged in the course of the TLS handshake.

All actors in the CT ecosystem rely – to different degrees and for different reasons – on the ability to parse X.509 artefacts, which often make use of ASN.1 encoding: The user-agent parses the purported certificate to conduct the TLS exchange and to establish a set of security properties of the end-point, among which the match between the requested domain and the certified one, the expiration/revocation status of the certificate and whether the certificate is included in sufficiently many CT log servers. The log server parses the incoming certificates and must establish a valid certificate chain from the submitted leaf to one of the trusted roots. Beyond that, log servers can verify the overall well-formedness of the certificates, but according to the letter [11] and spirit [19] of the standard, they are not tasked with performing any other verification and may accept improperly structured objects: after all, presence of a certificate in the log only means that the certificate is bound to get the broadest possible scrutiny. Finally, log monitors must parse all certificates to detect attacks such as the issuance (to a malicious third party) of a certificate for a domain that the monitor owns. For this reason, the monitor must use a parser that is not too strict: a strict parser might dismiss a malicious certificate which may be accepted by a user-agent with a more lax parser. Next, we investigate this scenario and how the inconsistent behaviour of parsers revealed by our differential fuzzer play a role.

A.2. Monitor evasion scenario

We consider a scenario where a malicious entity compromises (or colludes with) a certificate authority, and succeeds in obtaining a certificate that refers to a domain D_{target} the attacker does not own: in particular, the `Subject` of this certificate identifies an entity (and a DNS domain) over which the attacker has no control. We also assume that the certificate authority is one of the root CAs that is trusted by the end user's browser. The end-user uses the browser to access resources at D_{target}. We assume the attacker controls the network and can – for instance – cause the name resolution for D_{target} to return an IP address the attacker controls. We assume the user's browser supports CT and will only allow connections to servers that present certificates with an embedded SCT: this is the case for commonly used browsers today. We consider here a simplified CT ecosystem with a single log server and a single scanner acting on behalf of the rightful owner of D_{target}. We discuss a more general scenario with multiple servers and scanners below.

The scenario proceeds with the following steps:

1) The CA generates a certificate C for domain D_{target} for the attacker. The certificate is generated in such a way that all browsers will consider it well-formed, whereas the CT monitor will consider it unparsable and as such, will fail to flag it as a certificate for D_{target}.
2) To obtain proof of inclusion in the form of an SCT, the CA (or the attacker) submits the pre-certificate version of C (identical to C except for the poisoning extension) and obtains the SCT. The latter is then embedded in C.
3) Inclusion in the log prompts the monitor to download the certificate and attempt to match it against features of the monitored domain D_{target} (e.g. D_{target}'s DNS name). However, due to its unparseability, the monitor skips it and – crucially - does not register the fact that it refers to domain D_{target}.
4) The user connects to domain D_{target}. DNS resolution points them to an IP address controlled by the attacker, and certificate C is presented as the server certificate as part of the TLS handshake. The user's browser parses C with no issue and accepts it as valid given that it embeds an SCT from the CT log server.

To successfully evade monitoring, a certificate needs to enjoy the three following properties: it should i) be successfully parseable by the parser in the browser; ii) be accepted in the CT log; and iii) it should be flagged as malformed by the parser of the scanner. We show next how our differential fuzzer could be adapted to find such encoding property.

A.3. Fuzzer-aided search for parser deviations

Section 4.2.2 shows how the differential fuzzer could find differences in the way Go and `libssl` parse X.509 certificates. The scenario is applicable since most browsers use C/C++ libraries to parse certificates (Firefox uses NSS [7], Chrome uses BoringSSL [1]) whereas most CT

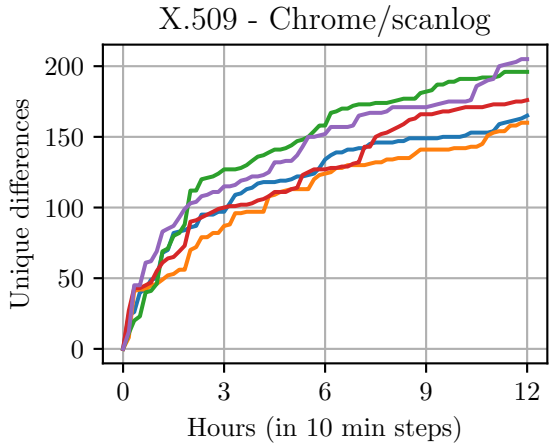

Figure 3. We fuzz a specialized target to find certificates that fulfil the special constraints of the monitor evasion scenario, namely, that they be accepted by the Chrome parser and rejected by the scanlog parser. We ran our experiments 5 times for 12 hours each, with 60 hours total fuzz time. Each line in this plot represents one independent fuzzing run for which we reset corpora and crashers. Since each run can differ, we perform this experiment multiple times to test whether results are comparable.

monitors are projects based on Go [5], [2]. The results from section 4.2.2 are however not directly applicable for two reasons: on the one hand, the Chrome parser applies a multi-tiered parsing, and as such, certificates that seem to be accepted by the `libssl` parser (BoringSSL is an OpenSSL clone) are not accepted by Chrome. On the other hand, the Go CT monitor code (scanlog [5]) relies on local forks of the relevant parser packages (ASN.1, X.509, pkix).

To adapt to this situation, we modify the fuzzing setup of Section 4.2.2 in the following way: we use the parsing function from the Go CT monitor code (scanlog [5]) instead of the runtime's parsing function. We also replace the invocation of the certificate parser from `libssl` with a subset of the parser code from Chrome. Chrome appears to have a multi-level parsing of certificates: a certificate is first parsed by the BoringSSL layer to establish the TLS session; it is then further parsed by the parsing code in the `net/cert` package[5] and to be accepted, a certificate must be successfully parsed by both layers.

We evaluate our fuzzer against this custom target with the same methodology applied in Section 4.1: we execute 5 times for 12 hours restarting in 10 minute intervals, resetting corpus and crashes for each run, with a total fuzz time of 60 hours. Each fuzz run finds over 150 unique error responses, see Figure 3.

After modifying the fuzzer configuration, we are able to identify several differences between the two parsers that are interesting in the context of our considered threat model. We verify that one in particular can be used to recreate the scenario described above. This difference relates to

5. Relevant functions are `x509_util::CreateX509Certificate-FromBuffers` and `ParsedCertificate::Create`.

TABLE 3. BEHAVIOUR OF VARIOUS PARSERS WHEN PARSING MALFORMED CERTIFICATES. THE FIRST COLUMN IDENTIFIES AN X.509 EXTENSION. FOR EACH OF THE TESTED CERTIFICATES, THE VALUE OF EACH EXTENSION CONTAINS NON-ASCII UTF-8 CHARACTERS. THE SECOND COLUMN DETERMINES WHETHER THE VALUE OF THE EXTENSION IS ENCODED AS AN ASN.1 `NumericString` (IDENTIFIED AS NUMERIC IN THE TABLE) OR AS AN ASN.1 `PrintableString` (IDENTIFIED AS PRINTABLE IN THE TABLE). ✓✓CORRESPONDS TO THE PARSER ACCEPTING THE INPUT. ✓CORRESPONDS TO THE PARSER ACCEPTING THE INPUT BUT RETURNING A WARNING, WHEREAS ✗CORRESPONDS TO THE PARSER REJECTING THE INPUT.

		Firefox	*Chrome*	*scanlog*
C	numeric	✓✓	✗	✗
	printable	✓✓	✗	✓
ST	numeric	✓✓	✗	✗
	printable	✓✓	✗	✓
L	numeric	✓✓	✗	✗
	printable	✓✓	✗	✓
street	numeric	✓✓	✗	✗
	printable	✓✓	✗	✓
postalCode	numeric	✓✓	✓✓	✗
	printable	✓✓	✗	✓
O	numeric	✓✓	✗	✗
	printable	✓✓	✗	✓
OU	numeric	✓✓	✗	✗
	printable	✓✓	✗	✓
CN	numeric	✓✓	✗	✗
	printable	✓✓	✗	✓
emailAddress	numeric	✓✓	✓✓	✗
	printable	✓✓	✗	✓

an overall inconsistent handling of strings containing non-ASCII characters encoded as either `NumericString` or `PrintableString`. This constitutes a violation of the ASN.1 standard since both string types must contain only a subset of ASCII symbols. In Table 3 we show the results of parsing certificates that contain these encoding violation in a selection of fields of the certificate `Subject`. The table shows that Firefox will accept all such certificates, scanlog (which is used as a CT monitor) will reject all cases where the value is encoded as a `NumericString`, but it will accept (with a warning) all cases where the value is encoded as a `PrintableString`, and Chrome will reject all certificates except for those where the `postalCode` (OID 2.5.4.17) or `emailAddress` (OID 1.2.840.113549.1.9.1) extension is encoded as a `NumericString`. As a result we obtain a certificate that is accepted by user agents and yet not detectable by a monitor.

A.4. Prototype

After we identified a suitable parser deviation, we proceed to prototype the scenario. Figure 4 describes how the four steps that we identified above can be tested in practice. In particular:

1) We use the popular `mkcert` utility [6] to generate a correctly formed root CA certificate. We then modify the code of the tool to introduce the necessary encoding deviation when generating a leaf certificate whose Sub-

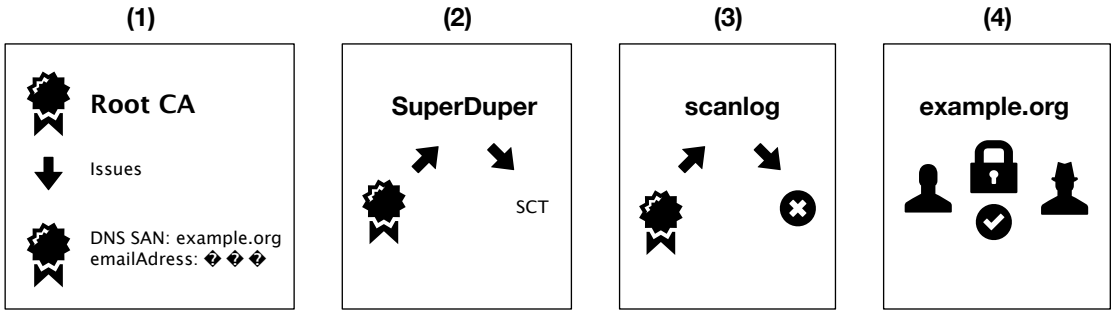

Figure 4. The four steps of the monitor evasion prototype. First, we set up a new trusted root CA and use it to issue a certificate for `example.org`, with a corrupted emailAdress field. Second, we configure the SuperDuper CT log to trust our CA and submit the previously generated certificate to retrieve an SCT. Third, we attempt to monitor the log with scanlog, a process that the owner of `example.org` might perform. Scanlog cannot parse the corrupted emailAdress field and will not detect the issuance of a certificate for `example.org`. Finally, when performing a network attack, the corrupted cert can be used to impersonate `example.org` with a certificate that possesses an SCT and is accepted by major browser vendors.

ject Alternative Name points to the `example.org` domain. In particular, we introduce the `emailAddress` extension in the `ExtraNames` field of the `Subject`. We then set the value of the extension to contain non-ASCII characters. We use a local fork of the ASN.1 and X.509 packages to force the `NumericString` tag for the value, despite it containing symbols other than digits.

2) We spin up a vanilla instance of the `SuperDuper` [3] CT log server[6]. We configure it to trust the root CA certificate generated in the previous step. We then submit the certificate chain (root and leaf) generated in the previous step. The certificate is accepted in the log and an SCT is returned to the caller.
3) We monitor the log after insertion of the certificate with the `scanlog` tool [5]. This tool is built in Go and uses the parser whose behaviour we analysed with the fuzzer. In accordance with the results produced by the fuzzer, the parser fails to parse the certificate. The tool is designed to just log parsing errors and move on to the next log entry. The tool thus fails to observe the distinguished name encoded in the certificate and match it against the search terms specifying `example.org`. This emulates the inability of the CT monitor to detect issuance of a certificate with features under observation.
4) We set up a Go TLS server that uses the certificate generated in step 1 as the server certificate. The server process needs to operate on local forks of the Go TLS (and related) packages in order to accept the improperly-encoded certificate as the server certificate. We then simulate control of the network by overriding name lookups for `example.org` and return the IP address of the TLS server. We then test Safari (version 15.3 (17612.4.9.1.5)) on MacOS 12.2 and Chrome (version 99.0.4844.51), Firefox (version 98.0) on Ubuntu 18.04.6 LTS, by adding the root CA certificate as one of the trusted roots and by then attempting to browse `example.org`. We confirm successful browsing of the rogue `example.org` without any certificate warning in all three cases.

6. We discuss other servers in the next section

A.5. Discussion

Parsers and the CT ecosystem The main issue with this scenario lies in the security implications of inconsistent parsing. Such inconsistencies however must be expected for a standard as complex and as rich of de-facto accepted variations as the X.509 one. From a security perspective, the two relevant sets of parsers are those running in the browsers and those running at the monitors. Ideally those two sets would interpret certificates identically. Alternatively, monitor parsers should be laxer so that they might (unnecessarily) accept and analyse certificates which browsers would anyway refuse to process. Achieving equivalent parsing between browsers and monitors is not simple, given the high number of different implementations of both, and the different programming languages and dependencies that are used when building either. The set of parsers used by the most popular browsers might be used to augment the parser of monitors, in an attempt to present to the monitor entity the same "view" that a user-agent is able to construct, and reason about the artefacts generated by the consuming entity. We tested another monitor, SSLmate's `certspotter` [2] which successfully parsed the certificate and was able to flag it as referencing `example.org` despite the unparseable extension. This speaks in favour of minimalistic parsing on the part of the monitor code [40] – accepting as wide a set of certificates as possible in order to catch as many rogue certificates as possible, even if browsers wouldn't be able to parse them.

Multiple CT log servers and different implementations thereof The `SuperDuper` [3] CT log server that we use in our prototype is deprecated in favour of its Go version [5] based on `Trillian`. However: i) SuperDuper is still in use by production servers [4]; ii) the core issue behind the vulnerability is not that the CT log server is too accepting, rather, that CT monitor and browsers do not

have a consistent view of what a well-formed certificate is; furthermore iii) CT log servers are not meant to be policing entities since this role is reserved to monitors. Indeed CT log servers are simply tasked with the creation of an append-only, highly available log of certificates. The syntactic and semantic inspection of this log is delegated to monitors. In our prototype we considered a single server. Having multiple servers alone would not solve the issue: as long as a critical mass of servers accepts certificate C, the latter would obtain an SCT. At this point, its ability to evade detection of the rightful domain owner is not affected by the number of available log servers, rather, by the ability of its monitor's parser to observe its existence in at least one log.

Mitigating factors The CT RFC [11] does not clearly specify what monitors should do beyond watching certificates of interest. In particular, it does not specify how monitors should handle certificates they are unable to parse. An RFC draft that analyses the threat model of Certificate Transparency [12] discusses in Section 4.2 our threat model as part of the analysis of syntactic mis-issuance with a malicious CA context. However the authors do not cover our exact scenario, namely, the case in which mis-issuance is intentional as part of an attempt to evade scrutiny of a monitor (syntactic mis-issuance to evade discovery of semantic mis-issuance). Fortunately the way in which the CT ecosystem operates in practice[7] leads to parser failures by monitors being flagged and manually investigated in public issue trackers such as the bugzilla CA bugs [17]. This investigation carries the threat for the issuing CA of de-listing from the set of trusted CAs. The combination of manual investigation by the community and the negative incentive to issue such certificates to begin with represents a powerful mitigation for this scenario.

7. We discovered these real-world operational details by interacting with prominent operators in the CT ecosystem and by reading mail threads [21], [39] related to the RFC draft.