Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MASTER THESIS

Security Evaluation and Vulnerability Assessment of FlatBuffers

An Analysis of the inherent vulnerabilities and risks associated with the FlatBuffers encoding format

MIN Faculty
Department of Informatics
at Research Group Computer Networks (NET)

Lilly Sell

li@lly.sh or bav3001@studium.uni-hamburg.de
M.Sc. Informatik
Student ID Number: 7047081

First Reviewer: Prof. Dr.-Ing. Mathias Fischer
Second Reviewer: Prof. Dr. Janick Edinger
Supervisor: August See

Abstract

FlatBuffers are a popular encoding format in distributed systems of which rely on performance and lightweight encoding/decoding of messages. This is achieved by FlatBuffers through encoding rules that closely match the execution and memory model of the underlying hardware, however, a component as central as a communications protocol must actively incorporate security mechanisms in its design too to prevent the whole system from collapsing. To the contrary, the FlatBuffers project facilitates insecure coding practices caused by the underlying format constituting a number of attack vectors which may lead to a systems degradation and even full collapse. While protocol security is a well-established area of research, the FlatBuffers format has not thoroughly been investigated. This work investigates different attack vectors permitted by the format and systematically compares their impact on applications using FlatBuffers in different programming languages and configurations. It is shown that while FlatBuffers is an efficient protocol, it should be used with care and expertise. Additionally, guidelines for improving FlatBuffers safety are included.

Acknowledgements

Many thanks to all my friends and family who supported me through these stressful months. I especially want to mention Benedikt Ostendorf for his valuable insights and without whom this whole would have never been started.

Also, I acknowledge that ChatGPT has been used for drafts.

Contents

List of Figures

1 Introduction

In the contemporary digital landscape, computers and devices are interconnected through vast and intricate networks, enabling numerous applications and services to interoperate seamlessly. The efficiency and security of data communication in such networks are paramount, and protocols for data serialization play a critical role in this context. An encoding format determines how a piece of information is translated into a set of bytes as well as how a set of bytes is translated back into usable information. Figure 1 presents a binary value whose information cannot be extracted without any additional context. Depending on the used encoding, the value can have different meanings and it is precisely the encoding format that dictates which interpretation is the correct one.

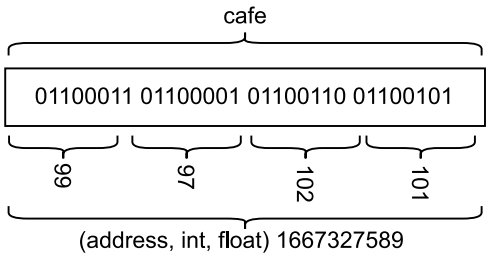


Figure 1: A binary value which among others can be interpreted either as the string "cafe", one 64bit integer, or four 8-bit integers

Many different encoding formats have been developed, each with their own design choices, resulting in very different strengths and weaknesses. For example, in web applications, JavaScript Object Notation (JSON) enjoys great popularity for its simplicity and great support built into all web browsers. In other contexts, with different requirements and use-cases, different formats are used instead. One such format is FlatBuffers[1]. It is a serialization protocol developed by Google, that is designed to provide efficient encoding and chiefly decoding of arbitrary data with minimal overhead. FlatBuffers achieves this through explicit encoding rules that are as close as possible to the requirements of most general-purpose hardware and by utilizing predefined schemas that specify the structure of the data being transmitted. If that structure is known in advance, it does not need to be included in encoded values, which reduces a lot of overhead. FlatBuffers is advertised as being especially useful in computer- or mobile games as well as performance-critical data processing applications, where it has indeed gained popularity.

Since encoding protocols play such a central role in interconnected systems and directly dictate how a computer interprets a set of bytes, their design and implementations are often scrutinized extensively and in great detail. Protocol Buffers (ProtoBuf), JSON, XML, and other well-established formats have undergone extensive work to achieve this. However, a comprehensive security evaluation of FlatBuffers remains notably absent in existing literature, leaving a critical gap that this thesis aims to fill. This thesis

contains an in-depth investigation of the FlatBuffers protocol, focusing on two areas: Analyzing inherent flaws within the protocol that could be exploited in various scenarios and evaluating the impact this has on real world applications. The necessity for such research becomes clear when compared to already scrutinized protocols like ProtoBuf, which have undergone similar detailed security analyzes.

The main contributions of this thesis are the in-depth analysis of the security properties of the FlatBuffers encoding format as well as the reference implementations security properties. Different attacks and exploits are demonstrated to be extremely effective and cause severe service degradation in applications that exhibit common FlatBuffers usage patterns, even when those applications follow all best practices and utilize every available security mechanism. Furthermore, a systematic comparison of those results as well as recommendations for securely handling FlatBuffers have been created.

The structure of this thesis begins with foundational concepts and theory, moves through the practical implementation of measurement and analysis tools, and culminates in the evaluation and discussion of the findings.

- Section 2 lays out the essential concepts necessary for understanding the subsequent research. It includes a general definition and explanation of schemas in data serialization and communication protocols and what they are used for. Additionally, an overview of code generation methods and their application in FlatBuffers is given, followed by a detailed discussion of memory layout considerations, on top of which the FlatBuffers protocol is built. Finally, a small demonstration of the basic usage of FlatBuffers is given.
- Section 3 contains a comprehensive review of existing research and methodologies relating to schema reverse-engineering, DOS vectors and attacks, the design of secure communication protocols and discoveries relating to securely designing and documenting an Application Programming Interface (API). Moreover this section describes the exact scope and requirements within which this research was conducted including an exact formulation of the resolved research questions.
- Section 4 is the first section containing results including an analysis of the validation processes implemented by FlatBuffers. It also contains details about the attack vectors inherent to FlatBuffers protocol design, which includes detailed examinations of specific attacks.
- Section 5 then presents the other major results and goes into detail on security mechanisms and general implementation quality of the different programming languages that *flatc* supports.
- Afterwards, Section 6 ties everything together, drawing conclusions about the effectiveness and implications of the methods employed, and discusses the implications of those results. It also contains guidelines for decreasing and mitigating the impact of the discovered vulnerabilities.

- Section 7 and Section 8 then end by acknowledging the constraints and limitations encountered during the research and drawing a summary of the main findings, contributions, and potential directions for future research.

2 Background

This section explains foundational concepts required to understand the specific topics and work described in this thesis. It provides basic introductions to the techniques used in the development of consistent distributed systems which are also utilized by FlatBuffers. Additionally, a foundational explanation of memory layout requirements that dictate the design of the FlatBuffers encoding format is provided. To ensure proper understanding of the described concepts, the use of FlatBuffers is demonstrated in a simple example.

2.1 Schemafull Communication

Realizing communication within a distributed system can be done using a predefined and strict messaging schema which defines a consistent and structured format for the messages that different parts of the system use to exchange data. The schema dictates the exact layout, type, and constraints of the data that can be sent or received. Common formats for these schemas include JSON Schema[2], ProtoBuf[3], and the XML Schema Definition Language (XSD)[4].

Key concepts in message-handling involve several steps to ensure smooth communication within a distributed system. First, a *Message Definition* is established by creating a schema using a specific schema-definition-language. This schema defines the structure and types of all possible messages, and in some cases also other information. For example, Remote-Procedure-Calls can be defined in *gRPC*[5] or *XML-RPC*[6]. At runtime, messages then undergo *Serialization*, where they are converted into a format suitable for transmission, typically binary or text-based formats. Upon receipt, these messages go through *Deserialization*, where they are reverted to their original form as defined by the schema. To ensure data integrity and conformance, *Validation* against the schema should be performed on both incoming and outgoing messages against the schema.

Adopting a predefined schema brings numerous advantages. It ensures consistency and predictability across the distributed system, with all components knowing the exact message format, thereby reducing errors and misinterpretations. Type Safety is also maintained, preventing type-related runtime errors. Another benefit is that the schema acts as a contract between services, allowing proper version management and clear documentation of the represented interface. Interoperability can also be enhanced since different programming languages or platforms can communicate using a common schema instead of relying on their own internal data representation. Automatic validation can ensure the messages' integrity and protect the robustness of the system.

However, there are drawbacks to this approach. The rigidity of strict schemas can be a bottleneck if the system needs to evolve quickly, as any data structure changes require schema updates and may affect all services using it. This can introduce overhead, as maintaining and validating schemas can be resource-intensive both in development and runtime. Complexity in evolution arises when introducing new fields or modifying existing ones, necessitating a thoughtful versioning strategy to maintain backward

compatibility. However, some schema systems support schema development that are compatible to older schema versions. This is the case when using FlatBuffers for many types of schema changes. The initial setup can be time-consuming, as it entails implementing or integrating schemas and associated tooling into the development and deployment workflow. Furthermore, performance can be be impacted, with serialization, validation and deserialization potentially adding latency and computational overhead, particularly for complex schemas.

Example use cases for predefined schemas include microservices, where reliable, documented, and predictable communication between different services is essential. They are also employed in public APIs to define the structure of requests and responses and in inter-service communication within environments where components are written in different programming languages.

In summary, using a predefined and strict messaging schema introduces structure and reliability to distributed systems, albeit at the expense of flexibility and potential overhead. It is crucial to weigh these trade-offs against the specific requirements and constraints of the system being developed.

2.2 Code Generation

Code generation is the process in which tools automatically generate source code based on predefined schemas or templates. It is often used when using message schemas for communication to produce the boilerplate code necessary for serialization, deserialization and validation of messages based on the schema. By leveraging code generation, developers can streamline the implementation of schemafull communication, ensuring that their distributed systems remain consistent, reliable, and easier to maintain. However, they must also navigate the complexities and constraints introduced by this automated approach.

A typical workflow when using a schema in conjunction with code generators involves the following steps. First, a schema is defined using a format like ProtoBuf or JSON Schema. This schema specifies the structure, data types, and constraints for the messages. Next, specialized tools, such as `protoc` for Protocol Buffers or `jsonschema2pojo` for JSON Schema read the schema files and generate the corresponding source code in the desired programming languages. The generated code typically includes classes or data structures that map to the ones defined in the schema, along with methods for serializing, deserializing and validating these structures. Developers then integrate this generated source code into their applications, allowing them to easily and consistently handle messages as defined by the schema[1]–[3].

There are several advantages to this approach. Automatically generated code is consistent with the schema, reducing the risk of discrepancies and ensuring that all parts of the system adhere to the same format. It also eliminates the need for developers to manually write boilerplate code, saving time and reducing the likelihood of human error. Generated code often includes type-safe structures and methods, which help

prevent type-related runtime errors. Additionally, changes to the schema can be systematically reflected in the codebase by regenerating the code, simplifying updates and maintenance. Building distributed systems that span different technological stacks can often also be simplified when the code generator supports different programming languages and technologies.

However, this method is not without disadvantages. The development process becomes dependent on the code generation tools, which may have limited capabilities or usability. In certain cases, these tools may introduce bugs into any part of the system that handles messages.

Introducing code generation into the build process is also often nontrivial as not all build tools are readily equipped to handle it which may require additional configuration as well as integration efforts. Generated code can be less flexible, as it adheres strictly to the predefined schema or the code generator lacks customization options. Any customization or tweaking may require manual modifications, which can be cumbersome and complicate the build process even further[7]. Developers also need to familiarize themselves with the schema-definition-language and the code generation tools that are now part of their technology stack, which could necessitate a learning period. Moreover, generated code may include additional overhead or inefficiencies that developers would optimize out if writing the code manually.

2.3 Memory Layout of Data Structures

The term memory layout refers to the organization and arrangement of data in a computer's memory. Discussing memory layout involves understanding how the data structure's components (such as the elements in an array or the fields in a struct) are stored in memory.

Good memory layout is important for performance optimization, memory usage, concurrency and parallel processing. For performance optimization, cache efficiency is improved when related data are stored contiguously, increasing the likelihood of cache hits and thereby reducing latency. Additionally, algorithms perform better when data access patterns align with the memory layout, such as sequential versus random access. The amount of memory required also relates to the way that memory is organized. Aligning data or introducing padding ensures that data elements are correctly positioned, leading to efficient access patterns despite potential memory waste. Tightly packed memory layouts on the other hand can save space, which is beneficial for systems with limited memory but only possible for those that don't depend on alignment[8]. Proper memory layout can also minimize conflicts when multiple threads access data structures in parallel.

However, restrictions are commonly present on modern systems. Data types typically have specific alignment requirements, with compilers often inserting padding between structure members to ensure correct alignment, possibly resulting in wasted memory. If such alignment requirements are not met and a processor is asked to work with

improperly aligned memory, some hardware may complete the task with performance degradation, while other hardware may completely reuse the instructed operation[9]. The order in which bytes are stored, known as endianness, can also differ between systems, affecting data interpretation when moved between systems with different endianness.

Consider the C struct from Listing 1. On many systems, due to alignment requirements:

- the struct itself is aligned on a 4-byte boundary as that is the largest alignment requirement of any structs field.
- `char a` is at offset 0 from the struct start.
- `int b` may be placed at offset 4 (not 1) to align on a 4-byte boundary.
- `short c` may be placed at offset 8 (following the 4-byte `int`) or if the compiler decides to tightly pack the structs content, at offset 2 between the other two fields.

The memory layout might thus include padding bytes between `char a` and `int b` to ensure proper alignment, potentially resulting in more space being used than the sum of individual field sizes. This, however, largely depends on the compiler, its configuration and the target hardware for which a program is compiled.

```
1   struct Example {
2       char a;      // 1 byte
3       int b;       // 4 bytes
4       short c;     // 2 bytes
5   };
```

Listing 1: An example C struct whose fields have different alignment requirements

Understanding memory layout helps in designing data structures that maximize performance and efficiency by leveraging optimal memory alignment and cache usage. It is a fundamental aspect when doing systems-level programming and developing performance-critical applications. Since the goal of the FlatBuffers encoding format is to stay as compatible to the hardware as possible, these layout aspects need to be considered as well. Adhering to the system's memory layout restrictions ensures that the data structures work correctly and efficiently across different hardware architectures.

2.4 FlatBuffers Workflow

Assuming that data about a `Person`, specifically their name and an age should be encoded with FlatBuffers, the following steps must be performed to do so. This example assumes that the Rust programming language is used and only includes explanations pertaining to the encoding. Transmitting or receiving the encoded data is not shown here since only the FlatBuffers specific parts are relevant.

1. **Write a schema file**:

   This file contains all data types, including their relations and nestings, that will be used by the application. Custom data types can be either of simple scalar values (e.g. integers of various sizes, floats, strings, etc.) or complex / combined types such as

structs, tables, unions or vectors. Data types allow for optional fields, defaults, and the ability to iterate on the schema without compromising backward compatibility.

An example schema is shown in Listing 2 which defines only a single data type called `Person` that contains their age and name as transferred information.

2. **Run the Code-Generator**:

   Use the *`flatc`* program to run a code generator specific to the target programming language. For example, to generate Rust code in the current directory, *flatc* could be invoked with *`flatc`* `--rust schema.fbs`. This would produce a `schema_generated.rs` file which contains Rust type definitions as well as serialization, validation and deserialization logic according to the schema.

3. **Encode a person's data**:

   After being generated by *flatc*, the code from `schema_generated.rs` can be used like any other. Listing 3 shows how a person whose name is "Erika Mustermann" and age is 42 is encoded using the provided `FlatBufferBuilder` struct. In line 4, a new `FlatBufferBuilder` is created, whose functions are subsequently used in lines 5 to 8 to create a `Person` object in the buffer and fill it with data. Finally, lines 10 and 11 use the FlatBufferBuilder API to mark the buffer as finished and retrieve the encoded data in a byte slice format. Like any other byte slice, the encoded data could now be transmitted via any preferred transport mechanism.

4. **Decode a person's data**:

   After having received an encoded person's data into the `encoded` variable, a receiver could validate and access the encoded data in a similar fashion to Listing 4. There, in line 3 the received binary data is interpreted as a `Person` object, which resides at the root of a FlatBuffers message. Line 4, which is a continuation of that statement, explicitly instructs the program to panic with the message "Invalid Person data" if the `encoded` variable does not contain valid data. Afterwards, the generated accessor functions can be used to retrieve the person's data as shown in lines 5 and 6.

```
1 table Person {
2   name: string;
3   age: int;
4 }
5
6 root_type Person;
```

Listing 2: An example FlatBuffers schema that defines a *Person* type.

```rust
use schema_generated::{Person, PersonArgs};
use flatbuffers::FlatBufferBuilder;

let mut fbb = FlatBufferBuilder::new();
let name = fbb.create_string("Erika Mustermann");
let person = Person::create(
  &mut fbb,
  &PersonArgs { name: &name, age: 42 });

fbb.finish(person, None);
let encoded: &[u8] = fbb.finished_data();
```

Listing 3: Example using code generated from the schema in Listing 2 to encode a Person object

```rust
use schema_generated::root_as_person;

let person = root_as_person(encoded)
  .expect("Invalid Person data");
assert_eq!(person.name(), "Erika Mustermann");
assert_eq!(person.age(), 42);
```

Listing 4: Example using code generated from the schema in Listing 2 to decode a Person object

3 Scope and Related Work

This section presents and overview over the combined work that has already been done on the topics of protocol design and security, protocol reverse engineering, patterns for dependable and secure API design and prior work specifically on FlatBuffers security. The exact formulation of the research questions guiding this thesis are also given and explained in Section 3.2.

3.1 Related Work

Much work has already been done covering the different aspects of designing, protecting and exploiting different protocols and data formats. This section focuses first on reverse-engineering, which is often a requirement to exploitation, then presents research into Denial-of-Service (DoS) vectors, attacks and scenarios and finally covers prior work on the topic of FlatBuffers.

3.1.1 Protocol Extraction and Reverse Engineering

In the literature, protocol reverse engineering and schema reverse engineering are closely related. Most protocol reverse engineering techniques can also be understood as schema reverse engineering since the general rules of a protocol are usually known and only the specific transmitted data types or messages are extracted. Related work exists on automatic protocol extraction, which can be further categorized into *Generic Automatic Extraction* and *Specific Protocol Extraction*. These categories are then further subdivided based on whether the extraction is performed from network traces or application binaries.

3.1.1.1 Generic Automatic Extraction

Approaches in this category are agnostic of the protocol itself. They extract the semantics from completely unknown protocols without any a priori knowledge and are often used in identifying which bits are utilized as counters, offsets or data fields. Being agnostic of the protocol is a significant advantage regarding the scope in which the techniques can be applied, however, there are notable drawbacks. Since these approaches operate generically, they cannot leverage protocol specific knowledge that might be known before the analysis. For example, they cannot leverage that the *Table* data structure in a FlatBuffers message is always encoded following certain rules that makes them easy to identify. Simultaneously, these approaches struggle with nested and complex schemas and can often only be used for gaining a basic understanding of an assortment of bytes. Consider the example of a FlatBuffers table that includes vectors with embedded tables. If the employed approach functions perfectly, it would only perceive the structure as a combination of length fields, offsets and data fields while not being able to identify protocol specific data types such as tables, vectors or integers in the case of FlatBuffers. Consequently, a reverse engineer would need to translate them into the structure of a FlatBuffers schema manually. While this is bene-

ficial for completely unknown protocols, when the protocol is known, it is advantageous to employ extractions methods tailored to that protocol.

Such generic extraction techniques are implemented utilizing techniques such as n-grams [10], sequence alignment algorithms [11], or delimiters [12] to tokenize messages by splitting them into segments. Then, by comparing the values of these tokens, they perform clustering to identify similar messages.

A prominent example is the *Netzob* software developed by G. Bossert, F. Guihéry, and G. Hiet [11], as they have also co-published code that is still maintained nine years later. For this purpose, the sequence alignment approaches of Needleman & Wunsch [13] and other clustering algorithms are used to identify similar messages throughout different inputs. However, as discussed before, the focus is less on the specific field format of the messages and more on field boundaries and their role in the encoding format.

There are also systems such as *ReFormat* by Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace [14] which is a system designed for protocol reverse engineering of encrypted messages. In contrast to *Netzbob*, *ReFormat* works on an application binary instead of encoded messages. Its methodology involves finding and classifying bitwise operations to detect potential cryptographic functions. The inputs and outputs of the discovered functions are then marked using a technique called taint-tracking to identify buffers affected by them. This allows the preceding unencrypted buffer to be found and the plaintext to be extracted. This approach could be adapted to find messages before encoding using the same methodology. While this can be used to reverse the encoding of a specific message, a generalized message schema cannot be reconstructed. Furthermore, it is not applicable when other parts of the application perform many bitwise operations.

3.1.1.2 Specific Protocol Extraction

A simple example and very prominent example of extractors designed for specific protocols is the *Wireshark* tool which can automatically detect and parse a wide range of network protocols, such as Transmission-Control-Protocol (TCP) streams, different VPN tunnels or application data transferred via HTTP. *Wireshark* is able to handle protocols such as TCP effectively because most are well-defined transport protocols that have consistent encoding rules that are independent from the transported data. This is not the case with protocols like ProtoBuf or FlatBuffers. In those cases, the schema is agreed upon between communication partners before transmission and then implicitly used[1], [3].

Analyzing protocols that use an implicit schema often requires specifically designed tools. These may offer assistance by semi-automatically analyzing messages and letting the user inspect and influence the analysis. One such example is the *protobuf-inspector* tool which can restore field boundaries and infer data types of ProtoBuf messages based on observed messages. The tools strategy however cannot be easily adapted to

fit FlatBuffers messages because ProtoBuf retains much more schema information on the wire, which is then utilized by the tool [15].

In cases where no automatic tool is available, manual protocol extraction is still often performed. One explanation for such a situation is the lack of implementations for theoretically working techniques because often, one is published in theory but an implementation is not. The *PRE-List* project contains many such cases [16]. Manual reverse engineering is especially often seen in botnet analysis, where understanding the communication mechanisms is important to enumerate or disable them, but intentionally kept hidden or obfuscated by the botnet authors. Thus, their protocols need to be manually reverse-engineered as were the cases for e.g. GameOverZeus [17], Sality [18], and many other botnets. An advantage to doing reverse engineering manually is that the schema and the semantics of its fields are determined to a high degree of satisfactions, however, it requires significant time and effort.

3.1.1.3 Reverse Engineering FlatBuffers

Some attempts have already been made to reverse-engineer messages encoded using FlatBuffers. For example, multiple approaches are discussed on the FlatBuffers GitHub repository under Issue #4258 [19].

One author suggests identifying known schemas via their `file_identifier`. This is a field whose position in FlatBuffers messages is known and which is used to uniquely identify schemas. While this method is very simple, it is only able to recognize already known schemas and cannot reconstruct them. Moreover, including a file identifier into a FlatBuffers message is optional. Another author gives a code example in Go on how to recognize and decode the *Table* data type in arbitrary FlatBuffers messages. This is a useful start since Tables are a useful and prominent data type that can be recognized with high accuracy.

A FlatBuffers schema can generally consist of only a few structural types which can be combined to form semantic types[20]. For example, Listing 2 shows a schema consisting of a *Person* type that is encoded via the *table* structure. It contains the attributes *name* which is a string, *age* which is an integer and *children* which is a list of more *Persons*. The different structures used here (i.e. *string*, *table*, *int*, *list*) all have different rules that dictate their encoding. Some of them (i.e. *tables*) are easier to recognize, as was discovered on the aforementioned GitHub issue, but others are not.

This work from the FlatBuffers community has already been expanded upon in the context of a research project which was conducted as part of the University of Hamburg's information security courses. That work is in the process of being published but since this process is still ongoing, an overview is given in. An overview over additional reverse engineering insights regarding FlatBuffers, including those that were gained as part of the research project is given in Section 4.2.

3.1.2 Denial of Service Vectors and Attacks

A DoS attack is a malicious attempt to render a computer, network service, or network resource unavailable to its intended users. The primary goal is to temporarily or indefinitely interrupt or suspend services of a target. Such attacks are often executed via the network and aim to disable services that are offered on it.

To mitigate DoS attacks, several preventive measures can be employed: Traffic filtering and rate limiting could be used to monitor and prevent traffic flooding, load balancing can be enabled to distribute traffic across multiple servers and intrusion detection systems (IDS) can be installed to identify and address suspicious activities early. An operator can also organize redundancy and failover systems to switch to when primary systems are attacked.

3.1.2.1 DoS Vectors

Commonly used attack vectors in include volumetric attacks, protocol attacks, application layer attacks, and distributed Denial-of-Service (DDoS) attacks.

Volumetric attacks, such as ICMP floods, UDP floods, and DNS amplification attacks, aim to consume the bandwidth of the target by overwhelming it with high volumes of traffic. Protocol attacks include techniques like SYN floods, which exploit properties of the protocol, in this case the TCP handshake by sending numerous SYN requests without completing the connection[21]. In the past, Ping of Death attacks, which send malformed or oversized ICMP packets to crash the target system, were also a reliable DoS vector[21]. Application layer attacks are similar in that they exploit certain behaviors of an application. The difference lies in the fact that an attacker technically conforms to the protocol, however their behavior impacts the application that is running on top of that protocol. Examples for these kinds of attacks are *Slowloris*, which keeps numerous connections to a target web server open to consume server resources[22], or DNS amplification attacks in which the global Domain-Name-System is manipulated into overwhelming a target[23]. DDoS attacks involve multiple compromised systems working together to execute a coordinated and amplified attack on a single target, making it tougher to mitigate.

3.1.2.2 Resources commonly impacted by DoS-Attacks

Attackers exploit various system properties as described above to deny service availability. Often specific resources of a target system are targeted and brought to exhaustion.

An attacker could overload the CPU with excessive computations or complex requests, exhaust memory by sending high-memory-consuming requests to force a system crash or slow response, or network connections through SYN floods or connection floods that exhaust available network sockets. Alternatively, a system's bandwidth can be exhausted by conducting high-volume attacks that choke the target's network capacity, disk I/O can be saturated with intensive read/write operations. Specific application resources can naturally also often be exhausted, rendering it nonfunctional for legitimate users.

3.1.3 Input-Validation and Language-Security

In computing, accepting untrusted input in a safe way has always been a difficult task and is a constant source of high-profile security bugs (i.e. *Heartbleed*, *GNU TLS CVE-2014-3466*, *Apples goto fail;* or *OpenSSH GOBBLES*). S. Bratus *et al.* [24] have assembled the most common mistakes, many of which are applicable to the handling of FlatBuffers messages.

The most important take-away is that an input can only be considered trusted when it has been verified in a way that not only asserts the structure of each component but instead also considers the context in which it is going to be used.

For example, when verifying the length or offset tag for a field, the validator must not only ensure that those fields have valid values of their own, but also that they are valid on their own (i.e. the offset does not point outside of the message) and that they *agree* with the offset and length tags of other fields. Another example is that of object nesting (which FlatBuffers supports). If a message schema allows the nesting of objects, the shape of the resulting object tree must be verified according to the program semantics or otherwise the program behavior is not predictable anymore [24].

3.1.4 FlatBuffers Security

The FlatBuffers designers and authors of the prevalent implementations, *flatc*, know that validating FlatBuffers message is important and have invested some work into its security. The community has also discussed the security as well as reverse engineering aspects of FlatBuffers.

For example, in an issue on the *flatc* project page GitHub user *d4l3k* proposes a small proof-of-concept that can parse parts of a FlatBuffers message without requiring any knowledge about the messages schema. Their approach is very limited but nonetheless shows that merely depending on a FlatBuffers schema being secret is not sufficient for safeguarding [19].

The author of flatcc (which is the *C* language's FlatBuffers implementation that is separate from the main *flatc* implementation by Google) has also noted some general issues when dealing with untrusted FlatBuffers messages in the "Verification" as well as "Risks" sections of the "FlatBuffers Binary Format" documentation[25]. In their own words, the goal of validation is "a basic fast assurance that the buffer is safe to access. Any additional verification is application specific. The verifier makes it safe to apply secondary validation" [25]. They state that while validation should check for possible alignment issues and possible out-of-bound data access, properties such as data overlapping, UTF-8 compliance of strings or validity of enum values are explicitly not checked[25].

It is further stated that modifying encoded FlatBuffers values in-place is not safe, that "it can be dangerous to print JSON or otherwise copy content blindly if there is no upper limit on the export size" and that issues may arise from the capacity of a platform's native integer and size types[25].

3.1.5 Dependable API Design

Existing research has covered the question of what exactly makes a program and its subsystems dependable. This is especially relevant as FlatBuffers are usually worked with via libraries like *flatc* which should facilitate building secure programs through its own design. G. Candea [26] argues that, if component interaction is done wrong, failure conditions can cascade chaotically and bring whole systems down.

Y. Acar *et al.* [27] as well as P. L. Gorski, S. Möller, S. Wiefling, and L. L. Iacono [28] have studied in detail the effect which the placement and presentation of security related information in documentation has on the quality of apps produced by consumers of that documentation (or the respective software which the documentation is for). As key results, those studies found that when developers have an incomplete understanding about what a library or its parts do, or don't understand options with which those libraries are configured, they produce less secure apps. This can for example manifest when developers of android apps don't use specially designed account management APIs that the android operating system provides and instead store user credentials plainly on the filesystem because they did not know about the security issues of doing so and weren't aware of the more secure alternative APIs. To prevent issues like this, the studies suggest that security should not have its own separate documentation pages that are buried somewhere or linked to. Instead, security topics should be discussed alongside general explanatory documents whenever they are relevant. Especially when a library's documentation contains code examples, developers tend to focus on those examples and skim over larger, more detailed explanations that are parts of larger text bodies. For this reason, security topics should always be included and code examples be provided[28]. Another more basic course of action is ensuring that all the API surface that is exposed to developers even has properly accessible documentation in the first place[27]. This documentation should also include failure conditions so that systems fail in a known way that can be expected and handled by its users[26].

Another study, this time specifically regarding SSL (nowadays superseded by TLS), was done by S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith [29]. They systematically analyzed the usage of certificate and connection security related functionality and afterwards held detailed surveys and interviews with the responsible app developers. Their study strengthens the aforementioned findings in that a lack of understanding is a main cause for insecure library usage. One of the more surprising points highlighted by this study is that developers may misunderstand overarching concepts, i.e. how a public key infrastructure conceptually operates, as well as details about specific function calls. Even when working on configuration of Transport-Layer-Security, some developers did not realize despite the name that the affected source code had security implications at all. The authors then suggest different API design guidelines for improving upon this:

1. An API should generally use secure defaults. Validation checks that are necessary in almost all use cases should thus be enabled by default. In the case of SSL validation, the example was for hostname validation to be required by default.

2. API semantics should be explicit. While it is not always easy to provide an API that provides good abstractions and is still explicit, arguing about what a function does becomes easier if its semantics are clear.
3. At the same time, the set of available configuration options should be small, so as to not overburden developers, which can lead to them choosing insecure settings.

3.1.6 Summary

The collected related works of the scientific community contains clear indications pertaining to the importance of securely designing distributed systems from concept to implementation. Topics such as validating untrusted inputs correctly and reverse engineering are thoroughly, on an abstract and general level, thoroughly explored fields. Nonetheless, due to the large number of radically different protocols and systems, knowledge gaps are often present when specific details are concerned. FlatBuffers is one such case which this thesis tries to remedy.

3.2 Research Questions

The goal of this thesis is thus to generate a dependable data set with which the impact of certain protocol design decisions as well as implementation decisions can be proven and evaluated. The related work section (Section 3.1) has already highlighted the importance of dealing with untrusted input correctly and even though some security issues are already known conceptually, their exact impact, especially in the context of FlatBuffers, is not.

A dataset which includes all implementations of the FlatBuffers format would be desirable, but this thesis nonetheless focuses on the recommended implementation only, which is *flatc*. This procedure was chosen because *flatc* is the predominant implementation that is most widely used, is recommended by the official website, has a large variety of features and includes support for almost all major programming languages. It was hence deemed important to give this implementation an in-depth evaluation. Specifically, *flatc* version `23.5.26` was subject to the work described in this thesis.

In doing so, the following research questions will be answered:

1. **Under which circumstances can what attacks be successfully executed?** This includes aspects such as answering if a system-under-attack needs to perform certain operations on a malicious FlatBuffers message, whether an attacker needs to know the message schema (and in how much detail) and whether the programming language in which *flatc* is used matters.
2. **What impact does which attack have on the system-under-attack?** Different kinds of attacks using malicious FlatBuffers messages are shown in this thesis, most of which are known to exist conceptually already. This thesis aims to answer how exactly these attacks impact a vulnerable system. The usage of system resources such as CPU time or memory of course depend on what exactly a program

is doing with the data it handles, but this thesis aims to answer which kinds of impacts can be achieved by which attack method.

3. **What is the relation between resource requirement on the attacker's side as compared to the effects on the system-under-attack?**
   When considering DoS-attacks, one of the important metrics when considering the effectiveness of such attacks is how the resource usage between the attacking system and the one being attacked relate to one another. In other words, a DoS-attack is not as effective if generating a malicious payload requires more resources than processing it does. Producing information regarding this relationship for different kinds of attacks and systems-under-attack is another goal of this thesis.

4. **Which relevance do the findings of this thesis have to real-world applications?**
   To assess the actual importance of this thesis' contribution, it is paramount to determine the impact which the examined attacks have on applications that are in use today. In other words; It is important to know whether an attack or finding is only relevant in highly theoretical and constructed scenarios or if an attacks preconditions are fulfilled by normal usage patterns.

Although the security of networked applications was thoroughly evaluated during the active research of this thesis, it was was nevertheless a critical requirement that no active and publicly available systems availability were ever degraded in any way. Especially if those were operated by unaffiliated third parties. All attacks and exploits were instead developed and evaluated in isolated systems.

4 FlatBuffers Protocol Analysis

This section contains a detailed explanation about the encoding details of FlatBuffers (See Section 4.1). An examination into how the details of the encoding format can be used to reverse-engineer a specific schema is included in Section 4.2 as schema knowledge is often required for crafting specific exploits. Section 4.3 then presents different kinds of attacks that utilize specific details of the encoding format for exploitation.

Generally, the scenario that is considered here is that of a distributed system which uses FlatBuffers encoded messages for its communication. Such a system is shown in Figure 2 where an attacker, Eve, who has gained knowledge about the systems schema and determined it to be vulnerable against DoS attacks, performs one such attack against the participant Bob. This attack disrupts the normal behavior operation of Bob so that requests from the legitimate client Alice cannot be processed anymore. The scenario was chosen as it is very general and applicable to almost all distributed systems (assuming they use FlatBuffers). It is useful to discuss attacks that are useful in such a generic scenario because they are therefore applicable and relevant to many situations.

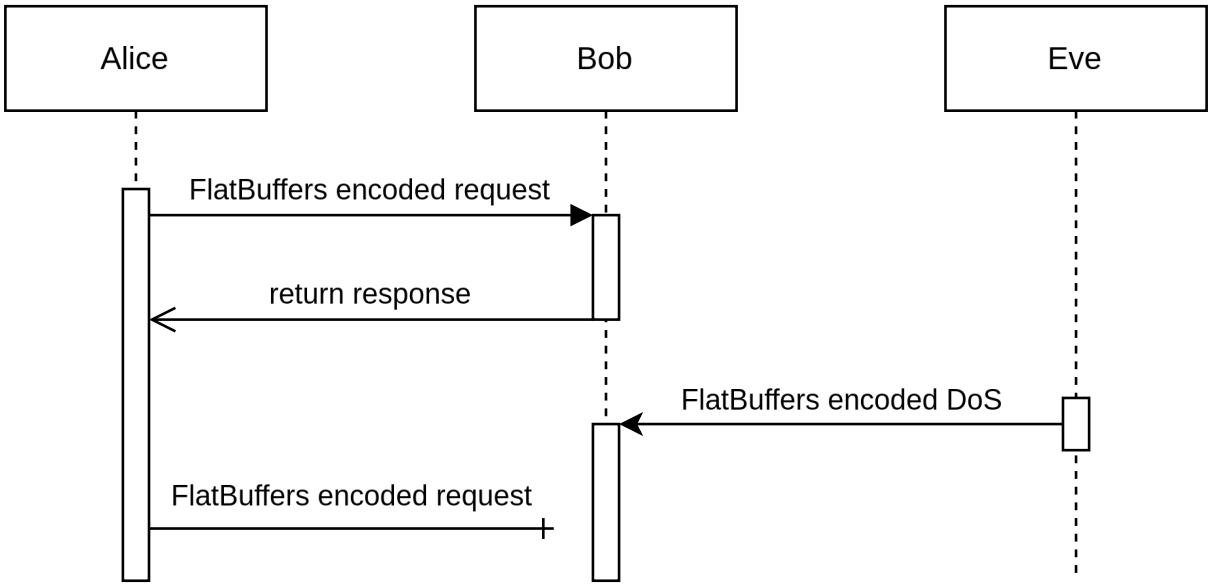


Figure 2: A scenario of a distributed system in which flaws in the FlatBuffer format and system behavior are be used for a denial of service

4.1 FlatBuffers Encoding Rules

FlatBuffers schemas can generally be freely composed from primitive data types (e.g. bytes, integers, floats, strings) and composition types like unions, lists, structs and tables. Apart from from some edge-cases these can be nested and combined arbitrarily. Each of these primitives has rules according to which they are encoded that are described in the following sections.

4.1.1 Scalar Primitives

- Double- and single-precision floating point numbers are encoded using the `IEEE-754`[1] format.
- Signed integer numbers are represented using the two's-complement.

All numbers are available from 8 to 64 bit width and integers can be declared signed or unsigned. All numbers are encoded as little-endian on the wire and aligned to their own size. They are also always encoded inline and never referenced via offsets.

4.1.2 Structs

Structs are the simplest way to combine other data types. They contain their field values linearly appended to each other while respecting the fields alignment rules. There is no additional metadata encoded in the buffer identifying a struct's format or content.

4.1.3 Tables

Tables are the most common and most versatile type of data structure that FlatBuffers offers. It is similar to a struct in that it contains named fields of different types but also offers advanced features.

For example, fields can be omitted when serializing objects or left at their default value. Doing this results in that field's data taking up no space in the encoded message. It is also possible to evolve a schema's table definitions while keeping forwards and backwards compatibility to messages and clients using an older schema version. Both these features are implemented by encoding storing addition information in the encoded format. This additional information is stored in an extra data structure called a *VTable*.

Table objects are encoded as shown in Figure 3. Whenever a table is referenced (for example being the root object of a buffer) it firstly contains another offset towards the accompanying VTable. Note that this is the only type of offset present in FlatBuffers that point backwards. After the VTable offset, a table encodes the values of its fields linearly and respecting field alignment. As discussed before however, which field's data is present or omitted is not always static, so a decoder needs to respect the referenced VTable. The VTable describes how many bytes the VTable as well as the referencing table need. Afterwards it has an entry for every field present in the schema with the offset from the start of the table as value. If an entry's value is 0, the corresponding field is decoded with its default value. Similarly, if a decoder expects more fields than are described by the VTable (for example when the schema has evolved and an older buffer is decoded), those fields are also decoded with their default values. The field values then have their own decoding rules according to the data type they encode. While most types are encoded inline, other tables are once again referenced using an offset which means only the offset is stored as a field's value.

[1] https://standards.ieee.org/ieee/754/6210/

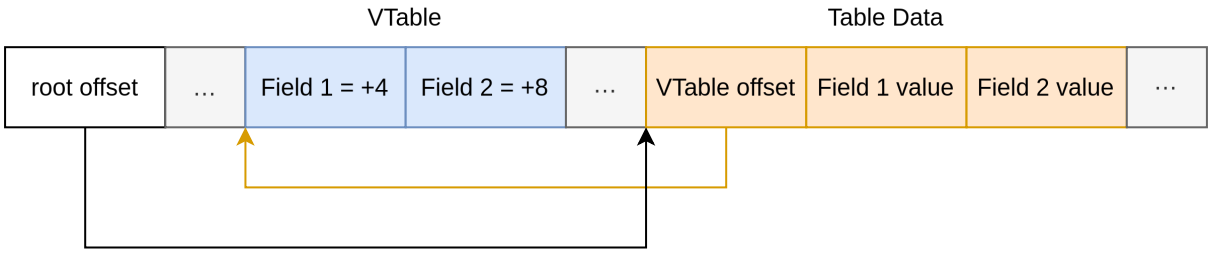


Figure 3: Example encoding of a table with two fields. The table data is marked orange and the accompanying VTable in blue.

4.1.4 Variable-Length Vectors

FlatBuffers includes support for lists of other data types whose child type is chosen at schema-build-time but whose length is only known at runtime. These lists are encoded as shown in Figure 4, which is to say they store their element contiguously aligned prefixed by a 32-bit unsigned element count. Vectors themselves are never stored inline but rather always referred to by offsets.

Figure 4: Example encoding of a vector with 3 elements. Vector metadata is marked blue and element data in orange.

4.1.5 Strings

Strings are stored as byte vectors with an additional null-terminator that does not count towards the vector's length. They otherwise follow the same encoding rules as vectors themselves.

String data should be UTF-8 encoded but this is not strictly required by the specification.

4.1.6 Enums

Enums are a way to safely group a set of named constant values into a type that can only ever have one of those values. Enums can be backed by any unsigned integer primitive depending on the number of variants needed and are encoded on the wire by encoding only the backing integer. No additional metadata about this data representing an enum is present in the encoded format.

4.1.7 Unions

Unions are encoded by combining an enum describing the union variant followed immediately by the value.

4.1.8 Offsets

FlatBuffers is very explicit in the way it handles offsets. All offsets used in encoded message are 32-bit unsigned integers and their direction is predetermined based on the place in the buffer the offset is used. In most cases, offsets point forward (towards higher buffer indexes) but there is the exception of VTable references (see Section 4.1.3).

This also means that it is impossible to create cyclic object references in a buffer since an object can not refer to itself or other objects that are placed before it.

4.1.9 Other Encoding Considerations

FlatBuffer messages as a whole also adhere to a largely consistent layout. The first 4 bytes of a buffer are always required to be an offset to the root object of the buffer which is also always required to be a table. Between these two is the table's accompanying VTable (since the table → vtable offset always points backwards). One such encoding can be seen in Figure 3. Additionally, a buffer can include a *filename* attribute which is a FlatBuffers encoded string that can be freely chosen at encoding time and which is placed directly after the initial root offset.

FlatBuffers data types also follow alignment rules which may result in additional padding bytes being generated. For example, structs which are always stored inline in their containers are themselves aligned according to the greatest alignment contained within them. When structs with large alignments are to be stored in e.g. tables, the encoder may choose to fill the padding space with the data of other, smaller fields contained in the table. This is unfortunately not always possible, for example when the table contains only that one struct or the container type is not a table at all (such as a vector). In those situations, the padding bytes are filled with zeros and otherwise ignored.

4.2 Reverse Engineering Considerations

When evaluating the attack surface of a potential target that uses FlatBuffers for communication, it is essential to know the schema which is used by that target. Inspired by the community's prior work, a tool specifically designed for reverse-engineering a FlatBuffers schema was developed in the course of a master project at the University of Hamburg's information security courses. This tool utilizes the few bits of structural information that are present even in an encoded FlatBuffers message to figure out which structural types that message could possibly contain.

To use this reverse-engineering tool, encoded FlatBuffers messages must first be captured. How to do so is outside of the tools scope but for example, other tools like *tcpdump* could be used for such a purpose. Once, a sufficiently large data set is assembled, the captured messages can be analyzed. The developed tool does so by attempting to interpret a message as each possibility that exists in the FlatBuffers type system (i.e. table, vector, enum, int, etc.). When a possibility is determined to be a structurally valid interpretation of the underlying bytes, and the interpreted type value contains

more fields, those fields are treated the same. The result of this analysis is a graph data structure that contains all possible variants as which a message could be interpreted. Naturally, since FlatBuffers has only few markers and encoding rules restricting interpretation, this resulting graph is very large, especially for complex schemas. For this reason, the analysis process can be repeated with multiple messages (all of the same schema) to build an intersection and hence reduces the size of the graph.

Listing 5 shows the described reverse-engineering algorithm in pseudocode. In Line 3, the entry point to the algorithm is shown which accepts an arbitrary byte buffer as input. The main functionality is implemented in the function `parse_unknown()` which attempts to interpret the buffer using all known data types such as an integer, a float and a table. Finally, the analysis result is intersected with previous results in line 4.

Figure 5 then shows an example binary value as well as a graph data structure similar to the ones produced by the reverse-engineering tool. In this case, the binary value could be interpreted to be either a string, a 64-bit integer or a combination of smaller integer values packed into a struct which the shown graph also displays.

```
1  let result = *;
2
3  fn analyze(buf) {
4    result = intersect(
5      result,
6      parse_unknown(buf)
7    );
8  }
9
10 fn parse_unknown(buf) {
11   return parse_as_int(buf) +
12     parse_as_float(buf) +
13     parse_as_table(buf) +
14     ...
15 }
16
17 fn parse_as_table(buf) {
18   return parse_vtable(buf)
19     .map(|field| {
20       parse_unknown(field)
21     })
22 }
23
24 ...
```

Listing 5: Pseudocode of the reverse-engineering algorithm

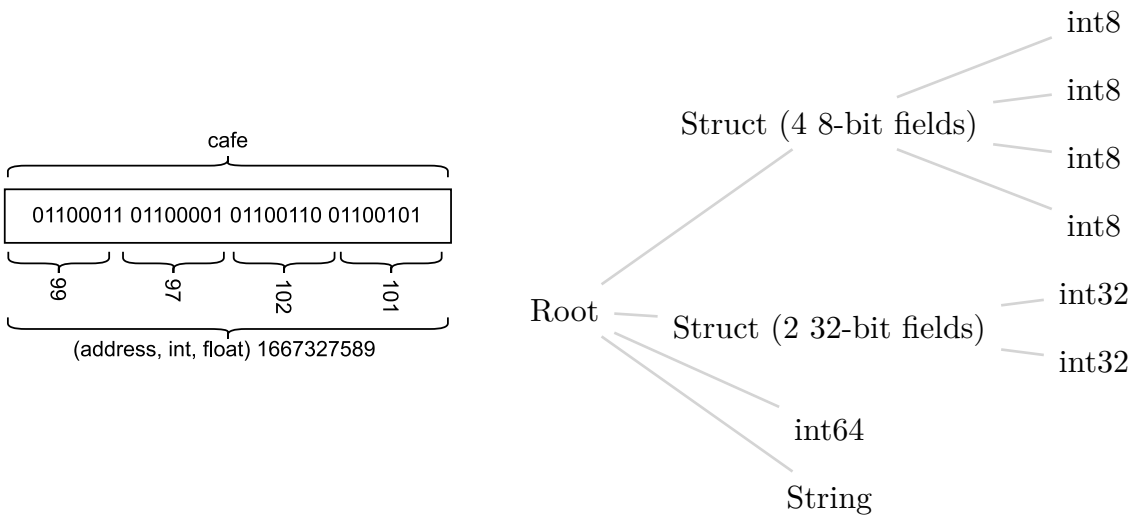


Figure 5: A set of bits that can be interpreted in different ways along with the resulting graph data that would be produced by the developed reverse-engineering algorithm.

The reverse-engineering tool is constructed with the following modules:

- An `encoding` module which parses the on-the-wire format of different FlatBuffers base types. It is completely independent from the actual research of this thesis and can pretty much be used as a generic parser for FlatBuffers independently from a schema. The functions defined in this module take a `&[u8]` buffer as argument and return a either an error if parsing failed or a specific type with safe accessors to the encoded data. It is implemented based on the *FlatBuffers Internals* documentation from *flatc*[20].
- A `typing` module that stores collected information about a schema in a memory-efficient way. It also implements type intersection and duplicate pruning which are needed during analysis.
- An `introspection` module which combines the two modules before. The functions in this module try to decode a buffer and if successful store the result in a type db. If the decoded type requires it (e.g. for a table), further introspection is performed on discovered child data (for all fields in the case of a table).

Parts of this infrastructure work have been reused for this thesis, for example to construct and introspect the various attack payloads that were used in Section 5.

As mentioned before, the general program structure as well as encoding rules for most FlatBuffers data types have been implemented as part of the Master project. Due to that project's time constraints, the resulting work can only be considered a proof concept as it does not yet support the encoding rules of *Union* data types as well as some specializations for type combinations. Nonetheless, the developed tooling was important for developing and verifying the different exploits introduced in the next section. It served as a baseline for all the work being done in this thesis while also highlighting that the mere secrecy of a communication schema is in no way sufficient in protecting a FlatBuffers-based communication endpoint as the schema can be reverse-engineered by a malicious actor.

4.3 Attack Vectors against the FlatBuffers format

Applications using FlatBuffers are vulnerable to certain attacks almost regardless of their implementation. Some risks and vulnerabilities just arise from the design of the encoding format and need to be treated with great care so as to not invite greater vulnerabilities that arise from mistreating these risks.

4.3.1 Format Violations

This category does not technically constitute an attack against the FlatBuffers format but rather its implementation. Conceptually, the kinds of attacks contained in this category intentionally violate certain encoding rules in order to exploit assumptions in a FlatBuffers implementations decoding and interpretation logic. Examples include invalid offsets that point outside of the encoded message, various forms of invalid string encodings, and other constructions in which two parts of a message should agree but don't. Optimally, these attacks don't impact a target system at all because all systems should validate all received messages and gracefully decline to process invalid ones. Related work, however, suggests that developers often make mistakes in security critical implementations or, as is the case for *flatc*, explicitly define certain invalid encodings as out-of-scope for validation. This suggests that, in reality, format violations may be able to cause a wide range of unpredictable behavior in victim applications.

4.3.2 Directed-Acyclic-Graph (DAG)-Explosion Attack

As discussed before, the FlatBuffers protocol provides the data structure of *tables* to hold arbitrarily structured data. They are encoded as described in Section 4.1.3. Under certain conditions a malicious sender is able to utilize these encoding rules so that a small encoded buffer grows exponentially large when its data is traversed.

The attack works similarly to a zip-bomb[30] or XML-bomb[31] and exploits the fact that offsets to tables are not enforced to be unique in a FlatBuffers message. Thus the same table data can be referenced any number of times without having to be copied.

A schema is vulnerable to DAG-Explosion attacks under the following conditions:

- It contains a type definition that is realized as a table
- That table is used somewhere in a list or array

The potential attack impact further varies depending on how many levels of *List of Table* nestings there are. Of course, a self-recursive schema is maximally vulnerable since it contains theoretically infinite levels of such nestings.

Listing 6 shows an example schema that contains the self-referential data type `VulnTable` and which is vulnerable against DAG-Explosion attacks. Figure 6 then shows a possible encoding of an attack against this vulnerable schema. The shown attack first contains a root value of `VulnTable` with the following field values:

- `space_waste` (line 9) is set to its default value. This means that the value is not stored in the encoded payload but instead synthesized by a decoder on the fly. It makes the attack payload smaller because no space is required to store this value.

- `children` (line 10) is set to a list containing n items where n can be chosen arbitrarily. Where the exploit comes in is that each of these children is actually the same `VulnTable` object.

This encoding schema can be repeated any number of times; in this case once more.

```
1  namespace SampleExploits.DagExplosionSchema;
2
3  // A struct that does nothing but takes up a bit of space in memory
4  struct LargeStruct {
5      ...
6  }
7
8  table VulnTable {
9      space_waste: LargeStruct;
10     children: [VulnTable];
11 }
12
13 root_type VulnTable;
```

Listing 6: A sample FlatBuffers schema that is vulnerable to DAG-Explosions

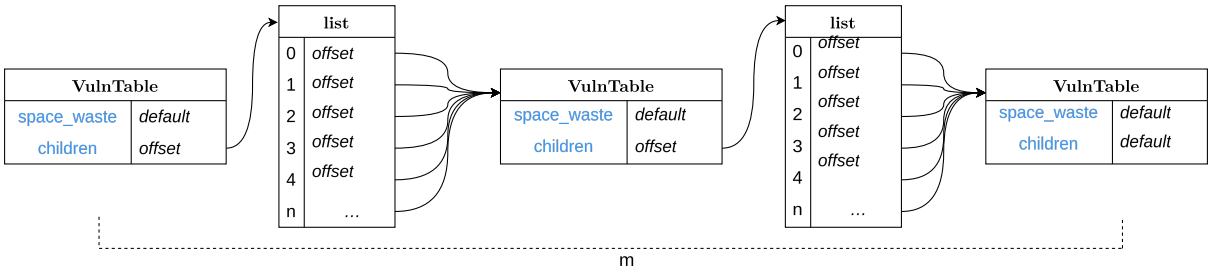


Figure 6: Schematic representation of a DAG-Explosion payload using Listing 6

Using the schema explained above, storing the deduplicated table data would require memory in the dimension of n^m where n is the number of table references in each list and m is the number of nesting levels. Encoding the attack payload on the other hand only required m tables in addition to $m \cdot n$ offsets stored in vectors. The resulting size dimensions are $\Theta(n^m)$ for an expanded representation and $\Theta(n \cdot m)$ in encoded form. This mismatch between polynomial and exponential growth is the problem exploited by a DAG-Explosion attack.

Other constructions of DAG-Explosions are also viable. For example, instead of a recursion using vectors, a binary tree node that always contains two children of its own type is equal to a recursive vector where the vector length is always two. While alternative DAG-Explosion compositions differ in exact details, the concept stays the same. For this reason, only the recursive vector construction is examined further.

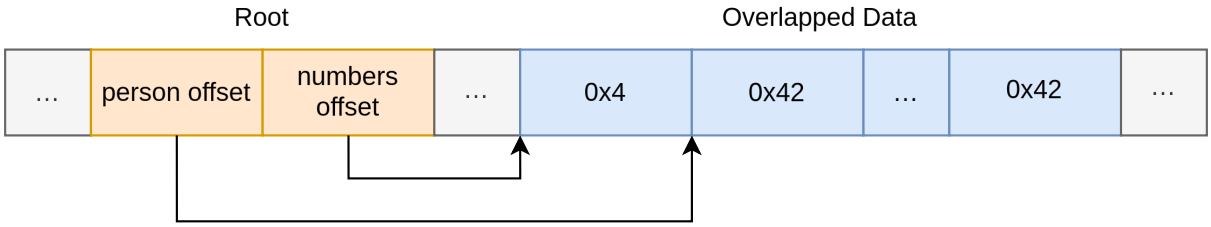
4.3.3 Data Overlap Attack

Another attack possibility for constructing malicious messages is the construction of unexpected overlaps. Overlapping data can lead to a number of issues similar to working with Unions in *C* especially when the overlapped data is being written to. For example, a set of bits which are overlapped to be interpreted as an integer as well as a boolean can lead to undefined behavior if, for example, the integer value `3` is written to it. `3` is, after all, not a valid value for a boolean. This section explains two different constructions for overlapping data, both equally dangerous.

The simplest form of overlapping data is to store multiple fields of one table at the same position in memory and describe that layout using a VTable that is constructed accordingly. An example is shown in Listing 7. There, a table *Root* is defined which contains two fields of different types. Those two fields are then encoded in a way so that they both point to the same area. When accessing `root.numbers`, `0x4` is interpreted as the length of the vector with the following data being 32-bit signed integer values. Accessing `root.person` on the other hand interprets the same `0x42` value that was an integer in the previous case as a VTable offset and everything else as table data.

Another, more complex, form of overlap is present when not just fields inside one table are overlapping but when completely unrelated data structures are stored at the same position. This attack involves setting offsets in a way that the pointed-to-data overlaps. They might originate from the same table but don't have to. Any type that is encoded by offset can be overlapped with any other data in a buffer as long as the offset direction allows it (remember: offsets only point further into the buffer (See Section 4.1.8)).

```
1  table Person {
2    …
3  }
4
5  table Root {
6    person: Person;
7    numbers: [int];
8  }
9
10 root_type Root;
```



Listing 7: An example FlatBuffers schema along with a possible encoding of it that contains such an overlap.

5 Practical Security Evaluation

The evaluation is split into two parts. First, various Open Source projects' schemas are analyzed in Section 5.1. The different languages supported by *flatc* are also evaluated for their built-in security mechanisms in Section 5.2. Afterwards, the DAG-Explosion attack is given special attention in Section 5.3 where the impacts that such an attack can have on a system is discussed. In doing so, the research questions explained in Section 3.2 are answered.

This section aims to provide detailed responses to the research questions formulated in Section 3.2. To briefly repeat, this entails answering which preconditions are required to execute attacks, what impact each attack has on the target, how many resources are bound on the attacker's and victim's systems, and how relevant the discussed attacks and vulnerabilities are for real-world applications.

5.1 Real-World Vulnerable Schemas

5.1.1 GitHub Schema Crawling

For evaluation of the schema analysis algorithm as well as for the part of this thesis that evaluated publicly available schemas themselves, it was crucial to have a large data-set of schema files. This dataset was created in the process of this work and is called *flatbuf-folklore*. It contains the metadata and FlatBuffers schema data shown in Table 1. It enables deduplication of schema texts that might be used in many repositories e.g. forks.

schemas		
schema_id	schema_text	valid
`text`	`text`	`integer`

githubmeta			
schema_id	repository	file_path	repository_url
`text`	`text`	`text`	`text`

Table 1: The schema of the folklore database

The best source for publicly available source code (which schema files can be considered being) is GitHub with over repositories. GitHub also offers a public API that can be used to search all public repositories using a dedicated search syntax[2]. This search API was used with the filter expression listed in Listing 8 to retrieve 496 from 1.064 repositories.

[2]https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28

```
table extension:fbs size:>255 size:<65536 in:file
```

Listing 8: The search-expression used to filter for public schema files

5.1.2 Automated Schema Analysis

For analyzing a schemas vulnerability regarding DAG-Explosion attacks (see Section 4.3.2) a custom script has been built that uses the *folklore* dataset, analyzes the contained schemas and calculates a vulnerability score.

The script uses *flatc* and performs the following steps:

1. Fetch a random schema from *folklore*.
2. Convert the schema text into a machine readable representation using *flatc*.

   This works by writing the schema text to a file, then calling *flatc* to generate an intermediate representation of the schema[3]. This intermediate representation is a FlatBuffers encoded binary blob. That binary blob is then read back into the program and decoded using the reflection schema provided by *flatc*[4].
3. The schema is then walked from its root in depth-first order and converted into one flattened list which has the *path* in the schema as its item.

   This path description brought into human readable form could look like the path seen in Listing 9. There, the script analyzed the *Apache Arrow* schema and found a recursive definition of `Field::children` referring to other `Field`s via a vector. As discussed in Section 4.3.2 this is a prime example of a schema vulnerable to DAG-Explosions.

   ```
   /org.apache.arrow.flatbuf.Schema::fields/[]
     /org.apache.arrow.flatbuf.Field::children/[]
     /org.apache.arrow.flatbuf.Field::children/[]
     /org.apache.arrow.flatbuf.Field::children/[]
     /org.apache.arrow.flatbuf.Field::children/[]
     /…
   ```

   Listing 9: An example of a single path description of a schema vulnerable to DAG-Explosions.

4. All vulnerable paths of all analyzed schemas are aggregated and stored back into a database for further manual verification and analysis.

5.1.3 Results

Using the procedure described above, 496 unique schemas were analyzed from 1064 different GitHub repositories. In this analysis, some schemas were found to principally vulnerable against DAG-Explosions. Many of those store generic other data structures

[3] https://flatbuffers.dev/intermediate_representation.html

[4] https://github.com/google/flatbuffers/blob/fb9afbafc7dfe226b9db54d4923bfb8839635274/reflection/reflection.fbs

e.g. ProtoBuf inside FlatBuffers, store graph data e.g. the node graph in a visual programming editor, or are data processing libraries which must also be able to handle arbitrarily complex data provided at runtime. The most prominent and high profile schemas found to be vulnerable were those of *Tensorflow Lite for Microcontrollers (tflite-micro)* and *Apache Arrow* and the *Cocos2D* editor.

In the case of *tflite-micro* alone, 105 different data paths were found that could lead to large expansion in a DAG-Explosion attack. *tflite-micro* was also not found to contain any references to the `flatbuffers::Verifier` struct apart from the definitions generated by *flatc*. The generated validation functions for the root type (`bool VerifyModelBuffer(::flatbuffers::Verifier &verifier);`) was also not found to be referenced anywhere in the tflite code base. This suggests that the tflite authors consider the model files (which is the part that's encoded using FlatBuffers) trusted and not requiring validation. While one could argue that models are usually chosen, tested and integrated during development and are not provided dynamically by users at runtime, it is still a risk to skip the simple step of validating model files at all. This risk is especially relevant when considering that developers don't always train models from scratch and instead may rely on pre-trained models provided by third parties[5]. Downloading models from the internet and loading them into a system without validating the underlying file format may be technically described as a bad idea. This is especially true when considering that *tflite-micro* aims to bring machine learning models to embedded platforms, which may behave differently when handling the same model file. A developer's test device might ignore invalid data alignments in a model file and process it just fine while other deployment platforms might not. This would lead to very hard to debug faults during deployment.

Similar to *tflite-micro*, the Cocos2D editor was also not found to include any calls to validation related FlatBuffers functions. The FlatBuffers format is used by Cocos2D for various internal representations e.g. the bone structure of a model. These files are usually created when developing or packaging a game but nonetheless, not verifying their structure in any way is certainly negligent and causes a large attack surface on at least the editor and possibly packaged games as well.

Regarding *Apache Arrow:* While the schema itself contains the exploitable pattern of storing a type recursively in itself with an additional vector in between (see Section 4.3.2), the authors take care to not allow that fact to be exploitable. They do this by binding the allowed number of tables to the size of the encoded FlatBuffers. In doing so, they ensure that messages which contain a lot of data and thus a lot of tables pass this validation while DAG-Explosions which gain their effectiveness by reusing large parts of the buffer don't. To perform this validation, the *Apache Arrow* authors use the validator provided by *flatc* with the construction arguments shown in Listing 10.

[5]https://www.kaggle.com/models?format=lite

```
1 bool VerifyFlatbuffers(const uint8_t* data, int64_t size) {
2   flatbuffers::Verifier verifier(
3     data, size,
4     /*max_depth=*/128,
5     /*max_tables=*/(8 * size));
6   return verifier.VerifyBuffer<RootType>(nullptr);
```

Listing 10: Verifier construction in *Apache Arrow*

5.2 FlatBuffers Language Implementation Comparison

flatc ships with code generators and support libraries for the languages *C++*, *Java*, *C#*, *Go*, *Python*, *JavaScript*, *TypeScript*, *C*, *PHP*, *Dart*, *Lobster*, *Rust* and *Switft*.

5.2.1 Language Comparison Environment

For the evaluation, the scenario introduced in Section 4 is implemented by a simple TCP server in the languages *C++*, *Java*, *C#*, *Go*, *Python*, *TypeScript* and *Rust*. *C* was skipped because of its similarity to *C++* where one of the two was deemed enough and also because *C* support is not implemented by *flatc* but by *flatcc* instead. Similarly, *JavaScript* was skipped because the *TypeScript* implementation is the same one just with added type annotations. Other languages were deemed too niche to be worth the effort and an iOS device was not available to evaluate the *Swift* implementation.

The TCP Servers were configured to accept a schema that is intentionally vulnerable to the different attacks discussed in Section 4. It is shown in Listing 11 and the algorithm is drawn up as pseudocode in Listing 12. The algorithm and schema are crafted specifically for this evaluation and designed to evaluate all parts of a received buffer in a simple way so that possible attacks are as effective as possible. Whatever a user of FlatBuffers *can* do wrong, these servers *do* wrong. To summarize, each server listens for incoming FlatBuffers messages via TCP and interprets any received messages with the given schema. Validation is enabled and run (if the language has a validator implementation). Afterwards, the received data is walked recursively and the `vec_data` and `name` attributes of the root object are evaluated and printed.

```
namespace ThesisSell.ImpactEval;

struct LargeStruct {
    a: uint64;
    ...
    l: uint64;
}

table SelfTable {
    space_waste: LargeStruct;
    children: [SelfTable];
    name: string;
    vec_data: [uint8];
}

root_type SelfTable;
```

Listing 11: The vulnerable schema that was used for this evaluation

```
def main():
    conn = wait_for_tcp_connection()
    raw_buf = conn.receive()
    root_table = SelfTable.from_root(raw_buf)
    print("Sum:", sum(root_table.vec_data))
    walk_table(root_table)

def walk_table(root):
    if root.name is not None:
        print("Got name {root.name}")
    for child in root.children:
        walk_table(child)
```

Listing 12: The algorithm that is implemented in different programming languages for this evaluation

The environment of course also included an *attacker*, that is a TCP client which connects to the TCP servers and intentionally sends malicious payloads to them. This client implements the attacks listed below to evaluate the behavior of the servers.

1. **DagExplosion**

   DAG-Explosion attacks are described in detail in Section 4.3.2 and a schematic representation of a DAG-Explosion attack is shown in Figure 6. The client of this evaluation implements such an attack with a configurable amount of levels of nesting as well as list length on each level.

   This attack is expected to require significant time for processing and to be able to exhaust a victim application's memory if the message is attempted to be converted to a linear format.

2. **InvalidStringEncoding**

This experiment sends a string value that is not a valid UTF-8 value in the `SelfTable.name` field to the servers. The FlatBuffers specification states that strings must be UTF-8 encoded[20] which this experiment violates. Since the server algorithm evaluates and prints these strings, the experiment allows monitoring how each language's implementation reacts to such incorrectly encoded strings.

The concrete payload sent by the client is shown in Listing 13. The string framing (e.g. null terminator and length prefix) is valid while its content is set to byte `0x80` which is invalid because it indicates a continuation byte while no other byte is present in the string.

```
string (SelfTable.name):
  +0x18 | 0B 00 00 00 | uint32_t | 0x0B (11) | length of string
  +0x1C | 48 65 6C 6C | char[11] | Hell      | string literal
  +0x20 | 6F 20 57 6F |          | o Wo      |
  +0x24 | 72 6C 64    |          | rld       |
  +0x27 | 00          | char     | 0x00 (0)  | string terminator
```

```
string (SelfTable.name):
  +0x18 | 01 00 00 00 | uint32_t | 0x01 (1) | length of string
  +0x1C | 80          | char[1]  | ?        | string literal
  +0x1D | 00          | char     | 0x00 (0) | string terminator
```

Listing 13: Comparison of two annotated FlatBuffers messages, one of which is valid while the other is not. The incorrectly encoded UTF-8 content is marked orange.

Expected results from this attack vary. A victim system could potentially fail unexpectedly or enter a state of undefined behavior if UTF-8 strings are strictly required. Alternatively, if a programming language neither requires nor expects string to be valid UTF-8, the encoding error could go unnoticed.

3. **InvalidStringLength**

   This experiment, similar to the one before, sends an incorrectly encoded string to the target server. In this case, however, the string data is valid while its framing is invalid because the length prefix is shorter than the actual string data is long. This results in the string not having a valid null terminator. The exact encoding is given in Listing 14.

```
string (SelfTable.name):
  +0x18 | 1D 00 00 00 | uint32_t | 0x1D (29) | length of string
  +0x1C | 6E 61 6D 65 | char[29] | name       | string literal
  +0x20 | 20 77 69 74 |          |  wit       |
  +0x24 | 68 20 69 6E |          | h in       |
  +0x28 | 76 61 6C 69 |          | vali       |
  +0x2C | 64 20 76 65 |          | d ve       |
  +0x30 | 63 74 6F 72 |          | ctor       |
  +0x34 | 20 6C 65 6E |          |  len       |
  +0x38 | 67          |          | g          |
  +0x39 | 74          | char     | 0x74 (t)   | string terminator

unknown (no known references):
  +0x3A | 68 00 00 00 00 00       | ?uint8_t[6] | h.....                    |
```

Listing 14: An annotated FlatBuffers message that contains an incorrectly framed string. The incorrect string length is marked orange and the thus left over string content blue.

4. **InvalidVecLength**

   Similar to *InvalidStringLength* this experiment constructs a vector whose length prefix is invalid. The difference is that in this experiment, the length prefix is too large and that the impacted data is not a string. Strings might be handled in a special way so this experiment also doubles in uncovering potential differences there. The exact encoding is shown in Listing 15.

```
vector (SelfTable.vec_data):
  +0x18 | A4 00 00 00 | uint32_t     | 0xA4 (164) | length of vector
  +0x1C | 55 55 55 55 | uint8_t[164] | UUUU       | vector data
  +0x20 | 00 00 00 00 |              | ....       |
```

Listing 15: An annotated FlatBuffers message that contains a vector with an out-of-bounds length prefix. The vector length pointing outside of the message is marked orange.

   A successful execution of this attack can lead to out-of-bounds accesses. By the *C++* definition, this may already constitute undefined behavior. In practice, out-of-bounds accesses could lead to leaking secrets or causing a program to crash when it attempts to access invalid memory regions.

5. **InvalidForwardOffset**

   In this experiment, the attacking client constructs a buffer in which the offset to the `SelfTable.name` field points outside of the buffer. The exact encoding is shown in Listing 16.

```
root_table (SelfTable):
  +0x10 | 0C 00 00 00 | SOffset32 | 0x0C (12)  | offset to vtable
  +0x14 | A4 00 00 00 | UOffset32 | 0xA4 (164) | offset to field `name`
```

Listing 16: An annotated FlatBuffers message which contains an offset to outside the message. The offset pointing to data outside the message is marked orange.

The exploit scenario is the same as with the *InvalidVecLength* attack in that the victim is intended to be manipulated into performing out-of-bounds memory accesses. The difference is the that the exploit is delivered through a different part of the FlatBuffers format.

6. **InvalidBackwardOffset**

   This experiment also constructs a buffer with an invalid offset except that in this case, the manipulated offset is the offset to a *VTable* and it is constructed to point to a memory location *before* the message. The exact encoding is shown in Listing 17.

```
root_table (SelfTable):
  +0x10 | 64 00 00 00 | SOffset32 | 0x64 (100) | offset to vtable
```

Listing 17: An annotated FlatBuffers message which contains an offset to before the message. The offset to the `root_table`'s VTable, which points to before the buffer starts is marked orange.

The exploit scenario is the same as with the *InvalidVecLength* attack in that the victim is intended to be manipulated into performing out-of-bounds memory accesses. The difference is the that the exploit is delivered through a different part of the FlatBuffers format.

7. **InvalidRootOffset**

   The buffer constructed in this experiment also uses an invalid offset as its method of attack. In this case, the affected offset is the very first value encountered in any FlatBuffers message: The offset to the root table that is contained in the message. This is constructed as a separate experiment to evaluate whether the root offset is handled in a special way by different language's implementations. The exact encoding is shown in Listing 18.

```
header:
  +0x00 | 64 00 00 00 | UOffset32 | 0x64 (100) | offset to root `SelfTable`
```

Listing 18: An annotated FlatBuffers message which contains an invalid root object offset. The invalid root offset (which is the also the only content of this message) is marked orange.

The exploit scenario is the same as with the *InvalidVecLength* attack in that the victim is intended to be manipulated into performing out-of-bounds memory accesses. The difference is the that the exploit is delivered through a different part of the

FlatBuffers format. This exploit is especially important to consider as it requires no knowledge of the message schema in use as the exploit payload (See Listing 18) contains only a single 4 byte offset and no further schema specific data.

8. **AttributeOverlap**

   This experiment overlaps different attributes of the same table as discussed in Section 4.3.3. Its goal is to verify how the different language's validators check for these kind of constructions. The exact encoding is shown in Listing 19.

   The impact of this attack is not directly predictable as most conditions induced by it fall under the category of undefined behavior. If the attack can be successfully executed, it can cause a wide variety of problems that strongly depend on how exactly the overlapped data is used by an application.

```
1  vtable (SelfTable):
2    +0x04 | 0C 00       | uint16_t  | 0x0C (12) | size of this vtable
3    +0x06 | 04 00       | uint16_t  | 0x04 (4)  | size of referring table
4    +0x08 | 00 00       | VOffset16 | 0x00 (0)  | start of `space_waste`
5    +0x0A | 00 00       | VOffset16 | 0x00 (0)  | start of `children`
6    +0x0C | 04 00       | VOffset16 | 0x04 (4)  | start of `name`
7    +0x0E | 04 00       | VOffset16 | 0x04 (4)  | start of `vec_data`
8
9  root_table (SelfTable):
10   +0x10 | 0C 00 00 00 | SOffset32 | 0x0C (12) | offset to vtable
11   +0x14 | 04 00 00 00 | UOffset32 | 0x04 (4)  | offset to `name`
12   +0x14 | 04 00 00 00 | UOffset32 | 0x04 (4)  | offset to `vec_data`
13
14 string (SelfTable.name):
15   +0x18 | 0F 00 00 00 | uint32_t | 0x0F (15)  | length of string
16   +0x1C | 6F 76 65 72 | char[15] | over       | string literal
17   +0x20 | 6C 61 70 70 |          | lapp       |
18   +0x24 | 65 64 20 6E |          | ed n       |
19   +0x28 | 61 6D 65    |          | ame        |
20   +0x2B | 00          | char     | 0x00 (0)   | string terminator
```

   Listing 19: An annotated FlatBuffers message which contains overlapped attributes. The overlapping encoding inside the tables VTable is marked orange and the overlapped data blue. Notice how the address of the overlapped data (`+0x14`) is the same.

The servers and client are then run using the monitoring tooling discussed in Section 5.2.2.

5.2.2 Attack Impact Measurement

To determine the impact of each attack, targeted applications resource usage had to be closely monitored. Fortunately, the evaluation environment (see Section 5.2) runs on GNU/Linux which offers advanced capabilities to monitor a process's resource consumption and execution environment. One of these capabilities is the Control-Group (CGroup) system. They are a system of the Linux kernel that allows allocation, track-

ing and limiting of resources such as CPU time, memory, I/O bandwidth, device access or network traffic control for a collection of processes. They are for useful day to day server operations as well as security research as they can provide detailed resource management and isolation capabilities.

CGroups also play a crucial role in containerization technologies like Docker, LXC (Linux Containers), and Kubernetes. By using CGroups in conjunction with kernel namespacing features, processes are effectively isolated not just in terms of resources but also in terms of visibility and access to kernel objects. This isolation mitigates the risk of resource exhaustion attacks and improves overall system security.

CGroups are an abstract capability of the kernel and need a specific CGroup manager to actually use them. They also come in two flavors, V1 and V2. Most modern systems use SystemD as a manager with the CGroup v2 API. With SystemD, the concept of *Units* is used to manage CGroups as each unit can be associated with its own CGroup in the CGroup hierarchy. Units (and therefore CGroups) may be defined statically or created and modified at runtime (SystemD calls these transient units). For example, the CLI `systemd-run` can be used to run a command under a new transient CGroup similar to how `sudo` is used to run a command as root.

To measure the impact of attacks on application servers that use FlatBuffers as their message format CGroups are extensively used. The reason for using CGroups instead of just using per-process accounting features of the Linux kernel is that an application server may spawn an arbitrary number of worker processes and otherwise behave quite unpredictable in terms of its process management. A measurement tool would therefore be required to walk, measure and collate the whole process tree of a system under test. CGroups however do that automatically which means that the tool only needs to gather statistics about one kernel object which radically simplifies it.

The tool itself is implemented in Python and can be accessed as part of this thesis's accompanying source code. It is written in the Python programming language and spawns a given command-line as a subprocess using `systemd-run` with the right collection of arguments. `systemd-run`'s output is then evaluated to find the correct CGroup of the spawned process and while that process is running, the CGroups resource usage is recorded regularly and written to a user-specified log file in JSON. Standard output and standard error of the invoked program are also captured, written to a log file while and additionally copied to the instrumentation script's standard output. The tool captures the contents of the following files inside the CGroups sysfs directory

- `cgroup.procs` for the list of process IDs that are active in the CGroup,
- `cgroup.threads` for the list of thread IDs that are active in the CGroup,
- `memory.current` which contains the currently amount of bytes allocated to processes in the CGroup,
- `cpu.stat` for CPU time accounting as well as
- `memory.stat` and `pids.current` which turned out to not be needed and were not used.

An example invocation is given in Listing 20 where `echo Hello World` is run and instrumented. The program outputs which CGroup it uses for instrumentation (indicating

that invoking `systemd-run` and parsing its output was successful) and then prints "Hello World" to the terminal. At the same time, the file `stats.json` is filled with one entry (because `echo Hello World` is a very quick-to-finish program) that shows some data about the invoked program while the file `log.txt` is filled with the command's output ("Hello World"). In this example, only one process was active in the CGroup which used 1500 microseconds of CPU time and allocated `360.448` bytes of memory.

```
# invocation
./run_instrumented.py \
  ./stats.json \
  ./log.txt \
  echo Hello World
```

```
# log.txt
using cgroup … for instrumentation
Hello World
```

```
// stats.json
[
  {
    "time": 1721081848.269017,
    "memory.current": 360448,
    "pids.current": 1,
    "cpu.stat": {
      "usage_usec": 1500,
      },
  },
]
```

Listing 20: An example invocation of the `run_instrumented.py` script with abbreviated output

5.2.3 Language Comparison Results

As shown before, *flatc* implements support for many programming languages, however one of the obvious results is that indeed not all programming languages are supported equally well. The next sections discuss the results for each of the evaluated languages while also starting with some general remarks.

5.2.3.1 General Shortcomings

The official FlatBuffers website lists in its language comparison chart only the languages *C++* and *C* to even have a buffer validator[32]. This was found to be incorrect in that *Rust* can also validate buffers before processing them further. *C#* also has validation logic defined in its source code[6] but does not explain or mention how it can be used in any of the documentation material. The documentation is therefore obviously outdated.

An interesting detail is that the *Rust* language's implementation is the only one which enforces usage of its validation logic or even mentions the need for validation (and how to use it) clearly in its documentation. For the other languages in which a validator is implemented (*C++* and *C#*), the main tutorial site does not mention it at all[33]. Only the advanced documentation page that is "designed to cover the nuances of FlatBuffers usage, specific to C++" [34] has any mention of accessing untrusted buffers and using a validator. It even steers you back towards the main tutorial in its first paragraph which supposedly "has a complete guide to general FlatBuffers usage" [34]. That main tutorial then explains: "assuming you have a buffer of bytes received from disk, net-

[6] https://github.com/google/flatbuffers/blob/master/net/FlatBuffers/FlatBufferVerify.cs

work, etc., you can start accessing the buffer like so" [33] while showing the content of Listing 21. The documentation fails to mention any risks associated with using an untrusted FlatBuffers that way. Quite the opposite, it tells the reader that this is how they should deal with FlatBuffers that were received from the network.

```
1 uint8_t *buffer_pointer = /* the data you just read */;
2
3 // Get a pointer to the root object inside the buffer.
4 auto monster = GetMonster(buffer_pointer);
```

Listing 21: How to interpret a buffer as a FlatBuffers defined `Monster` object[1]

FlatBuffers implementations also heavily rely on external knowledge about the size of the buffer for much of their validation logic. In languages like *C++*, which operates on raw pointers, this is especially apparent. Other languages' implementations often use special *Buffer* types which carry a size information with it. When transmitting FlatBuffers messages, this size information is by default not transferred, so a programmer needs to ensure that a message's size is also transmitted so as to not falsify validation results. This can be enabled by using a size-prefixed encoding which *flatc* implements code generators for but must be manually done. This is also not mentioned in the documentation.

5.2.3.2 C++

The C++ languages FlatBuffers implementation can generally be considered one of the more mature ones. The FlatBuffers documentation as well as the experiments conducted here show as much.

The *flatc* authors supply a validator for C++ which is able to validate FlatBuffers messages and find most format errors. It also verifies that no offsets stored in the message point outside it so that the calling code does not unintentionally produce out-of-bound accesses. The same is true for vector elements that are arithmetically located outside the buffer. These out-of-bound checks however only produce accurate results when the validator is given the correct message size. The code shown in Listing 22 would thus produce incorrect results as the validator would have to be constructed with `n` as the length parameter instead of `4096`. Using an incorrect value makes it possible to manipulate user code into accessing any value between the start of the buffer and (in this case) `4096` bytes further.

On a technical note, the *C++* implementation uses pointer arithmetics whenever any part of a FlatBuffers encoded message is accessed. Pointer arithmetics can be used to efficiently access memory by treating memory addresses as numbers and perform calculations on them. This can lead to more efficient algorithms but the resulting new addresses must always be treated with great care to ensure they still refer to valid memory areas. *flatc*'s *C++* implementation was found to arithmetically combine a messages base address with offsets contained therein. Only if validation is enabled, are those offsets verified to ways refer to other areas inside the message. If validation is

disabled, an application using FlatBuffers can trivially be manipulated to read from (and depending on the application, also write to) arbitrary locations in the programs memory. This can lead to secrets being leaked, the application crashing (e.g. through a null-pointer dereference) or cause undefined behavior[7].

At the same time, the usage of this validator is entirely optional. In the normal data-access code-path, *flatc* basically only does pointer arithmetics without any bounds or alignment checks. If a user processes untrusted FlatBuffers messages, they **must** remember to use a validator as otherwise, an attacker can construct reads into almost arbitrary memory regions. This includes invalid ones such as `0x0` which crashes the program.

What Listing 22 also shows is the construction of the `Verifier::Options` struct with default values. Those default values are shown in Listing 23. They are chosen to "be sufficient for most uses" [34] which is true, but the rather large defaults leave a user vulnerable to potential DAG-Explosion attacks. Exactly how vulnerable, is discussed in Section 5.3.

It should also be noted that the validator only returns a simple *valid* or *not valid* response without any further details about what exactly is wrong with the buffer. This might not be a security concern, but it certainly makes debugging invalid buffers harder.

```
1 char buffer[4096];
2 n = read(socket, buffer, 4096);
3 auto opts = flatbuffers::Verifier::Options{};
4 auto verifier = flatbuffers::Verifier(buffer, 4096, opts);
```

Listing 22: An insecure way of constructing the FlatBuffers validator

```
1  struct Options {
2    // The maximum nesting of tables and
3    // vectors before we call it invalid.
4    uoffset_t max_depth = 64;
5    // The maximum number of tables we will
6    // verify before we call it invalid.
7    uoffset_t max_tables = 1000000;
8    // If true, verify all data is aligned.
9    bool check_alignment = true;
10   // If true, run verifier on nested flatbuffers
11   bool check_nested_flatbuffers = true;
12   // The maximum size of a buffer.
13   size_t max_size = std::numeric_limits<int32_t>::max();
14   // Use assertions to check for errors.
15   bool assert = false;
16 };
```

Listing 23: Definition of the default validator options

[7] https://en.cppreference.com/w/cpp/language/ub

Data overlap attacks are not detected by the validator at all.

While the validator checks whether string data is null-terminated, it does not verify the content to be valid UTF-8. This is not required in C++ itself since its string data types are just thin wrappers around raw bytes, but it does have an impact for downstream usages. For example, when a received string is logged without care being taken about its encoding, the string being invalid may corrupt the whole log file, since it can't be decoded or processed any further. This is an easy pitfall to fall victim to, as the code in Listing 24 demonstrates. Additional problems arise downstream if the string data is sent further along to other systems that either depend on the data being already verified and valid or just assume that strings are always valid UTF-8.

```
1 void foo(SelfTable *root) {
2   if (root->name() != nullptr){
3     printf("Got name %s\n", root->name()->c_str());
4   }
5 }
```

Listing 24: Logging of a received string that lacks steps to escape potential invalid UTF-8

5.2.3.3 C#

The *C#* language FlatBuffers implementation comes with a validator similar to the one seen in *C++*. It is however not documented in the guides published by the *flatc* authors. It must also be manually run before interacting with a FlatBuffers message, as the normal data access path does not perform much validation automatically. The validation options are shown in Listing 25. They have the same default values as the ones for *C++* from Listing 23. The same considerations regarding DAG-Explosion attacks and validator effectiveness as for *C++* apply.

```
public class Options {
  public Options(
    int maxDepth,
    int maxTables,
    bool stringEndCheck,
    bool alignmentCheck) {
      …
  }

  public Options() {
    max_depth = 64;
    max_tables = 1000000;
    string_end_check = true;
    alignment_check = true;
  }

  …
}
```

Listing 25: The signatures of the C# validator options constructors

Out-of-bounds accesses (from invalid offsets or from an incorrectly sized vector) are still not possible even if no validator is used, because all buffer accesses are wrapped in a specialized `ByteBuffer` class which detects out-of-bounds accesses and throws `System.ArgumentOutOfRangeException` accordingly. These checks can be disabled by conditional compilation of the *flatc* library. Some additional performance optimizations that require memory-unsafe code can also be enabled via the same mechanism. Even so, the defaults are chosen to be secure rather than fast.

In terms of string handling, the *C#* implementation does a good job by offering different kinds of accessors. On one hand, a user can get the raw bytes that are supposed to make up a string if needed but the default accessor properly decodes and validates the underlying data as UTF-8. Invalid data is properly replaced with the UTF-8 replacement character (`U+FFFD`).

Data overlaps are not detected at all.

5.2.3.4 Go

The *flatc* support for the Go programming language exists but is lacking in multiple regards.

To start with, it does not provide any support for validating untrusted messages, so a wide range of exploits is possible. This includes DAG-Explosion attacks, attacks involving out-of-bounds offsets, string manipulations and attribute overlaps.

In detail, supplying a FlatBuffers message with invalid offsets results in the current goroutine[8] panicking at runtime with the message "`panic: runtime error: slice bounds`

[8] A lightweight execution context that is used instead of threads in *Go*

`out of range […:…]`". The same behavior is observed when a vector's length is longer than the whole message and the vector elements outside of it are accessed. This is because the Go implementation uses a `[]byte` instead of raw pointers internally and Go's arrays always include out-of-bounds checks. To reiterate, a go-routine panic occurs every time an offset is invalid, which includes the root offset which is always contained in the very first 4 bytes of a message. An attacker thus does not need any knowledge about the message schema because they can construct an invalid message with just these 4 bytes.

The string handling is not much better. Encoding errors as well as missing null-terminators are silently ignored by the implementation and a `string` is always returned to the user. Go's `string` types are thin wrappers around `[]byte` so this is technically fine but can result in issues on downstream applications as discussed in the *C++* evaluation.

5.2.3.5 Java

The Java implementation is also very minimal when compared to e.g. *C++*. It does not ship with an explicit validator, but due to the way that Java is designed and built, its behavior is slightly less catastrophic. The reason for this lies in better error handling: Instead of panicking the current thread, a number of exceptions is thrown, and error conditions can thus be recovered from much easier. Unfortunately, all of them are *unchecked exceptions*[9] and none are documented in the function signatures generated by *flatc* or mentioned in any documentation material.

In detail: If a server written in Java is supplied with any sort of invalid offset (*InvalidForwardOffset*, *InvalidBackwardOffset* and *InvalidRootOffset*), this results in a `java.lang.IndexOutOfBoundsException` being thrown when the corresponding data is accessed. The same behavior occurs when accessing vector elements whose calculated position lies outside of the buffer (as is the case for *InvalidVecLength* attacks).

String length attacks that result in strings missing their required null-terminator are silently ignored and the implementation simply returns the shorter string as if there was nothing wrong with it. Incorrectly encoded UTF-8 strings on the other hand raise `java.lang.IllegalArgumentException` indicating the "Invalid UTF-8" of the string data.

The lack of validation also leaves the Java implementation completely open to DAG-Explosion attacks of any size. For the same reasons, overlap attacks also not detected.

5.2.3.6 Python

The Python implementation also does not implement a validator. The functionality provided by the implementation also strongly suggests that it was written with a specific usage pattern in mind. For example quality-of-life accessors are provided for using *numpy* arrays to process FlatBuffer vectors but accessing the same data using standard Python iterators, generators or lists is not supported. The *flatc* code generator also

[9] https://www.baeldung.com/java-checked-unchecked-exceptions#unchecked

does not generate any documentation or type hints which makes it hard to reason about what exactly a generated method does or which side effects it might have. This of course includes the different exceptions raised during data access.

Supplying the Python server with buffers that contain invalid offsets (*InvalidForwardOffset*, *InvalidRootOffset*) makes the *flatc* code throw `struct.error` exception when accessing the corresponding data. It also the exception `TypeError` with the error "bad number −84 for type uint32" when supplied with an invalid backwards offset. This is due to the offset calculation being a signed operation that goes below zero while the access into the buffer only supports unsigned or positive offsets. When the Python server is supplied with vectors whose data is longer than the buffer (*InvalidVecLength*), or when an element that would be located outside the buffer is accessed, the same exception is thrown in both cases.

String length attacks are ignored silently by the implementation. It returns the shortened string while ignoring that a null-terminator is missing. The string handling in the implementation can be considered inadequate because it does not handle strings as such. Instead, string data is treated as byte vectors (which is what strings are encoded as in FlatBuffers). Decoding is left to the user; the Python built-in `bytes` data type is used to return these byte vectors as-is. This means that a user of FlatBuffers in Python needs to decode the string themselves. While doing so leaves the flexibility of handling decoding errors to the user, it might also lead to users blindly calling `.decode('UTF-8')` on the bytes since they *"know"* that the data is a string. This of course leads to `UnicodeDecodeError` exceptions being thrown.

Logically, the lack of validation in the implementation allows undetected DAG-explosion attacks of any size as well as overlap attacks.

5.2.3.7 Rust

Rust has one of the more polished FlatBuffers implementations. It generally has good error handling, protects the user against most kind of attacks and has a straight forward API which is easy to use, has automatically generated docstrings and is easy to understand. The only downside is that the documentation on the FlatBuffers website seems to be severely outdated. For example it explicitly lists Rust as not having a validator implemented despite its obvious presence.

Rust is also the only implementation where the validation logic is built into the main accessor that is initially used when accessing the root object of a FlatBuffers. It also provides functions which skip validation but they are explicitly marked as `unsafe` and are delivered with instructions that describe under which circumstances their usage is valid. The default options, which are shown in Listing 26, are generally similar to the verifier options in other languages' implementations. There is a difference though: In Rust, an additional option for `max_apparent_size` is present which governs the maximum amount of bytes that a linearized arrangement of the buffers data graph are allowed to be take up. No option regarding alignment checks is provided. Instead, alignment requirements are always validated. In contrast to other implementations that only

return a boolean value stating whether the buffer is completely valid, a detailed error type is returned, allowing a programmer to debug where and why exactly an input buffer is invalid (See Listing 27 for reference).

The implemented validator is rather effective, since it catches and protects against all tested invalid offset attacks (*InvalidForwardOffset*, *InvalidRootOffset* and *InvalidBackwardOffset*). It also detects strings missing their null terminator (*InvalidStringLength*) as well as incorrectly encoded strings (*InvalidStringEncoding*). Attacks which place vector elements outside the buffer (*InvalidVecLength*) are also caught but overlapping of different data is not (*AttributeOverlap*). The validator protects against DAG-Explosions to a certain degree depending on the options it is configured with.

```
impl Default for VerifierOptions {
    fn default() -> Self {
        Self {
            max_depth: 64,
            max_tables: 1_000_000,
            max_apparent_size: 1 << 31,
            ignore_missing_null_terminator: false,
        }
    }
}
```

Listing 26: Default implementation for the VerifierOptions

```
Error: Could not interpret buffer

Caused by:
  String in range [28, 57) is missing its null terminator.
    while verifying table field `name` at position 20
```

Listing 27: An example error stack that is contained in the Rust validation error

5.2.3.8 TypeScript / JavaScript

The JavaScript/TypeScript implementation is simultaneously the one which performs the least amount of validation but is in the same time also very stable and difficult to exploit.

There is no validation specific to FlatBuffers messages performed by *flatc* and most of the security stems from the languages built-in data types and its tendency to fail softly instead of returning errors or throwing exceptions. For example, *flatc* implements a wrapper called `ByteBuffer` for accessing the content of a `Uint8Array` that stores a FlatBuffers encoded message. This wrapper provides utility methods for decoding parts of the array as more complex data types like `Int32`. These utility methods along with JavaScript's tendency to coalesce data types result in invalid reads from the underlying array being returned as `0` which in turn makes the FlatBuffers related code interpret many things as being zero-sized or having their default value. So when an attack is

targeting a server written in TypeScript/JavaScript that utilizes invalid offsets (*InvalidForwardOffset*, *InvalidRootOffset* and *InvalidBackwardOffset*) the server is mostly unaffected and interprets as having data having their default value.

When specifying vectors that are larger than the whole buffer (*InvalidVecLength*), the same behavior manifests. So when elements placed outside the buffer are accessed, *flatc* returns that element's default value. This behavior is technically safe in that it does not result in the program unexpectedly terminating or undocumented exceptions being thrown but depending on what the server does with vector elements, it may be manipulated into doing many calculations for elements that were fabricated this way. For example, the algorithm implemented by the test server (see Listing 12) calculates a sum over all elements of a vector. It can be tricked this way into calculating the sum of up to `2147483647` (`i32` maximum value) zeros. As Figure 7 shows, a considerable amount of CPU time is exhausted this way. The marker on the figure indicates the time at which the attack was started. We can see that from that point on, the server used CPU-time in a 1-to-1 ratio to realtime (equaling 100% CPU usage on one core) for about 17.5 seconds. The attacking client on the other hand only used about 10 milliseconds (not shown) and the attack payload was only 64 bytes large.

One detail to note is that JavaScript does not actually support unsigned integers. This means that any time the FlatBuffers protocol specifies an unsigned integer to be used, when that integer has its first bit set, JavaScript will instead interpret it as a negative number. This behavior is not immediately exploitable because of the behavior described in the previous paragraph but should still be protected against by validating numbers to not be negative when an unsigned integer is expected, which *flatc* does not do.

The implemented string handling is similarly permissive. It correctly replaces invalid UTF-8 bytes with the replacement character (`&#xFFFD;`) (*InvalidStringEncoding*) and ignores missing null terminators (*InvalidStringLength*).

Unfortunately, having no validator logic leaves the implementation completely vulnerable to DAG-Explosion attacks as well as data overlaps.

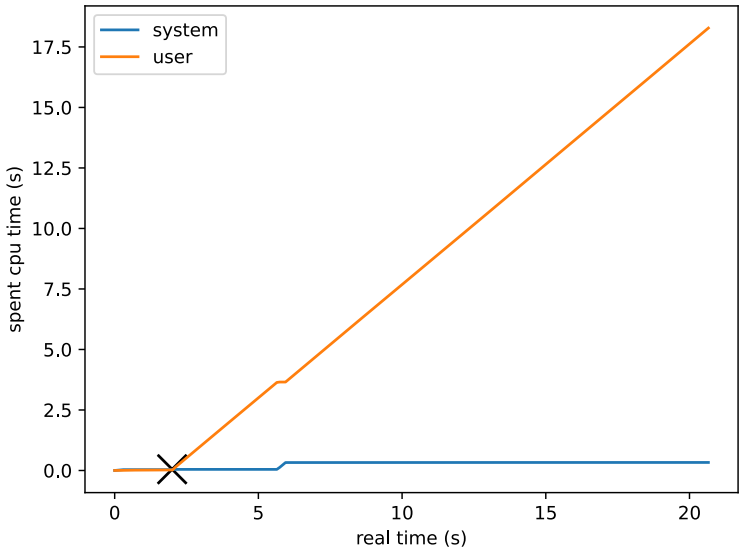


Figure 7: A graph of the CPU time used by the TypeScript server when attacked with an impossibly large vector.

5.2.3.9 Summary

The findings from the previous sections can best be summarized by saying that there are clearly two classes in terms of quality of implementation. There are those languages which have proper validation logic implemented (*C++*, *C#* and *Rust*) that provides a decent degree of protection against untrusted inputs. Then there is the other class of implementations, which are only partially protected because of the respective language's intrinsic properties (*Go*, *Java*, *Python*, *TypeScript* and *JavaScript*). Error handling is also very basic if not completely absent in all languages except *Rust* and a wide range of undocumented internal exceptions or complete program panics which are not documented can often be provoked by external attackers. Documentation as a whole is sometimes missing or outdated and often leaves out important details.

Table 2 lists the results of different experiments in a comprehensive table. All language implementations are rated against all tested attacks to be either

- *protected* which is to say that it behaves in a predictable and clearly documented manner when being presented with a malicious message or
- *vulnerable* meaning that the implementation crashes, throws undocumented exceptions or otherwise behaves in a way that is wide open to exploitation.
- *mitigated* for when the provided validation mechanisms are able to reduce the impact of an attack but not completely negate it.

In this presentation of results, the split between the two classes of implementation quality becomes clearly visible. It should be noted that *TypeScript/JavaScript* is an outlier should not be considered secure despite avoiding most of the tested vulnerabilities on first glance. For details see the specific section (Section 5.2.3.8).

	C++	C#	Go	Java	Python	Rust	TypScript/ JavaScript
DAG-Explosion	m[1]	m[1]	v	v	v	m	v
InvalidString-Encoding	p[2]	v	p[2]	v	v	p	p
InvalidString-Length	p[1]	p[1]	p[2]	p[2]	p[2]	p	p[2]
InvalidVec-Length	p[1]	p[1]	v	v	v	p	p[2]
Invalid-ForwardOffset	p[1]	p[1]	v	v	v	p	p[2]
InvalidBack-wardOffset	p[1]	p[1]	v	v	v	p	p[2]
Attribute-Overlap	v	v	v	v	v	v	v

Table 2: Comparison of the results for FlatBuffers implementations in different languages. Vulnerable implementations are marked *v*, protected ones *p* and the ones where attack impact is mitigated with *m*.

[1]: When using the optional validator
[2]: Not vulnerable itself but downstream consumers might be affected

5.3 DAG-Explosion DoS impact

Section 4.3.2 explains the possible attack vector of using FlatBuffers encoding features in a way that makes a very small message contain a very large data tree. This section details the results obtained found when executing such an attack against different servers using the schema and algorithm described in Section 5.2. The results were collected using the instrumentation tool introduced in Section 5.2.2 and include data for languages that ship with a validator as well as ones that don't. The following subsections each focus on a different property of the attacked server.

5.3.1 Memory Impact of Tree-Walk

In this experiment, the memory usage of the same algorithm implemented in different programming languages (as described in Section 5.2.1) are evaluated. Here the focus lies on the amount of program memory required to process FlatBuffers messages containing DAG-Explosions.

The first experiment was performed with a DAG tree depth of 7 and a vector length at each level of also 7. This was determined to be the parameter combination which produces the largest DAG-Explosion but still passes *flatc* buffer validation with default parameters. Figure 9 shows the amount of memory needed by the different implementations for processing one such FlatBuffers message. The upper plot displays how much memory was allocated to the process in total during its runtime while the lower plot shows the applications memory usage over time. It is centered with $x = 0$ around the point in time at which the malicious message was sent to the applications.

The implementations in *C++* (hidden almost underneath *Rust* in the plot), *Rust* and *Go* all have a very small runtime overhead when idling, which is reflected in the plot. Upon receiving the FlatBuffers message, no relevant amount of additional memory is required to process it, which is unsurprising considering that the data accessors provided by *flatc* in these languages are all thin wrappers around byte slices, or pointers respectively. The implemented algorithm is also simple enough so that all those languages' implementations are able to use stack-frame memory only, which results in the allocated data accessor objects to be freed as soon as they are no longer needed without having to wait for garbage collection[10]. The *Python* implementation also falls in this category as even though *Python* has a larger general runtime overhead, *flatc* uses *numpy* in its implementation so that the user receives only stack allocated and reference counted data structures that don't require garbage collection. This is also visible in the plot as the *Python* server consumes only a negligible amount of additional memory for processing the message.

In opposition to the language implementations discussed before, the implementations in *Java*, *TypeScript/JavaScript* and *C#* can all be seen allocating additional memory for walking the received data-tree. While the *C#* implementation does not require significant amounts, the *Java* and *TypeScript/JavaScript* implementations can be seen to allocate more than double their initial runtime overhead just for processing one single message. In the case of *Java*, the message is even processed faster than the garbage collector is able to free memory that is no longer used.

To evaluate further how much memory usage an implementation will peak at, another experiment was conducted using a DAG-Explosion message of depth 12 and vector length 12 while capping the experiment duration at 5 minutes. That time limit was chosen to provide enough data to evaluate the implementations memory usage as none was able to completely process the buffer inside this time frame. Figure 9 shows the relevant time span of this experiment. *Java* is by far the worst performer with allocating 190 MBytes before its garbage collector is able to catch up by cleaning up unused objects. *TypeScript/JavaScript* memory requirements cap much earlier at around 55 MBytes. Nevertheless, when considering that the attack payloads are only one to two KB large, this is an absurd amount of memory to process it.

[10]Yes, even in *Go*, the garbage collector is bypassed in certain situations: https://go.dev/doc/faq#stack_or_heap

It is possible that the garbage collected languages might behave differently when the system is under memory pressure and collect garbage more aggressively, but these conditions should not be considered favorable, as more processing time would be spent collecting garbage. Additionally, some systems, e.g. Linux, don't behave optimally under high memory pressure or decide to kill the processes responsible for using too much memory. As demonstrated by the experiments, it would be trivial to cause such memory pressure conditions, as the memory requirements shown here apply to the processing of a single request, while a truly malicious actor can easily send many such messages, each requiring memory for processing. Another aspect to consider is that the measurement tool is not able to deeply inspect programming languages whose runtime include a virtual machine (e.g. Java). Those VMs may allocate memory from the operating system while internally not always having all their allocated memory in use.

In summary, DAG-Explosions can be effective in cause a system-under-attack to allocate significant amounts of memory just for processing the message itself. Exactly how much and whether that is a security risk depends on which system is under attack.

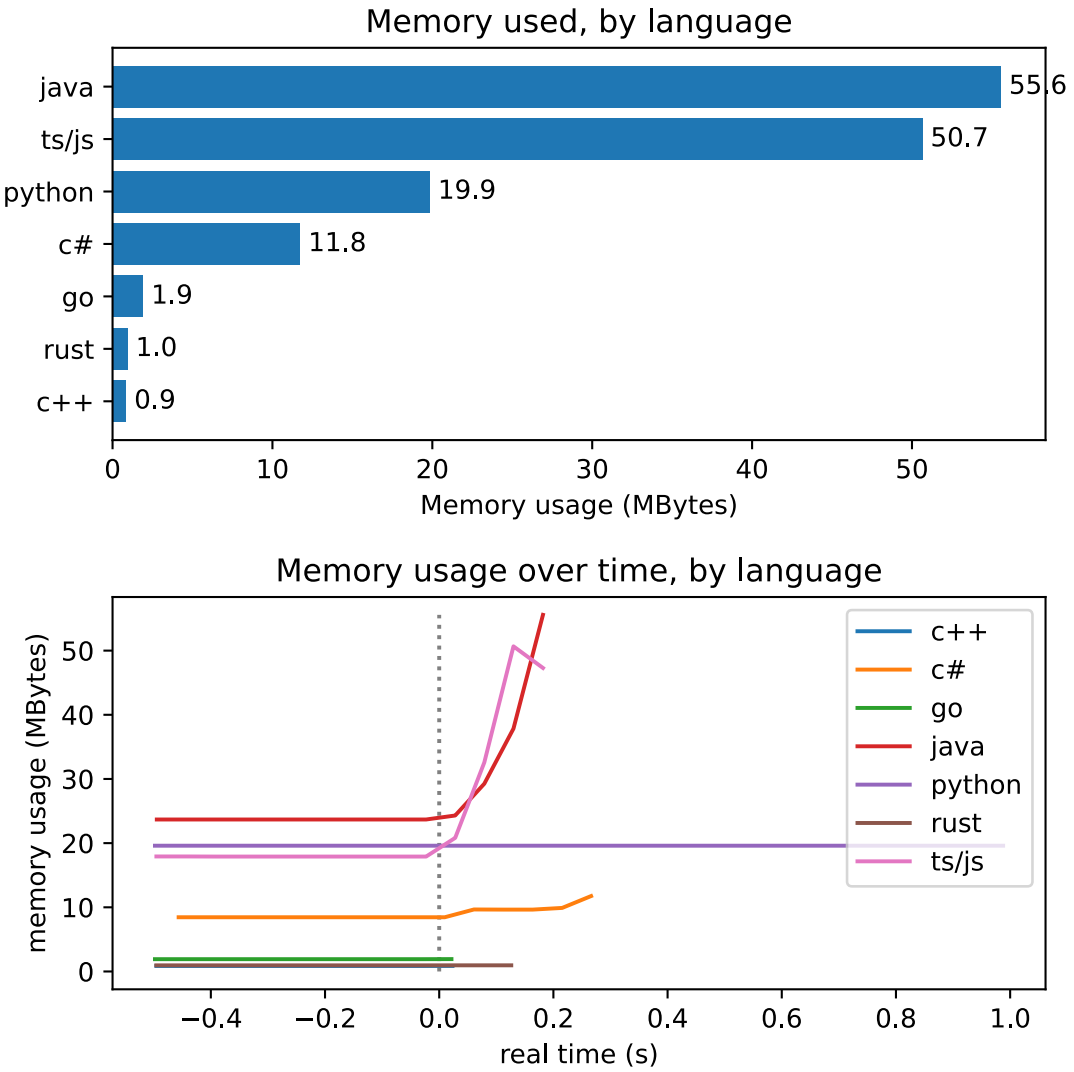


Figure 8: Memory used by different servers when encountering a DAG-Explosion of depth 7 and vector length 7

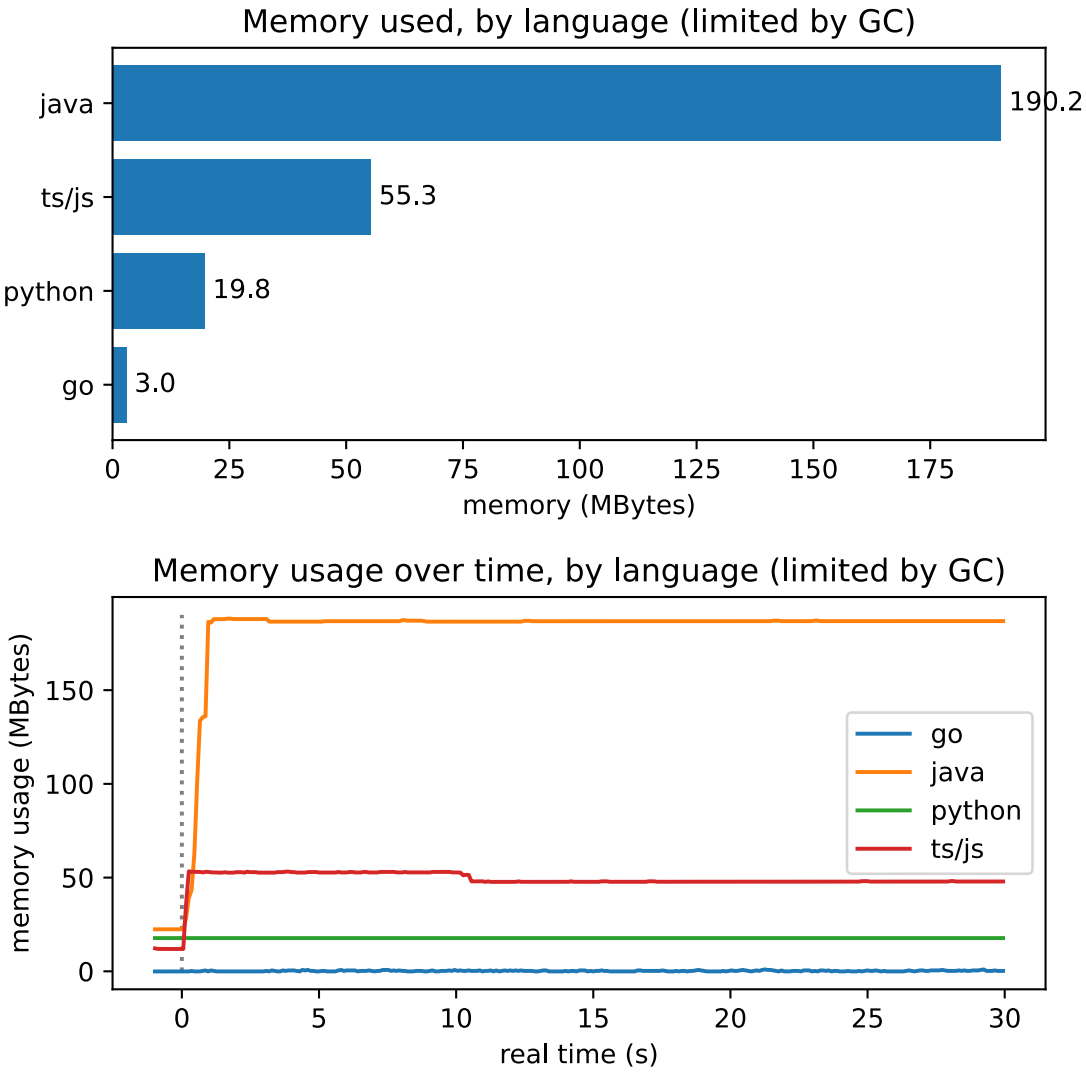


Figure 9: Maximum amount of memory ever used, limited by garbage collection

5.3.2 CPU Impact of Tree-Walk

This experiment uses the same setup as described in Section 5.2.1, but it focuses on the CPU usage observed by each programming language's implementation when encountering DAG-Explosion payloads.

For this purpose, two types of experiments were conducted:

- **Experiment A**: Different payloads which all pass the *flatc* validation when it is instantiated with default options were generated and evaluated for their effectiveness on servers implemented in different programming languages.

  The largest payload which was still able to pass validation used a depth of 7 and vector length of 7. The results from that experiment are shown in Table 3 on a

logarithmic scale. The figure plots how much a language's server took in real time on the X-Axis against how much CPU time was used by that computation on the Y-Axis. Markers on this plot tend towards the 45° line as that indicates 100% CPU utilization. If a process is below that line, it does not utilize the CPU to its full potential, probably due to waiting on external factors like I/O while being placed above the line is only possible if the server uses multi-threading which uses more than one CPU core at a time.

As expected, the compiled languages *C++* and *Go* are very fast in processing the message compared to the others. In fact, they are so fast that they process the request faster than the instrumentation tooling takes measurements, so their CPU utilization between the start and end of the measurement interval - and thus in the plot - is reported lower than it actually is for the request processing time. What is evident is that the *C++* and *Go* implementations processed the requests in less than 40 milliseconds. It should be noted though that since it is not possible to do validation in *Go*, a much bigger payload can still cause the *Go* implementation to require a lot of CPU time (see experiment B).

The next cluster of languages includes *Rust*, *TypeScript/JavaScript*, *Java* and *C#*. These process the message in less than 0.5 seconds, which might already be considered to be inside an exploitable range depending on the specifics of the application and attack scenario. *Python* is the definitive outlier in this experiment as it required 8.1 seconds to walk the tree of message data.

The reason *Rust* is not as fast as *C++* or *Go* is that the implementation in *Rust* does more validation. After validating the whole message using the dedicated validation algorithm, it also does out-of-bound checks on every field access, which is due to how *Rust* works internally. *C++* does not perform an out-of-bounds check on each access and *Go* does not perform the initial validation. This explains why the *Rust* implementation is half as fast as the other two, considering it does twice as much validation.

- **Experiment B**: An extremely large DAG-Explosion payload was sent to servers which don't use a validator with the goal of gauging the efficiency with which they are able to utilize the available CPU resources. The payload consisted of a depth of 12 as well as a vector length of 12.

  The results of this experiment are shown in Table 3. All implementations utilize the system's CPU nearly fully or in the case *TypeScript/JavaScript* even with a slight overhead. This result is expected since accessing parts of a FlatBuffers message is not I/O-bound and the *flatc* library does not need to call into any external systems to e.g. acquire additional resources.

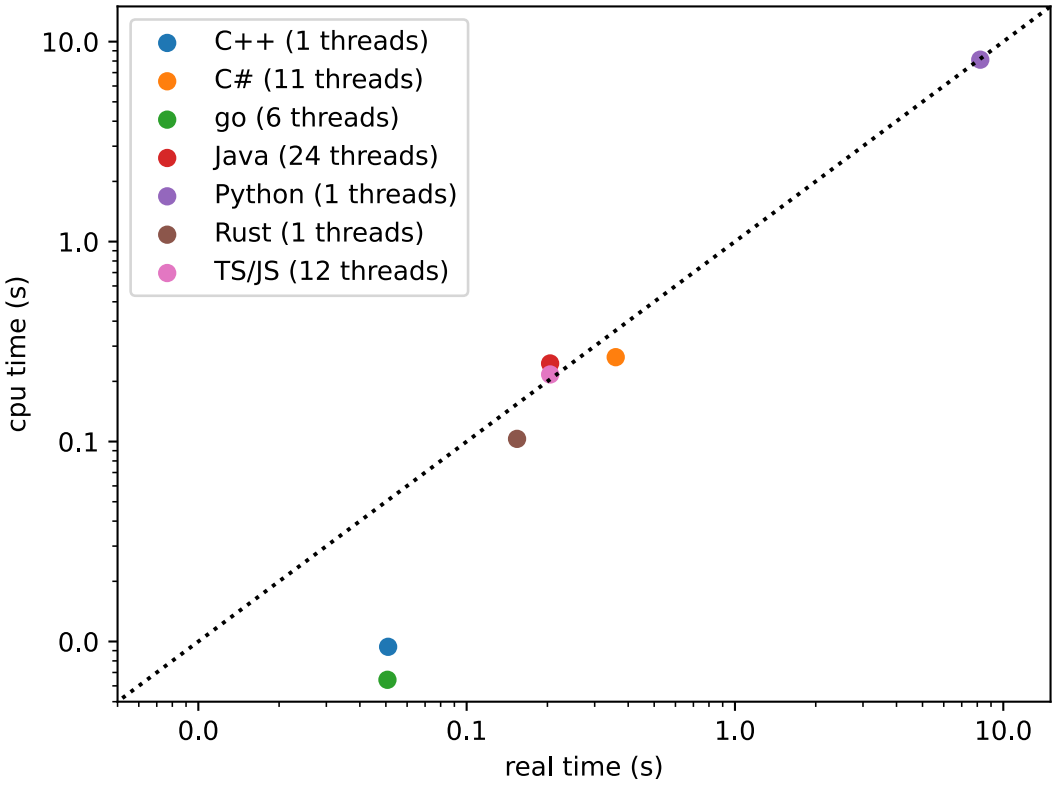


Figure 10: A system's CPU time compared against real time for walking the data tree of a DAG-Explosion payload with depth 7 and vector length 7

Language	**CPU Utilization**
TypeScript/JavaScript	102%
Java	99%
Go	99%
Python	99%

Table 3: CPU utilization efficiency of implementations in different programming languages during long running operations

In summary, DAG-Explosion attacks are capable of exhausting a systems CPU resources with very small payloads very effectively. An attacker only requires one payload per CPU core as the processing of FlatBuffers messages is happening at (near) 100% CPU utilization. The processing speed and thus the effectiveness of the attack varies largely between server implementation language but most languages can be considered vulnerable when taking in to consideration that an attacker needs to generate attack payloads only once and can then send them at line-speed of the transport layer.

5.3.3 Memory Impact of Linearization

This experiment was designed to precisely evaluate the behavior of a system which linearizes a FlatBuffer message when it encounters a DAG-Explosion payload. The term linearization in this context refers to the act of unpacking all references and storing them inline to where they are referenced.

The experiment was set up using a TCP server that receives a FlatBuffers message in the schema shown in Listing 11. This server then parses and converts the received FlatBuffers message to JSON. JSON was chosen as a conversion format because it is commonly used e.g. when servers log incoming requests using structured-logging. The specific choice of data interchange format has only minimal bearing on the outcome of the evaluation. The generated JSON is not actually logged anywhere but kept in memory for a small amount of time to give the instrumentation tool a chance to track memory usage. The final size of the JSON data is logged to allow evaluation of the size increase from initial FlatBuffers message to JSON equivalent. The conversion itself is done by writing the buffer to a file and calling the *flatc* tool which has JSON conversion built-in in a subprocess. Afterwards the JSON file generated this way is read back into the server. This was necessary because of a bug in the *flatc* code generator that surfaced when it was asked to generated `Serialize/Deserialize` implementations for a schema.

The described server was then presented with different FlatBuffers messages that contained DAG-Explosions of different depths and sizes (see Section 5.2.1). The experiments were all successful in generating a much larger linearized representation than the original buffer and only differed in the exact measurements. For brevity, only two experiment arrangements are explained here in detail:

- **Arrangement A**: DAG-Explosion with **depth 7** and **vector length 7** which is the largest payload that still passed validation when using default validator options.
- **Arrangement B**: DAG-Explosion with **depth 8** and **vector length 8** which was the largest possible experiment as others required more than the 32GB of system memory available for the experiment.

Running the experiment resulted in the following data:

- **Arrangement A**: The attack was very successful. The generated payload was only 1072 bytes large as a FlatBuffers but resulted in a roughly 775.91MB large JSON-string when converted. Using the exact sizes, the JSON-string is larger by a factor of $723,801$.
  The conversion itself required even more resources as can be seen in Figure 11. It needed 1.51GB of memory at its peak as well as 9.36 seconds of CPU-time.
- **Arrangement B**: This attack generated a payload 1240 bytes in size but *exploded* to a JSON-string that required 16.74GB to be stored. The scale-up in this case is a factor of $13,498,534$. As with the *a* setup, the conversion process required more resources than just storing the result itself which is also shown in Figure 11. This time, 29.59GB were used in 139.94 seconds of CPU time and 262.14 seconds real time.

Both results have in common that CPU utilization drops drastically at a certain point in the experiment. The reason this happens is that the *flatc* tool is instructed to write its JSON result into a file and the kernel's I/O write-cache is eventually filled up by the large data. *flatc* then needs to wait for file write operations to be committed to disk which is considered I/O-wait and not counted towards CPU-time.

Another shared result is that the total amount of memory required is in the magnitude of being double the resulting JSON-string. It is likely that multiple copies of the same data are held at different positions in memory, presumably once in userspace (*flatc* and the experiment server) and once in kernelspace where it was copied for I/O operations.

As a side-note, *flatc* itself does not seem to run any validation logic or at least disables checks regarding maximum buffer sizes and restrictions regarding the amount of tables allowed in a FlatBuffers message. Otherwise, the b experiment would not have been possible because it produces more data than those checks allow with their respective default settings.

In summary, the results prove that DAG-Explosion attacks are very much a threat to applications that blindly accept untrusted input messages and convert them into another data format which breaks the reference-heavy compression dynamics of FlatBuffers. The required memory for such a conversion is very much exponential when related to the attacking payload size, which makes attacks against such servers extremely effective. This attack works against applications that perform no validation as well as those which use the built-in *flatc* validator with its default options. Even with application specific validation options, the underlying problem still remains and its effects can only be minimized, not completely averted.

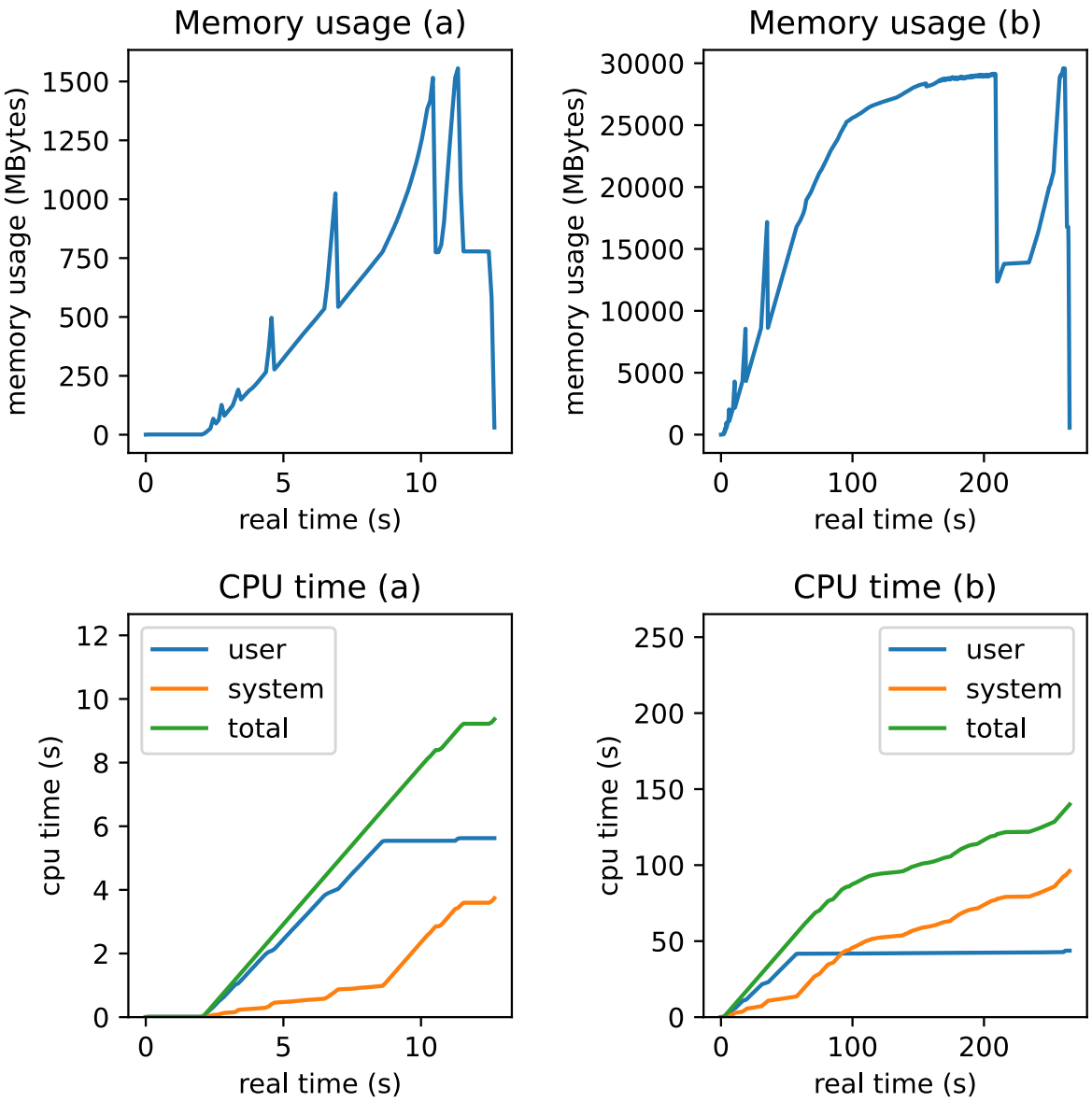


Figure 11: Resource usage when a DAG-Explosion of depth 7 and vector length 7 (a) and 8/8 (b) is converted to JSON

6 Discussion

The discussion section aims to summarize the evaluation results and provide an analysis of the implications and effectiveness of the methods and attacks discussed in this thesis. Afterwards, the relevance to real-world systems as well as broader system design implications including improvement suggestions are presented in Section 6.2.

Generally speaking, the research questions presented in Section 3.2 can be answered like this:

1. **Under which circumstances can what attacks be successfully executed?**
   Using FlatBuffers often requires knowledge about a targets message schema to successfully craft exploits such as DAG-Explosions. In other cases though, a system is vulnerable to attacks regardless of the used schema (e.g. *Go* with *InvalidRootOffset*).
2. **What impact does which attack have on the system-under-attack?**
   Different attacks were shown to have a wide range of impacts (see Table 4 for details) that highly depend on the configuration of the system-under-attack. Those can range from causing no harm to completely crashing a target application.
3. **What is the relation between resource requirement on the attacker's side as compared to the effects on the system-under-attack?**
   Factoring out the often required initial effort of determining a targets message schema, all examined attacks can trivially be constructed and require neither any noteworthy amount of CPU-time nor memory.
4. **Which relevance do the findings of this thesis have to real-world applications?**
   Every examined attack was demonstrated to be practicable. A number of source-available application schemas from GitHub were also examined and determined to be risking DAG-Explosion attacks. This includes high-profile data processing libraries such as tflite and *Apache-Arrow* of which only the latter implements appropriate defensive precautions.

6.1 Vulnerabilities

The performed examinations and experiments have demonstrated significant security risks associated with handling untrusted FlatBuffers encoded messages. The findings of this thesis align with and extend existing research on the vulnerabilities of serialization protocols. The literature reviewed in Section 3.1 consistently emphasized the criticality of proper validation and input handling, which this thesis further proves. One notable advance is the focus on protocol-level attacks inherent to the design of the FlatBuffers encoding format, such as DAG-Explosion attacks. These types of attacks expand the awareness of how serialization formats can be exploited beyond simplistic sanity-checks and highlight the importance of robust validation routines and application-level safeguards.

The ability for malicious actors to craft such DAG-Explosion attacks which exponentially increase in complexity during traversal or linearization poses a significant DoS risk. While creating them requires knowledge about a system's FlatBuffers schema,

those schemas are often known and reverse engineering such a schema was also shown to be possible. This type of attack is especially dangerous for systems that perform deep recursive data processing or convert the message into a different format. The exponential memory allocation (up to ~30GB) and CPU utilization (100% for extended periods) for a relatively small payload of only one to two kilobytes indicate that these vulnerabilities can lead to severe service disruptions with minimal attacker resources.

Furthermore, the implementations in different languages show great variance in their quality and provided security assurances. In some of these implementations, *flatc* does not provide a validator, which means the user cannot perform message validation or has to do so manually on their own. These can be trivially exploited and be made to crash or otherwise fail unexpectedly. Doing so often requires no further knowledge on the attacker's side, as only 4 bytes of buffer can be enough regardless of the system's schema.

The handling of string data is also inconsistent between all implementations. Because the FlatBuffers specification states that strings must be encoded as UTF-8, some language's implementations blindly expect them to always be valid UTF-8 while others don't enforce this, which naturally leads to incompatibilities. In some cases, *flatc* simply ignores those encoding errors, in other cases, accessing the data causes the system to crash while only a few implementations perform the ideal handling with symbol replacement. All but the latter can have impacts on the system itself and others interacting with it.

The FlatBuffers protocol's reliance on offsets rather than inline data makes it also extremely susceptible to data overlap attacks in many different ways. Those can lead to a single byte sequence being interpreted as different types. The experiments showed that such constructions work and are not effectively handled by any implementation. Related work suggests that this might be explicitly out-of-scope of the implemented validator. In any case, the possibility of such an attack makes it completely and absolutely unsafe to ever modify FlatBuffers data in place when there is the slightest possibility that the buffer comes from an untrusted source. Nonetheless, the *flatc* command-line tool includes the option `--gen-mutable` to "Generate accessors that can mutate buffers in-place" without even so much as a warning about the risks associated with this.

These risks and vulnerabilities are not merely theoretical. The experiments conducted demonstrate that the risks are real and affect real-world applications. This argument is strengthened by the analysis of freely available FlatBuffers schemas on GitHub which show signs of being vulnerable to a wide range of attacks. For example, the *tflite-micro* project which is actually authored by the same company as *flatc* (Google) was not found to use any validation for its FlatBuffers based model files.

Different attacks have been presented, all of which impact a target system in different ways. Table 4 contains a detailed overview about the kinds of impacts a user can expect when using FlatBuffers in different environments and configurations. The following fault categories are examined:

- **Access Violation**: An attacker can cause a receiving system to access arbitrary memory regions.
- **Memory Load**: An attacker can cause a system-under-attack to allocate large amounts of program memory, even exhausting all available resources. Naturally, any processing of requests induces some memory load, despite that, a configuration exhibiting this behavior uses more memory than is normally expected for processing a singular request.
- **CPU Load**: An attacker can cause a system-under-attack to perform large amounts of processing on the CPU thus preventing other programs from utilizing it. Similarly to the *Memory Load* category, any request processing is expected to use CPU time during request processing, however, this category indicates that an implementation can be made to use more than an acceptable amount.
- **Program Crash**: An attacker can cause an irrecoverable fault which completely stops the target system.
- **Thread Panic**: An attacker can cause the thread which processes the received message to panic.
- **Exception**: An attacker can induce an (undocumented) exception to be thrown.
- **Undefined Behavior**: An attacker can manipulate the target system into a state where further computation results are not clearly defined. Note that this is not a categorization where the effects of an attack could not be determined but instead, the attacked system can be made to perform operations that violate the rules of its execution environment[11].

The programming languages and configurations displayed in Table 4 have been classified to either manifest these faults fully, partially (e.g. when a validator can be used for mitigation) or not at all. Evidently, the *Rust* language's implementation is unsurpassed by any other.

[11]See https://doc.rust-lang.org/reference/behavior-considered-undefined.html or https://doc.rust-lang.org/reference/behavior-considered-undefined.html

	C++		C#		Go	Java	Python	Rust		TS/JS
Validator	on	off	on	off	n/a	n/a	n/a	on	off	n/a
Access Violation	no	yes	no	no	no	no	no	no	no	no
Memory Load	no	no	semi	yes	no	yes	no	no	no	yes
CPU Load	no	yes	semi	yes	yes	yes	yes	no	yes	yes
Program Crash	no	yes	no	no	no	no	no	no	no	no
Thread Panic	n/a	n/a	n/a	n/a	yes	n/a	n/a	no	yes	n/a
Exception	no	no	yes	yes	n/a	yes	yes	n/a	n/a	yes
Undefined Behavior	yes	yes	no	no	no	no	no	no	yes	no

Table 4: Comparison between the behaviors that different implementations may exhibit when attacked. Different system setups are categorized into whether they exhibit certain undesirable behaviors (yes), whether they don't (no), or only partially exhibit it (semi). If a categorization does not apply, it is marked as n/a. The default or recommended validation is marked in purple.

6.2 Improvement Suggestions

Given the severity of the vulnerabilities and practices discovered while analyzing different application's usages of FlatBuffers, several recommendations can be derived:

- **Enforce Validation**: Any application using FlatBuffers, if it is written in *C++*, should mandatorily utilize the provided validation routines. The validation settings (e.g. `max_table` and `max_depth`) should be tightened based on specific application contexts to reduce and mitigate the impact of DoS-attacks. In most if not all cases, there are no downsides to using the validator since there is no large performance cost in running it and the security gained through it protects against a large range of attacks. As a consumer of *flatc*, changing its API is not possible. In that case, linting rules could be introduced to warn when an unverified buffer is used.
- **Improve *flatc* API Design**: As emphasized in Section 3.1.5, secure defaults and API design decisions are important in enabling users to build secure applications. Documentation and APIs should make using them securely the path of least resistance for developers. This would however require the *flatc* authors to break their existing API so that using the validator becomes the default while not using it is converted to an explicitly unsafe operation, like they have done with the *Rust* API. Of course, every programming language supported by *flatc* would need to have a validator implemented in the first place, which should be considered crucial functionality.

The default validator options, especially those concerning the amount of data contained in a FlatBuffers message, would also profit from refinement work. Currently, the *flatc* authors err on the side of producing software that *just works* in as many scenarios as possible, but the chosen default size limits may already present a DoS surface. The technique used by *Apache Arrow* in which a FlatBuffers size limiting validator options are related to the size of the encoded buffer should be incorporated into *flatc*. That way, large messages are still permitted as long as they actually contain relevant data while exponential DAG-expansion is not.

- **Update and Improve Documentation**: Comprehensive and up-to-date documentation is critical. It is negligent to only mention security topics like validation in the advanced language-specific topics. Instead, the prominent examples shown in the *flatc* tutorial should, if not always show then at least explicitly mention buffer validation and handling of untrusted input.

  The documentation included in the code generated by *flatc* for a target language should also be improved to at least include inline docstrings. These docstrings should explain what a function does, explain under what conditions it is safe to use and in which ways it fails when those conditions are unmet (e.g. throwing an exception vs. returning `null`). The typing information of the generated code should also be updated to include that information whenever possible.

- **Memory and CPU Usage Monitoring and Limiting**: Systems should generally incorporate robust monitoring and possibly circuit breakers to detect and halt abnormal resource usage patterns indicative of attacks. All container-based deployment tools support assigning of resource limits to their workloads. While this does not eliminate the underlying problems, it can mitigate the risks associated with them and is generally considered good practice.

- **Careful Downstream Usage**: FlatBuffers should only very carefully be reused and sent to other (downstream) applications. Section 5.2 illustrates that different languages and systems behave differently when encountering certain malicious FlatBuffers. Just because one system can process a buffer, another system written in a different programming language or doing different things with it might still crash or behave unexpectedly. If data must be passed onward, one should consider re-encoding it so that it is structured in a known way.

  It should also always be avoided to mutate FlatBuffers data in-place. Through different kinds of data overlap attacks, this can cause almost arbitrary mutation of other bytes in a buffer as a side effect. A previously cautiously verified buffer can become invalid and cause many unexpected issues.

In summary, while the FlatBuffers encoding format provides benefits in terms of encoding and access speed, it should always be used carefully and with an understanding of the impacts certain access patterns have on an application's attack surface.

7 Limitations and Future Work

The research shown in this thesis provides valuable insights, however there are limitations worth acknowledging.

The evaluation focused on selected common programming languages. Further research could expand this to other languages and alternative implementations like *flatcc*. The existing experiments could also be further differentiated and subdivided. For instance, examining the effects of turning off the validator in languages where it is implemented would yield additional data. Different allocator implementations (e.g. running *TypeScript/JavaScript* with bun's `--smol` flag) could also be included in further experiments. These additional experiments are not expected to produce radically different results but would be interesting for painting a complete picture.

Regarding a complete picture, experiments that test different implementations using incorrectly aligned data or technically invalid enum values were also not performed. Unaligned data is expected to be caught by validation while the FlatBuffers documentation strongly suggests invalid enum values to not be detected. Nonetheless, the behavior of applications encountering these issues would be interesting to know. Another, very specific way of triggering out-of-bounds data access is to construct a vector whose `vector_len * size_per_element` calculation overflows. This might fool a validator into treating a buffer as valid while accessing elements of that array could produce out-of-bounds accesses.

Additionally, the interaction of FlatBuffers with other serialization mechanisms could present different or compounded vulnerabilities that are not explored in this study.

The work affecting vulnerable schema files on GitHub could also be included as the tools used in this thesis were only able to analyze singular `.fbs` files which do not import definitions from other schema files. The same tool also only checked for very specific ways of constructing DAG-Explosions. More and different construction methods could be integrated into the analysis tool to more accurately detect schemas that allow such attacks and that were currently not successfully detected.

8 Conclusion

This thesis has provided an extensive analysis of the FlatBuffers encoding protocol, highlighting its security implications and inherent vulnerabilities. Through a structured approach, various aspects such as schema reverse-engineering, protocol attack vectors, including DAG-Explosion and Data Overlap attacks, and the practical impact of these vulnerabilities have been explored. The findings demonstrate significant risks associated with handling untrusted FlatBuffers encoded messages. Malicious actors can craft messages that pose significant DoS risks. The inherent design flaws in the protocol, such as dependency on offsets and lack of built-in comprehensive validation, leave systems open to severe exploitation.

The comparative evaluation of different programming language implementations of FlatBuffers reveals substantial discrepancies in their robustness and security measures. While the implementations in languages like *Rust*, *C++*, and *C#* offer validation mechanisms that can mitigate some of these risks, others like *Go*, *Java*, *Python*, and *TypeScript/JavaScript* present more severe vulnerabilities due to the absence of such counters and their general failure to gracefully handle errors.

Different attacks were demonstrated to show substantial effectiveness in exploiting these vulnerabilities, leading to exponential memory and CPU resource consumption, which could drastically impair the targeted systems while requiring only minimal attacker resources. Other attacks significantly increase the risk of ambiguous data interpretation, leading to potential unpredictable behaviors and security lapses stemming from unsafe in-place modifications.

Several recommendations emerge from the research to enhance security when using FlatBuffers:

1. **Mandatory Validation**: Enforce rigorous validation, especially for C++, using routines built into *flatc*, and ensure that these are well-documented and integrated into the usual data handling workflows.
2. **Improved API Design**: Revise the FlatBuffers API to adopt secure defaults, explicitly marking unsafe operations, and integrate validator usage clearly within the default workflow.
3. **Comprehensive Documentation**: Update and enhance FlatBuffers' documentation to prominently feature security practices and failure conditions, ensuring developers are aware of and can easily implement protective measures.
4. **Resource Monitoring**: Implement robust monitoring and limiting of memory and CPU usage to detect and mitigate abnormal patterns indicative of attacks.
5. **Cautious Downstream Use**: Always avoid in-place modifications and carefully manage the transition of FlatBuffers data to downstream systems to prevent cascading vulnerabilities.

There is however still open research which could improve the FlatBuffers security landscape such as extending the analysis to additional programming languages and alternate implementations, exploring other attack vectors such as alignment issues or overflow conditions, and improving tooling for detecting and preventing vulnerable schemas.

In conclusion, while FlatBuffers offers numerous advantages for fast and efficient serialization, its proper and secure implementation requires a careful approach, robust validation mechanisms, and comprehensive developer awareness to mitigate potential security risks.

Appendix

Glossary

***API* – Application Programming Interface**: A software interface over which two or more computers or components interoperate.

***CGroup* – Control-Group**: A linux kernel feature that allows limiting and accounting of a collection of processes.

***DAG* – Directed-Acyclic-Graph**: A type of directed graph whose nodes don't form a closed loop.

***DoS* – Denial-of-Service**: A type of attack whose goal it is to reduce the availability of a target system.

FlatBuffers: The binary encoding and data transfer protocol whose inherent security and implementation quality are the topic of this thesis.

***JSON* – JavaScript Object Notation**: An open standard data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays.

***ProtoBuf* – Protocol Buffers**: Another language-neutral, platform-neutral extensible mechanism for serializing structured data similar to FlatBuffer in many aspects.

***TCP* – Transmission-Control-Protocol**: One of the main protocols of the Internet protocol suite which provides reliable, ordered, and error-checked delivery of a stream of bytes between networked applications.

flatc: The main FlatBuffer implementation by Google.

flatcc: The C implementation for FlatBuffer done by dvidelabs.

sysfs: A pseudo-filesystem which exports information about the state of various kernel objects to user-space through virtual files. Usually mounted under /sys.

***tflite-micro* – Tensorflow Lite for Microcontrollers**: A mobile library for deploying machine learning models on microcontrollers and other edge devices.

Bibliography

[1] "FlatBuffers." Accessed: Oct. 17, 2023. [Online]. Available: https://flatbuffers.dev/

[2] "JSON Schema." Accessed: Aug. 13, 2024. [Online]. Available: https://json-schema.org/

[3] "Protocol Buffers." Accessed: Aug. 13, 2024. [Online]. Available: https://protobuf.dev/

[4] "W3C XML Schema Definition Language (XSD) 1.1." Accessed: Aug. 13, 2024. [Online]. Available: https://www.w3.org/TR/xmlschema11-1/

[5] "gRPC." Accessed: Aug. 13, 2024. [Online]. Available: https://grpc.io/

[6] S. S. Laurent, J. Johnston, E. Wilder-James, and D. Winer, *Programming Web Services with XML-RPC: Creating Web Application Gateways*. O'Reilly Media, Inc., 2001.

[7] T. Greifenberg *et al.*, "A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages," in *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2015.

[8] "6.64.10 Structure-Layout Pragmas," *Using the GNU Compiler Collection (GCC)*. Accessed: Aug. 13, 2024. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/index.html

[9] "2.6 Load and Store Instructions," in *The RISCV-V Instruction Set Manual Volume I: User-Level ISA, Document Version 20240411-draft*. Accessed: Aug. 13, 2024. [Online]. Available: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view

[10] Y. Wang *et al.*, "A semantics aware approach to automated reverse engineering unknown protocols," in *2012 20th IEEE International Conference on Network Protocols (ICNP)*, 2012.

[11] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014.

[12] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[13] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, 1970.

[14] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "ReFormat: Automatic reverse engineering of encrypted messages," in *European Symposium on Research in Computer Security*, 2009.

[15] "Encoding," *Protocl Buffers Documentation*. Accessed: Nov. 06, 2023. [Online]. Available: https://protobuf.dev/programming-guides/encoding/

[16] "PRE-list." Accessed: Nov. 06, 2023. [Online]. Available: https://github.com/techge/PRE-list

[17] D. Andriesse, C. Rossow, B. Stone-Gross, D. Plohmann, and H. Bos, "Highly resilient peer-to-peer botnets are here: An analysis of gameover zeus," in *2013 8th International Conference on Malicious and Unwanted Software: 'The Americas-' (MALWARE)*, 2013.

[18] S. Haas, S. Karuppayah, S. Manickam, M. Mülhäuser, and M. Fischer, "On the resilience of P2P-based botnet graphs," in *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016.

[19] "Reverse engineering #4258." [Online]. Available: https://github.com/google/flatbuffers/issues/4258

[20] "FlatBuffers: FlatBuffer Internals." Accessed: Aug. 07, 2024. [Online]. Available: https://flatbuffers.dev/flatbuffers_internals.html

[21] K. M. Elleithy, D. Blagovic, W. K. Cheng, and P. Sideleau, *Cybernetics, and Informatics 3.1*, pp. 66–71, 2005.

[22] "Slowloris DDoS attack | Cloudflare." Accessed: Aug. 14, 2024. [Online]. Available: https://www.cloudflare.com/learning/ddos/ddos-attack-tools/slowloris/

[23] "1.1 DNS amplification attack," in *SSAC Advisory SAC008 DNS Distributed Denial of Service (DDoS) Attacks*, 2006. Accessed: Aug. 14, 2024. [Online]. Available: https://itp.cdn.icann.org/en/files/security-and-stability-advisory-committee-ssac-reports/dns-ddos-advisory-31mar06-en.pdf

[24] S. Bratus *et al.*, "Curing the vulnerable parser," *USENIX ;login*, 2017, [Online]. Available: https://www.usenix.org/system/files/login/articles/login_spring17_08_bratus.pdf

[25] "FlatBuffers Binary Format." Accessed: Aug. 07, 2024. [Online]. Available: https://github.com/dvidelabs/flatcc/blob/master/doc/binary-format.md

[26] G. Candea, "Predictable Software–A Shortcut to Dependable Computing?," *arXiv preprint cs/0403013*, 2004, [Online]. Available: https://arxiv.org/pdf/cs/0403013

[27] Y. Acar *et al.*, "You get where you're looking for: The impact of information sources on code security," in *2016 IEEE symposium on security and privacy (SP)*, 2016.

[28] P. L. Gorski, S. Möller, S. Wiefling, and L. L. Iacono, ""I just looked for the solution!" On Integrating Security-Relevant Information in Non-Security API Doc-

umentation to Support Secure Coding Practices," *IEEE Transactions on Software Engineering*, vol. 48, 2021.

[29] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking SSL development in an appified world," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[30] "Decompression bomb vulnerabilities," AERA Network Security. Accessed: Aug. 13, 2024. [Online]. Available: https://web.archive.org/web/20160303233826/http://www.aerasec.de/security/advisories/decompression-bomb-vulnerability.html

[31] "CVE-2003-1564," MITRE Corporation. Accessed: Aug. 13, 2024. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564

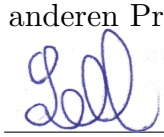
[32] "FlatBuffers: Platform / Language / Feature support ." Accessed: Aug. 15, 2024. [Online]. Available: https://flatbuffers.dev/flatbuffers_support.html

[33] "FlatBuffers: Tutorial." Accessed: Aug. 15, 2024. [Online]. Available: https://flatbuffers.dev/flatbuffers_guide_tutorial.html

[34] "FlatBuffers: Use in C++." Accessed: Aug. 15, 2024. [Online]. Available: https://flatbuffers.dev/flatbuffers_guide_use_cpp.html

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe.

Hamburg, den 07. March 2025: 

Einstellung in die Bibliothek der Informatik

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereich Informatik zu.

Hamburg, den 07. March 2025: