Data Type Bugs Taxonomy: Integer Overflow, Juggling, and Pointer Arithmetics in Spotlight

Irena Bojanova
SSD, ITL
NIST
Gaithersburg, MD, USA
irena.bojanova@nist.gov

Carlos Eduardo Galhardo
Dimel, Disme
INMETRO
Duque de Caxias, RJ, Brazil
cegalhardo@inmetro.gov.br

Sara Moshtari
GCCIS, GCI
RIT
Rochester, NY, USA
sm2481@rit.edu

***Abstract*—In this work, we present an orthogonal classification of data type bugs, allowing precise structured descriptions of related software vulnerabilities. We utilize the Bugs Framework (BF) approach to define four language-independent classes that cover all possible kinds of data type bugs. In BF each class is a taxonomic category of a weakness type defined by sets of operations, cause→consequence relations, and attributes. A BF description of a bug or a weakness is an instance of a taxonomic BF class with one operation, one cause, one consequence, and their attributes. Any vulnerability then can be described as a chain of such instances and their consequence–cause transitions. With our newly developed classes Declaration Bugs, Name Resolution Bugs, Type Conversion Bugs, and Type Computation Bugs, we confirm that BF is a classification system that extends the Common Weakness Enumeration (CWE). The proposed classes allow clear communication about software bugs that relate to misuse of data types, and provide a structured way to precisely describe data type related vulnerabilities.**

***Keywords*—Bug classification, bug taxonomy, software vulnerability, software weakness, type conversion, integer overflow, pointer scaling, juggling.**

I. Introduction

Data types and the operations they define are an abstraction for real-world modeling problems. The purpose of data types is to mitigate bugs, which is achieved by compilers and interpreters enforcing specific data type rules.

Unfortunately, misunderstanding data type peculiarities could create severe software weaknesses and lead to vulnerabilities, such as the recently discovered Type Confusion vulnerability on Chromium [1]. This work aims to classify all data type bugs and to define the kinds of related errors that would allow us to precisely communicate and teach about them, as well as to identify them in code and avoid related security failures.

Secure software development is critical for securing cyberspace and critical infrastructure. However, writing secure code is hard, time consuming, and thus often omitted. Many software bugs/weaknesses get introduced by unaware developers and possibly left undetected by testers and code review tools. As a result, software security vulnerabilities are still a huge part of the attack surface used by threat agents.

A largely used repository of known vulnerabilities is the NIST National Vulnerability Database (NVD) [2]. It links the Common Vulnerabilities and Exposures (CVE) [3] entries to Common Weaknesses Enumeration (CWE) [4] entries. The CISA Known Exploited Vulnerabilities Catalog (KEV) [5] is also based on the CVE. It identifies publicly exploited vulnerabilities with top priority for remediation. However, CVE has proven to be difficult to use in research, as many CWEs and CVEs have imprecise descriptions with unclear causality and lack explainability [6]. Being an enumeration, the CWE also has gaps and overlaps in coverage.

The Bugs Framework (BF) [7] aims to address all these CWE and CVE problems and to further benefit NVD and KEV, as well as to facilitate code review tools and vulnerability research. It is being developed as a structured, complete, orthogonal, and language-independent classification system of software bugs and weaknesses. Structured means a weakness is described via one cause, one operation, one consequence, and one value per attribute from the appropriate lists defining a BF class. This ensures precise causal descriptions. Complete means BF has the expressiveness power to describe any software bug or weakness. This ensures there are no gaps in coverage. Orthogonal means the sets of operations of any two BF classes do not overlap. This ensures there are no overlaps in coverage. BF is also applicable for source code in any programming language. The cause→consequence relation is a key aspect of BF's methodology that sets it apart from any other bugs/weaknesses classification effort. It allows describing and chaining the bug and the weaknesses underlining a vulnerability, as well as identifying a bug by going backwards from a final error or a failure, and what is required to fix that bug.

We utilize the BF approach to define four language-independent, orthogonal classes that cover all possible kinds of data type bugs and weaknesses: Declaration Bugs (DCL), Name Resolution Bugs (NRS), Type Conversion Bugs (TCV), and Type Computation Bugs (TCM). The BF Data Type Bugs taxonomy can be viewed as a structured extension to the conversion, calculation (incl. comparison), wrap-around (incl. integer overflow), pointer scaling, coercion (juggling), and other data type related CWEs, allowing bug reporting tools to produce more detailed, precise, and unambiguous descriptions of identified data type bugs.

The main contributions of this work are: i) we create a model of data type bugs; ii) we create a taxonomy that has the expressiveness power to clearly describe any data type bug or weakness; iii) we confirm our taxonomy covers all data type related CWEs; iv) we showcase the use of our data type bugs taxonomy.

We achieve these contributions respectively via: i) iden-

tifying the operations, where run-time errors may happen despite the compiler or interpreter checks; ii) developing four new structured, orthogonal BF classes: DCL, NRS, TCV, and TCM; iii) generating digraphs of CWEs related to data type weaknesses, as well as to data type consequences and mapping these CWEs to BF DCL, NRS, TCV, and TCM by operation and by consequence; iv) describing real-world vulnerabilities using BF DCL, NRS, TCV, and TCM: CVE-2021-21834 and the Bad Allocation Chain, CWE-468, Example 1 – Incorrect Pointer Scaling, and CVE-2021-23440 and Type Mismatch bypassing Input Validation.

The rest of the paper is organized as follows: In Section II, we recall BF's approach and methodology. In Section III, we discuss the role of type systems. In Section IV, we define the BF Data Type Bugs model. In Section V, we present our new BF DCL, BF NRS, BF TCV, and BF TCM classes. In Section VI, we analyze the correspondence of the conversion, calculation (incl. comparison), and other data type related CWEs to the new BF classes. In Section VII, we use the BF Data Type Bugs taxonomy to provide better, structured descriptions of two real-world vulnerabilities (CVE entries [3]) and an example from a CWE [4] entry. Finally, in Section VIII, we discuss related works and in Section IX we summarize our contributions and propose future works.

II. BF Approach and Methodology

The Bugs Framework (BF) is being developed with the goals of: (1) Classifying software bugs and weaknesses to allow precise descriptions of vulnerabilities that exploit them. (2) Identifying secure coding principles, such as type safety, memory safety, and input/output safety.

In this paper, we use the terms software bug, weakness, and vulnerability as they are defined by Bojanova and Galhardo in [7]. We utilize the latest BF approach and methodology, as described in [8] and present here updates on the main ideas.

BF describes a bug or a weakness as an improper state and its transition. The transition is to another weakness or to a failure. An improper state is defined by the tuple $(\texttt{operation}, \texttt{operand}_1, \cdots, \texttt{operand}_n)$, where at least one element is improper. The initial state is always caused by a bug – a coding error within the operation, which, if fixed, will resolve the vulnerability. An intermediate state is caused by ill-formed data; it has at least one improper operand. The final state, the failure, is caused by a final error – undefined or exploitable system behavior – that usually relates to a CWE. A transition is the result of the operation over the operands.

BF describes a vulnerability as a chain of improper states and their transitions. Each improper state is an instance of a BF class. The transition from the initial state is by improper operation over proper operands. The transitions from intermediate states are by proper operations with at least one improper operand.

Operation or operand improperness defines the causes. A consequence is the result of an operation over its operands. It either becomes the cause for a next weakness or is a final error.

A BF class is a taxonomic category of a weakness type, defined by a set of operations, all valid cause→consequence relations, and a set of attributes. The taxonomy of a particular bug or weakness is based on one BF class. Its description is an instance of a taxonomic BF class with one cause, one operation, one consequence, and their attributes. The operation binds the cause→consequence relation – e.g., a large value of an argument to an addition arithmetic operator leads to integer overflow.

CWEs coverage by newly developed BF classes can be visualized via digraphs, based on CWEs parent-child relationships. Once analyzed, these digraphs can help understand the CWE structure and how the CWE entries translate to BF.

The taxonomies of newly developed BF classes can be demonstrated by providing structured BF descriptions of data type related CVEs.

The methodology for developing a BF class comprises identifying/defining: (1) the phase specific for a kind of bugs; (2) the operations for that phase; (3) the BF Bugs model with operations flow; (4) all causes; (5) all consequences that propagate as a cause to a next weakness; (6) all consequences that are final errors; (7) attributes useful to describe such a bug/weakness; (8) the possible sites in code; (9) CWE digraphs by class and consequence; (10) CVE showcases.

BF is a taxonomy for clearly describing bugs and weaknesses that leverages secure coding education and training. Secure coding is a software development technique introducing awareness of well-known weaknesses to avoid vulnerabilities. This paper introduces the *BF Data Type Bugs* classes, which cover bugs related to type safety. Previous works focused on memory safety [9] and input/output safety [8], both critical steps of secure coding development [10].

III. Type Systems

Type safety of programs is ensured by programming languages' Type Systems [11]. A *Type System* checks explicit typing rules and applies built-in implicit typing rules to eliminate run-time errors. Explicit typing rules are on name resolution or any predetermined behavior, (e.g. allowing multiplication only for numeric data types). Implicit typing rules are such as on type inference, argument coercion, or polymorphic call resolution.

Each data type bug or weakness involves one data type related operation. Each of these operations is over an *entity*: object, function, data type, or namespace. An entity is referred in source code via its declared *name*. Names may be organized in namespaces (or modules, or packages) to avoid name collisions. This eliminates ambiguity, allowing the same name to be declared in different scopes. A *scope* is a block of code (e.g. { }) where a name is valid, i.e., associated with exactly one entity. Scopes provide the Type System with a context for names lookup, i.e., what namespaces and in which order should be referred to resolve the name.

A *data type* defines a set or a range of values (e.g. char is within `[-128, 127]`) and the operations allowed over them (e.g. `+`, `*`, `mod`). A data type can be *primitive* (e.g. int, float, double, string, boolean) or *structured* (e.g. array, record, union, class, interface). Primitive data types mimic the hardware units; they are only language defined. Structured data types are built from other data types; they have primitive or structured members. The Type System checks if data types have been declared and used properly.

An *object* is defined by Bojanova and Galhardo in [9] as "a piece of memory with well-defined size", which "address should be held by at least one pointer or determined as an

offset on the stack". This piece of memory is used to store the value of a primitive data or a data structure. An object is *declared* via a name and a data type. It may be a member of a data structure (e.g. an *attribute* of a class). An object is referred in source code via its name. If a data type is not specified, the Type System may infer it from the object's data value. An object is *defined* by the data it stores.

A *function* is an organized block of code that when called takes in data, processes it, and produces a result(s). Functions also get stored in memory. A function is *declared* via a return data type, a name, and a list of parameters. It may be a member of a data structure (e.g. a *method* of a class). If a return data type is not specified, the Type System may infer it from the returned data value. A function may be *anonymous*, t.e. it may have no name – e.g. a lambda expression. A function is *defined* by its block of code – the function implementation. A function is referred in source code via its name and called via its signature (name and arguments – number, order, and data types matter). A function call may be an argument to another function – e.g. to an operator in an arithmetic expression[1].

The declaration of an entity may be prefixed by a *modifier* that enforces required restrictions. For example, access modifiers restrict whether and how a data type, an object, or a function can be used outside its declaration scope.

Object Oriented Programming (OOP) languages extend to *Polymorphic Type Systems*, allowing use (a) of same code for different data structures or (b) of different code via same function name. They introduce the notions of type parameters, subtyping (via inheritance), dynamic typing and overriding [12], and the concept of overloading.

Type Parameters allow encoding of a generic data type, parameterized by the data types of its member objects and its member functions. A generic member function essentially works the same way on different types of data structures. An example of parametric polymorphism (*generics*) would be a generic sort function that works the same way over a list of integers as over a list of strings.

Subtyping is the ability to define new data types based on previously defined data types. Via such inheritance, two structurally different data types may share members. Dynamic Typing is the ability for objects to change their data types at run time – e.g. the data type of a base class object may be any of its class subclasses. An example of subtype polymorphism (*overriding*) would be a dynamically typed element of an array of `Shape` objects invoking the proper `draw()` method depending on the element's actual subtype.

Both overriding and overloading are about the ability to use the same function name for different implementations: overriding is across subtypes in the same hierarchy; overloading is in the same declaration scope. An example of compile-time polymorphism (*overloading*) would be using the proper integer or double addition operator (`int +(int x, int y);` or `double +(double x, double y);`) depending on the call arguments data types.

[1]Operators are implemented as functions (e.g. `int +(int x; int y);`), although, their names are special characters instead of alpha-numeric and they are referred via mathematical notations (e.g. `a+b` instead of `+(a,b)`). Note that the assignment operator (=) is also a function – the value being assigned is passed in as an argument.

The Type System *resolves* the declaration scopes of the referred entity names; then *binds* the appropriate declared (or inferred) data types and the defined objects data and called functions implementations.

An entity name referred out of its scope gets resolved according to specific scoping rules. Names in nested scopes get resolved through declarations from inner to outer scopes. This may involve resolving one or more namespaces, then – a class name to a context, and then – the member name within that context. A fully qualified name lists all these scoping names – e.g., via a dot notation (`java.lang.Math.sin(x);`). It unambiguously specifies which object, function, or data type is being referred even when out of its declaration scope. The listed names play the role of name qualifiers. The Type System applies the proper *qualifiers* when a `using` (`import`/`include`) statement is specified.

After an entity name is resolved, it gets bound appropriately: for an object – its data type and its data are bound; for a function – its return type and its parameters' types are bound. A function is called with resolved and bound arguments (actual parameters); then its implementation is bound.

An object or a function must be completely resolved before it is used. In the case of a polymorphic object, the Type System resolves it also among the corresponding hierarchical structure. In the case of a polymorphic function, the Type System resolves it based on its type arguments (generics), its invoking object data type (overriding), or its signature (overloading). The type argument, the dynamic object data type, or the arguments or function declarations, may correspondingly be a problem. In most programming languages, generics and function/operator overloading are resolved at compile-time (early binding and ad-hoc binding, correspondingly) and virtual methods overriding is resolved at run-time (late binding).

A resolved and bound object may be *converted* to another data type and used in *computations*. Conversions could be explicit – *casting*, or implicit – *coercion*[2]. Computations could be *calculation* of arithmetic expressions or *evaluation* of boolean expressions. Arithmetic expressions are used to calculate for example how much memory to allocate, the size of a structured object, or the position of an index. Pointer arithmetics and pointer scaling may be used to calculate position of pointer or memory address. Boolean expressions are used to evaluate conditions.

IV. Data Type Bugs Model

Data Type bugs could be introduced at any of the declaration, name resolution, data type conversion, or data type related computation phases. Each data type related bug or weakness involves one data type operation: *Declare*, *Define*, *Refer*, *Call*, *Cast*, *Coerce*, *Calculate*, or *Evaluate*.

The BF Data Type Bugs model (Fig. 1) helped us identify the phases and the operations where such bugs could occur. The phases correspond to the BF Data Type Bugs classes: *Declaration Bugs (DCL)*, *Name Resolution (NRS)*, *Type Conversion Bugs (TCV)*, and *Type Computation Bugs (TCM)*. All data type operations are grouped by phase.

The operations under DCL (Fig. 1) are on declaring entities names and on defining objects data and functions

[2]Type Coercion is known also as *Type Juggling.*

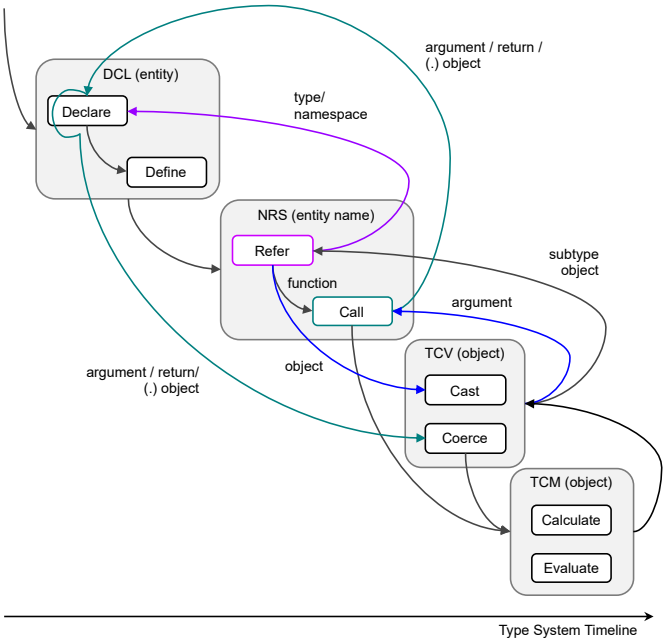


Fig. 1: The BF Data Type Bugs model. Comprises phases corresponding to the BF classes DCL, NRS, TCV, and TCM. Shows the data type operations flow.

implementations: *Declare* and *Define*. See definitions of DCL operations in Table Ia.

The operations under NRS (Fig. 1) are on resolving referred entities names and on binding their data types and the objects data, and on resolving called functions and binding their implementations: *Refer* and *Call*. See definitions of NRS operations in Table Ib. The NRS operations are tied to the name resolution and binding that the Type System performs. Fig. 2 shows the corresponding sub-model. Note that the object data value is bound via the *Initialize* and *Write* operations of the *BF Memory Use Bugs (MUS)* class [9].

The operations under TCV (Fig. 1) are on explicit conversion and on implicit conversion into a different data type of a passed data value in/out of a function: *Cast* and *Coerce*. See definitions of TCV operations in Table Ic.

The operations under TCM (Fig. 1) are on calculating an arithmetic operation (part of an algebraic expression) and on evaluating a boolean operation (part of a condition): *Calculate* and *Evaluate*. See definitions of TCM operations in Table Id.

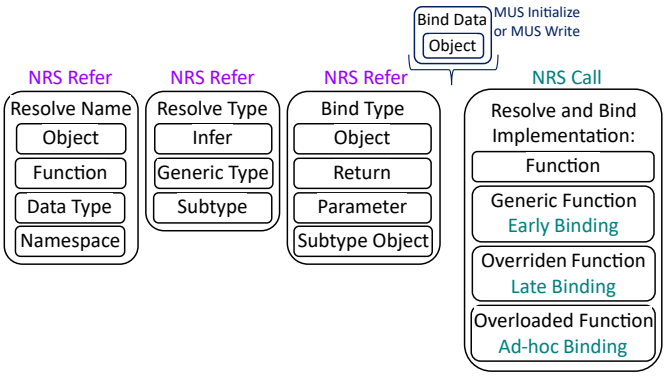


Fig. 2: Name Resolution and Biding.

The possible flow between operations from different phases is depicted in Fig. 1 with arrows. A declared and defined entity is referred in source code via its name. Names, referred to in remote scopes, get resolved via resolved namespaces; resolved data types get bound to objects, functions, or generic data types according to their declarations (see the purple arrow flow). A resolved and bound object may be converted to another data type and used in computations as an argument or as a return of a called function, or to call a member function. A passed in argument is expected to be of the declared parameter data type and the passed out result is expected to be of the return data type. Otherwise, casting (explicit conversion) is expected before or at the end of the call (see the blue arrows flow), or the value will get coerced (implicitly converted) to the parameter data type or the return data type, correspondingly (see the green arrows flow). Note that the green arrows flow is only about coerced passed in/out objects – it starts only from NRS Call, it never starts from DCL Declare. The presented operations flow helps in identifying possible chains of bugs and weaknesses.

V. Data Type Bugs Classes

We define the BF Data Type Bugs classes as follows:

- Declaration Bugs (DCL) – *An object, a function, a data type, or a namespace is declared or defined improperly.*
- Name Resolution Bugs (NRS) – *The name of an object, a function, or a data type is resolved improperly or bound to an improper data type or implementation.*
- Type Conversion Bugs (TCV) – *A data value is cast or coerced into another data type improperly.*
- Type Computation Bugs (TCM) – *An arithmetic expression (over numbers, strings, or pointers) is calculated improperly, or a boolean condition is evaluated improperly.*

Each of these classes represents a phase, aligned with the Data Type Bugs model (Fig. 1), and is comprised of sets of operations, cause→consequence relations, and attributes, allowing precise causal descriptions of bugs/weaknesses in the declaration or definition of an object, a function, or a data type, the resolution of their names, the binding of their types or implementations, or of any type related conversions and computations.

Fig. 3, Fig. 4, Fig. 5, and Fig. 6 show the specific sets for declaration, name resolution, computation, and conversion bugs, respectively. Only the values listed on the corresponding figure should be used to describe that kind of bugs or weaknesses.

A. Operations

All BF classes are being designed to be orthogonal; their sets of operations do not overlap. The operations in which data type bugs could happen (defined in Table I) correspond to the operations in the BF Data Type Bugs model (Fig. 1).

The DCL operations are Declare and Define. They reflect the improper declaration or definition of an object, a function, a data type, or a namespace. The NRS operations are Refer and Call. They reflect the improper name resolution, and data type, data, or implementation binding. The TCV operations

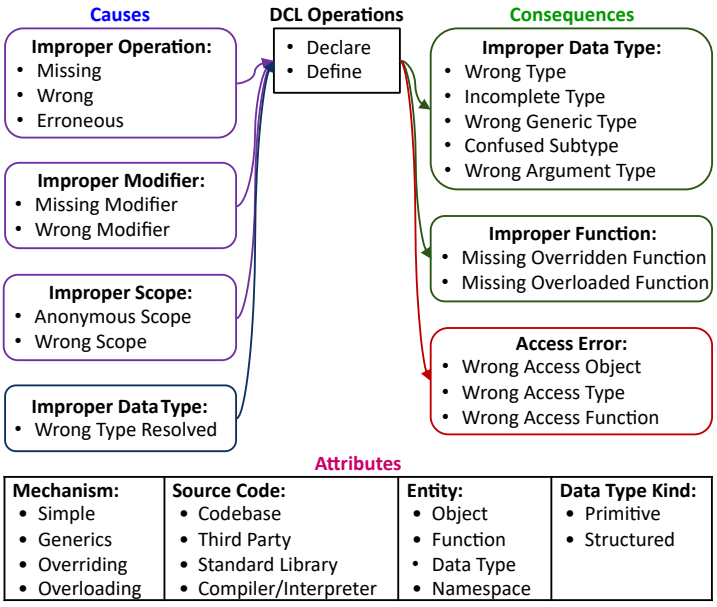


Fig. 3: The Declaration Bugs (DCL) class.

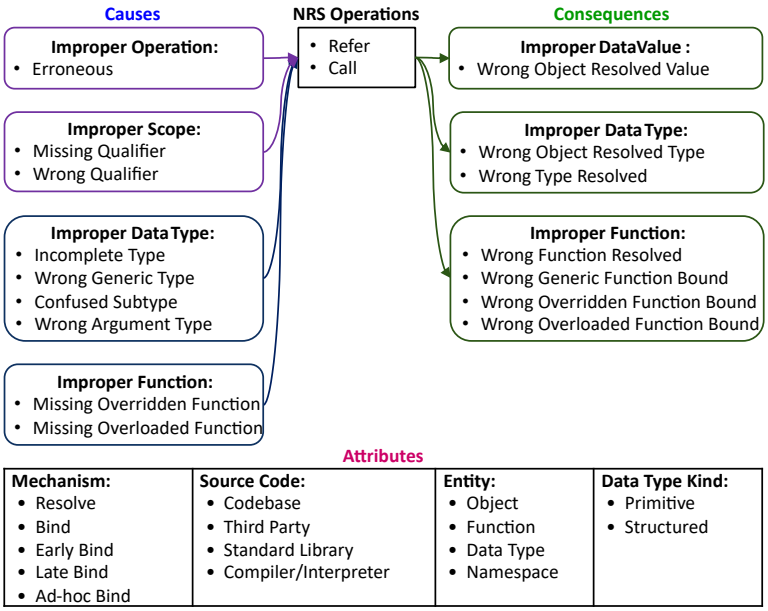


Fig. 4: The Name Resolution Bugs (NRS) class.

are Cast[3] and Coerce. They reflect the improper use of data types in object data value conversion to another data type. The TCM operations are Calculate[4] and Evaluate[5]. They reflect the improper use of data types in arithmetic calculations and condition evaluations.

[3]Cast operators: `(int) x`; in C, `int(x)` in Python, `x as int`; in Rust.

[4]Calculate involves: Arithmetics (`+`, `++`, `unary+`, `-`, `--`, `unary-`, `*`, `/`, `=+`, `=-`, `=*`, `=/`, `rem`, `div/mod`), Bitwise Shift: (`<<`, `>>`, `&`, `|`, `^`, `~`), Pointer Arithmetics and Pointer Scaling (`=+`, `+`, `++`; `=-`, `-`, `--`; and use of `SizeOf`), Concatenation (`+`), and Math/String Functions (`max()`, `sin()`, `log()`, `strlen()`, `memchr()`, etc.).

[5]Evaluate involves: Comparison (`<`, `>`, `<=`, `>=`, `!=`, `!==`, `==`, `===`, `strcmp()`, `memcmp()`) and Boolean Logic (`&&`, `||`, `!`).

B. *Causes*

A cause is either an improper operation or an improper operand. If a BF class instance is the first in a chain, describing a vulnerability, it is always caused by an improper operation or an improper operation rule. The values for improper data type related operations are Missing, Wrong, and Erroneous. The possible operation rules are Improper Modifier and Improper Scope. The values for these are correspondingly: Missing Modifier and Wrong Modifier; and Anonymous Scope, Wrong Scope, Missing Qualifier, and Wrong Qualifier. See definitions and examples in Table II.

The possible operands of a Data Type operation are: Data Value, Data Type, and Function. See definitions in

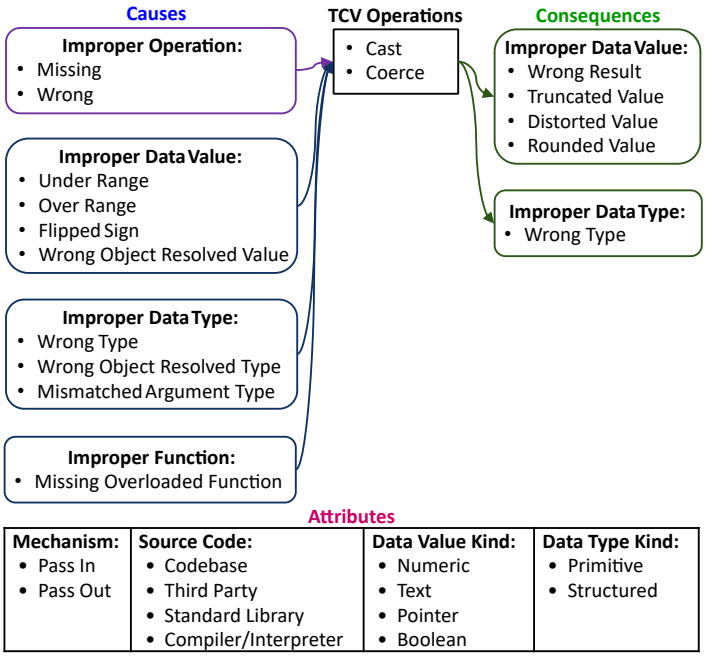


Fig. 5: The Type Conversion Bugs (TCV) class.

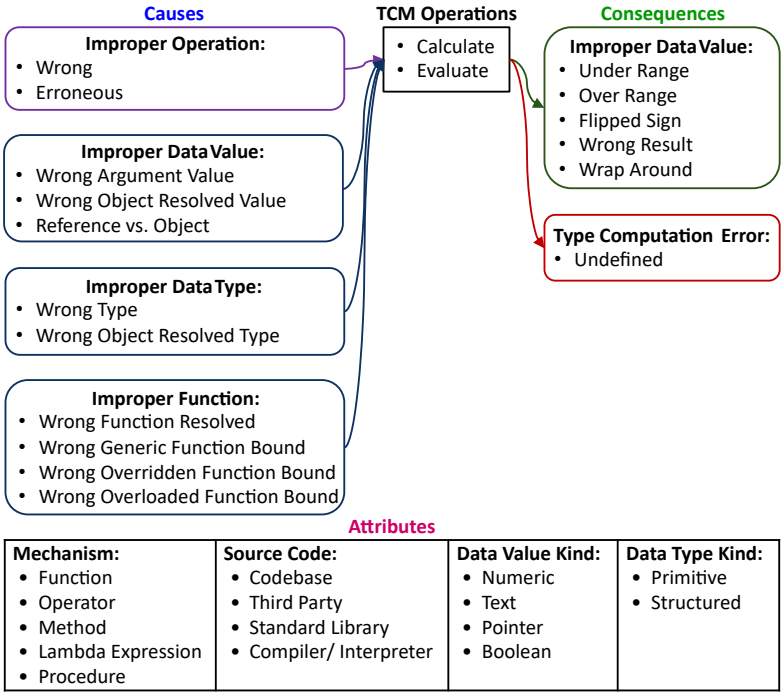


Fig. 6: The Type Computation Bugs (TCM) class.

Table III. All values for an improper operand of a data type operation are defined in Table IV.

All possible causes for data type bugs are defined in Table II and Table IV. However, refer Fig. 3, Fig. 4, Fig. 5, and Fig. 6 for causes applicable to each class.

C. Consequences

A consequence is either a final error or a wrong result from the operation that propagates as an improper operand for a next weakness.

The possible improper operands as consequences from data type bugs classes are Improper Data Value, Improper Data Type, and Improper Function. All their possible values are defined in Table IV. As a consequence, they would become causes for operations of next weaknesses. These consequence–cause transitions explain why some appear both as causes and as consequences.

TABLE I: Operations

(a) DCL (Declaration)

Operation Value	Definition
Declare	Specify name and data type of an object; name, return data type, and parameters of a function; or name and type parameters of a data type. May specify modifiers for required behaviour restrictions.
Define	Specify data of an object; implementation of a function; or member objects and functions of a type.

(b) NRS (Name Resolution)

Operation Value	Definition
Refer	Use a name in local or remote scopes of source code. The Type System resolves the name and binds a data type to it.
Call	Invoke a function implementation. The Type System binds the implementation to the resolved function name. A polymorphic implementation is first resolved and then bound.

(c) TCV (Conversion)

Operation Value	Definition
Cast	Explicitly convert the value of an object to another data type.
Coerce	Implicitly (forced by the Type System) convert the value of a passed in/out argument or the return into the corresponding parameter or return data type. (Type Coercion is known also as *Type Juggling*.)

(d) TCM (Computation)

Operation Value	Definition
Calculate	Find the result of a numeric, pointer, or string operation.
Evaluate	Find the result of a boolean condition (incl. comparison).

The possible data type final errors are: Access Error and Type Computation Error. Their possible values are defined in Table V.

All possible consequences for data type bugs are defined in Table IV and Table V. However, refer Fig. 3, Fig. 4, Fig. 5 and Fig. 6 for consequences applicable to each class.

D. Attributes

An attribute provides additional useful information about the operation or its operands. All possible attributes for data type bugs are defined in Table VI. The operation attribute Source Code explains where the bug is in the program. The operand attribute Data Type Kind explains the data type structure. See definitions of these attributes values in Table VIa.

Each of DCL, NRS, TCV, and TCM also have the operation attribute Mechanism but with different possible values, specific to the particular data type operations. See definitions of this attribute values in Table VIb, Table VIc, Table VId, and Table VIe.

DCL and NRS also have the operand attribute Entity, which explains what is being declared or resolved. See the definition of this attribute value in Table VIf.

TCV and TCM also have the operand attribute Data Value Kind, which explains the actual data value. See its definition in Table VIg.

E. Sites

A site for data type bugs is any part of the code for which the Type System checks explicit typing rules or applies implicit typing rules.

TABLE II: Improper Operation

Value	Definition	Examples
Missing	The operation is absent.	Missing: • constructor • `+(int,double)` overload • function override in subtype.
Wrong	An inappropriate data type is specified; or an inappropriate function/operator is used.	• An object is declared `int`, while it should be `float`. • A class implements a cloneable or a serializable interface. • Comparison via `=` vs. `==`.
Erroneous	The Type System or a compute function implementation has a bug.	• Incorrect data type inference. • Wrong order or number of arguments to a function call. • Incorrect deep objects comparison implementation.
Missing Modifier	A required behavioral restriction is absent.	• Access: `public`, `private`, `protected`, `internal` • Type: `long`, `long long`, `short`,`unsigned`,`signed`
Wrong Modifier	A wrong behavioral restriction is specified.	• Use of `private` instead of `protected` modifier.
Anonymous Scope	The declaration is in an unnamed scope.	• An inner class.
Wrong Scope	The declaration should be in another scope.	• Object declared as local, while it should be global.
Missing Qualifier	A namespace include is absent; or a scope is not specified in a fully qualified name.	• A user defined method with same name is invoked instead of a needed library one.
Wrong Qualifier	A wrong namespace is included, or a wrong scope is specified in a fully qualified name.	• Use of `math.log` instead of `numpy.log` when the second one is needed.

TABLE III: Operands

Concept	Definition
Data Value	A numeric, text, pointer/address, or boolean value stored in an object's memory.
Data Type	A set of allowed values and the operations allowed over them.
Function	An organized block of code that when called takes in data, processes it, and produces a result(s).

DCL sites are the entities declarations and definitions. NRS sites are the entities references and the function calls. TCV sites are the cast operators and the in/out argument passing and return statements. TCM sites are the arithmetic, bitwise shift, concatenation, pointer arithmetics, pointer scaling, relational, and boolean operators.

VI. BF Data Type Bugs Taxonomy as CWE Extension

In this section, we analyze the correspondence of the data type related CWEs [4], such as *Numeric Errors* [13], *Type Errors* [14], and *String Errors* [15], to the four newly developed BF Data Type Bugs classes. We show that DCL, NRS, TCV, and TCM cover all data type related CWEs, and potentially beyond, while (as demonstrated in Section VII) providing a better structured way for describing these kinds of bugs/weaknesses.

We identified data type related CWEs in three steps: *1) CWE Filtering*: Since different types of bugs/weaknesses are

TABLE IV: Improper Operand

(a) Improper Data Value

Concept	Definition
Under Range	Data value is smaller than type's lower range.
Over Range	Data value is larger than type's upper range.
Flipped Sign	Sign bit is overwritten from type related calculation.
Wrong Argument Value	Inaccurate input data value; i.e., non-verified for harmed semantics.
Wrong Object Resolved Value	Object is resolved from wrong scope.
Reference vs. Object	Object's address instead of object's data value.
Wrong Result	Incorrect value from type conversion or computation.
Wrap Around	A moved around-the-clock value over its data type upper or lower range, as it exceeds that range. (*Integer Over-/Under-flow* is a wrapped-around the upper/lower range integer value; may become very small/large and change to the opposite sign.)
Truncated Value	Rightmost bits of value that won't fit type size are cut off.
Distorted Value	Incorrect value (although fits type size) due to sign flip or signed/unsigned and vice versa conversions.
Rounded Value	Real number value precision loss.

(b) Improper Data Type

Wrong Type	Data type range or structure is not correct.
Wrong Type Resolved	Data type is resolved from wrong scope.
Wrong Object Resolved Type	Object is resolved from wrong scope,so it's data type might be wrong.
Wrong Sign Type	Unsigned instead of signed data type is specified or vise versa.
Wrong Precision Type	Higher precision data type is needed (e.g. `double` instead of `float`).
Incomplete Type	Specific constructor, method, or overloaded function is missing.
Mismatched Argument Type	Argument's data type is different from function's parameter data type.
Wrong Generic Type	Generic object instantiated via wrong type argument.
Confused Subtype	Object invoking an overriden function is of wrong subtype data type.
Wrong Argument Type	Argument to an overloaded function is of wrong data type.

(c) Improper Function

Missing Overridden Function	Function implementation in a particular subclass is absent.
Missing Overloaded Function	Implementation for particular function parameters' data types is absent.
Wrong Function Resolved	Function is resolved from wrong scope.
Wrong Generic Function Bound	Implementation for a wrong data type is bound due to wrong generic type arguments.
Wrong Overridden Function Bound	Implementation from wrong subtype is bound due to a wrong invoking subtype object.
Wrong Overloaded Function Bound	Wrong overloaded implementation is bound due to wrong function arguments.

TABLE V: Data Type Errors

(a) Access Errors – as DCL Consequences

Value	Definition
Wrong Access Object	Unauthorized access to an object exposes sensitive data or allows access to member functions.
Wrong Access Type	Unauthorized access to a data type allows access to member objects and functions.
Wrong Access Function	Unauthorized access to a function.

(b) Type Compute Errors – as TCM Consequences

Value	Definition
Undefined	The Type System cannon represent the computation result (e.g. division by `0`).

TABLE VI: Attributes

(a) DCL, NRS, TCM, and TCV Attributes

Name	Value	Definition
Source Code	Codebase	The operation is in the programmer's code – in the application itself.
	Third Party	The operation is in a third-party library.
	Standard Library	The operation is in the standard library for a particular programming language.
	Compiler/ Interpreter	The operation is in the language processor that allows execution or creates executables (compiler, assembler, interpreter).
Data Type Kind	Primitive	Mimics the hardware units and is not built from other data types – e.g. int (long, short, signed), float, double, string, boolean.
	Structured	Builds of other data types; have members of primitive and/or structured data types – e.g. array, record, struct, union, class, interface.

(b) DCL Attribute

Name	Value	Definition
Mechanism	Simple	A non-polymorphic entity.
	Generics	An entity parameterized by type.
	Overriding	Functions with the same name as one in the base type, but implemented in different subtypes.
	Overloading	Functions with the same name in the same declaration scope, but implemented with different signature.

(c) NRS Attribute

Name	Value	Definition
Mechanism	Resolve	Look up entity name and if needed determine data type (infer from value, through hierarchy, via generic type attribute).
	Bind	Connect object data type, function return type, parameter data type, or simple function implementation.
	Early Bind	Resolve subtype and set generic function implementation.
	Late Bind	Resolve overriden function via subtype object and set implementation.
	Ad-hoc Bind	Resolve overloaded function via signature and set implementation.

(d) TCV Attribute

Name	Value	Definition
Mechanism	Pass In	Supply "in" arguments' data values to a function/ operator.
	Pass Out	Supply "out" or "in/out" arguments' data values or a return value to a function/ operator.

(e) TCM Attribute

Name	Value	Definition
Mechanism	Function	An organized block of code that when called takes in data, processes it, and returns a result.
	Operator	A function with a symbolic name that implements a mathematical, relational or logical operation.
	Method	A member function of an OOP class.
	Lambda Expression	An anonymous function, implemented within another function.
	Procedure	A function with a `void` return type.

(f) DCL and NRS Attributes

Name	Value	Definition
Entity	Object	A memory region used to store data.
	Function	An organized block of code that when called takes in data, processes it, and returns a result.
	Data Type	A set or a range of values and the operations allowed over them.
	Namespace	An organization of entities' names, utilized to avoid names collision.

(g) TCV and TCM Attribute

Name	Value	Definition
Data Value Kind	Numeric	A number stored in an object's memory.
	Text	A string stored in an object's memory.
	Pointer	A holder of the memory address of an object.
	Boolean	A truth value (`true` or `false`; 1 or 0), stored in an object's memory.

described in CWE [4], we filtered a set of CWEs which descriptions contain keywords such as "type", "string", "class", "cast", and "compare". *2) Automated Extraction*: Starting from the filtered CWEs and following their parent-child relationships, we extracted all the clusters of potentially data type related CWEs. *3) Manual Review*: All the authors, who are a team of professional security researchers, performed iterative rounds of manual CWE reviews, identifying the type related CWEs among the extracted CWE clusters. Finally, we could collect 84 CWEs, 78 of which are data type related. The additional six CWEs: 573, 664, 668, 710, 758, and 1076 (shown with gray outline) were included only for parent-child completeness. We peer-reviewed their detailed descriptions, examples, and listed CVEs, as well as the corresponding literature; and performed weekly discussions brainstorming to confirm each of these CWEs is covered by the operations, causes, and/or consequences defining the BF *Data Type Bugs* classes.

We mapped each of the identified CWEs to a BF *Data Type Bugs* class based on the operations that are defined in DCL, NRS, TCV, and TCM and identified an operation for the CWE. Then, we generated digraphs of all data type related CWEs to show their correspondence to the BF *Data Type Bugs* classes by operation (Fig. 7) and by consequence (Fig. 8). In the digraphs, each node is a CWE weakness, shown by its CWE ID, and the edges show the parent/child relationship. The outline style of a CWE node indicates the abstraction level: pillar, class, base, or variant.

In Fig. 7, the outline color of a CWE node indicates the BF class(es) and operation(s) associated with that CWE: DCL Declare, DCL Define, NRS Refer, NRS Call, TCV Cast, TCV Coerce, TCM Calculate, and TCM Evaluate.

Most of the CWEs, visualized on the digraph, are covered by the DCL class. They relate to improper declaration and definition of structured data and are under the pillars CWE-664 (*Improper Control of a Resource Through its Lifetime*) and CWE-668 (*Exposure of Resource*), and CWE-710 (*Improper Adherence to Coding Standards*), correspondingly.

Two other large clusters of data type related CWEs are covered by TCM. They relate to improper calculation and evaluation and are descendants of the pillars CWE-682 (*Incorrect Calculation*) and CWE-697 (*Incorrect Comparison*), correspondingly. Two of the CWE-628 (*Function Call with Incorrectly Specified Arguments*) descendants are also covered by TCM Calculate and Evaluate. In addition, CWE-351 (*Insufficient Type Distinction*) is covered by TCM Evaluate.

The CWEs covered by TCV are under CWE-704 (*Incorrect Type Conversion or Cast*). They mostly relate to improper coercion – CWE-843 (*Access of Resource Using Incompatible Type ('Type Confusion')*) and the children of CWE-681 (*Incorrect Conversion between Numeric Types*). Only one CWE was identified as improper casting – CWE-588 (*Attempt to Access Child of a Non-structure Pointer*).

Only a few CWEs are partly related to the NRS class, which is surprising as bugs related to polymorphic calls are not rare [16]. Our explanation is that CWE considers improper name referring/resolving and improper function calling/binding to be part of a computation weakness. For instance, CWE-468 is under CWE-682 (*Incorrect Calculation*), but it lists an example that starts with a TCV bug leading to an NRS weakness, and is actually a five weaknesses chain (see the BF description in section VII-B).

Fig. 9 shows the percentages of data type related CWEs by BF class operation. It shows that most of these CWEs are about weaknesses that occur at declaration and definition (50% combined) of objects, types, and functions. Next are weaknesses related to performing data type related calculations or evaluations (33.3% combined). The least represented are the name resolution (at refer and call) and type conversion (at cast and coerce) weaknesses.

While the CWEs only enumerate weaknesses, the *Data Type Bugs* classes ensure precise descriptions, as a weakness is described via one cause, one operation, and one consequence. The CWEs exhaustive list is prone to gaps in coverage and some weakness types may be missing. For example, in the type related categorization the CWEs are mainly focused on primitive data type errors, such as *Numeric Errors* [13] and *String Errors* [15], while our developed BF classes consider both primitive and structured data types. Besides that, the data type related CWEs focus mostly on types of errors that happen during arithmetic calculations and comparison evaluations [13], while the BF Data Type Bugs classes define type related bugs based on different stages of the data types development: from declaration and definition to resolution and usage.

The BF Data Type Bugs classes present a taxonomy with structured cause→consequence relations that is complete and orthogonal. It could be viewed as a structured extension over the CWEs related to Wrong Result, Rounded Value, Truncated Value, Wrap Around (incl. Integer Overflow), Under Range, Over Range Flipped Sign, Wrong Access, and Undefined Behaviour (see Fig. 8). It is a taxonomy that explains the causal relationships between weaknesses and would be easier to use than the nested hierarchical CWEs.

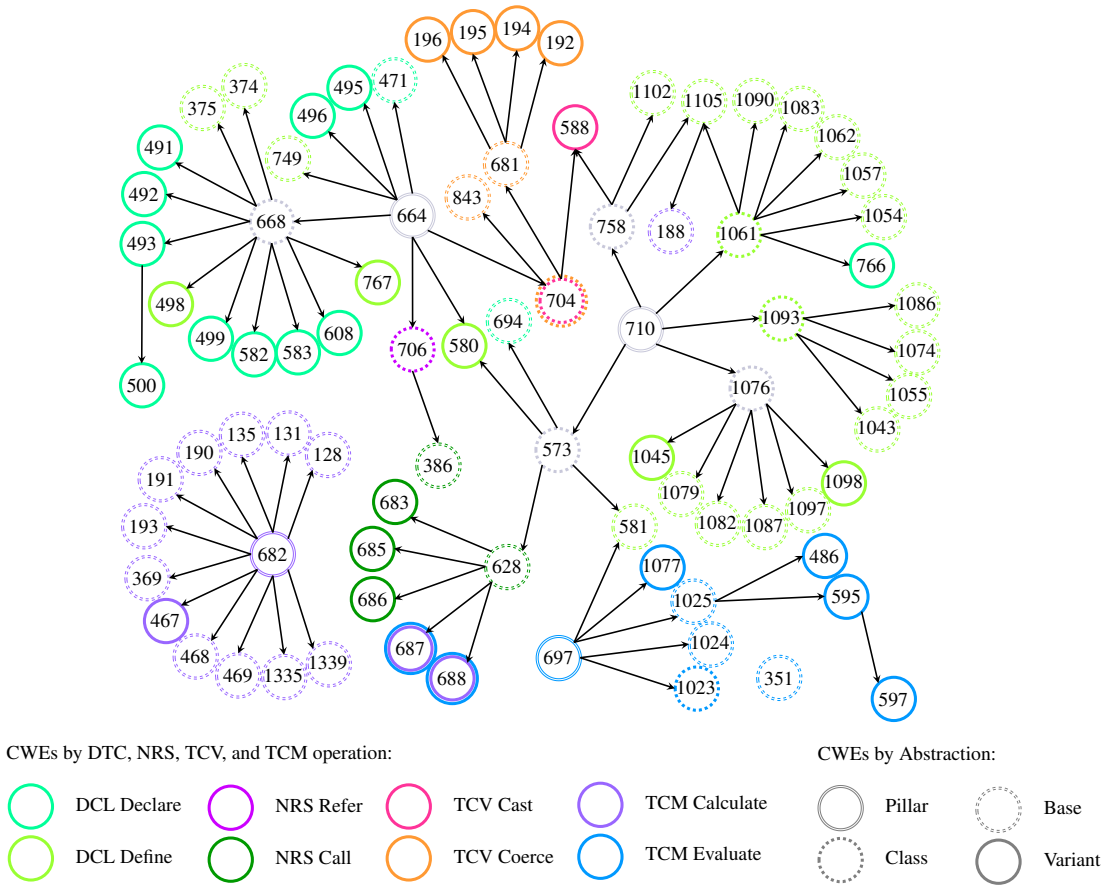


Fig. 7: A digraph of the data type related CWEs, *mapped by DCL, NRS, TCV, and TCM operations*. Each node represents a CWE ID. Each arrow represents a parent-child relationship. → Click on an ID to open the CWE entry.

Many bug reporting tools use the CWE [4] to describe detected bugs/weaknesses [17]. As a structured extension over the data type related CWEs, the BF Data Type Bugs taxonomy can be used to report identified data type related bugs/weaknesses (including those leading to integers overflow, juggling, and pointer arithmetic errors). Fig. 7 shows how data type related CWEs translate to BF DCL, NRS, TCV, TCM by operation; Fig. 8 shows how they translate by consequences.

VII. Showcases

In this section, we use the new BF *Data Type Bugs* classes for precise descriptions of software vulnerabilities. We also provide the fixes of each bug.

A. *CVE-2021-21834 and the Bad Allocation Chain*

Our first example is an instance of a BF Chain that defines a pattern ("BadAlloc") observed by CISA and reported in its Industrial Control Systems Advisorys, ICSA-21-119-04 [18]. The advisory lists 25 similar vulnerabilities found in multiple real-time operating systems.

Using the BF classification, we describe BadAlloc as a DVR → TCM → MAL → MAD → MUS chain. That is, due to improper verification (DVR), an attacker can craft an input that creates an integer overflow (TCM), leading to allocation (or reallocation) of an undersized buffer (MAL). Therefore, a pointer can be moved overbounds (MAD), leading to a buffer overflow error (MUS). This vulnerability could lead to a failure, such as denial of service or, even worse, (remote) code execution.

To clearly illustrate how this pattern happens, we describe in detail CVE-2021-21834 that occurs in an open source project. It was identified by the Cisco Talos team [19]. The vulnerable code can be found in [20].

1) Brief Description: The GPAC Project on Advanced Content is a C language implementation of the MPEG-4 audio/video compression standard. In version 1.0.1, the library is vulnerable to decoding a specially crafted MPEG-4 input file.

2) Analysis: The library code reads the "number of entries" value from a file into the 32-bit integer object (`ptr→nb_entries`) and checks if it is not larger than the 64-bits input size (`ptr→size/8`). Then, the size of memory that should be allocated is calculated by multiplying the "number of entries" by the size of a u64 object (`sizeof(u64)`), which can result in an integer overflow on a 32-bit platform. When such an overflowed integer is used, the allocation routine will create an undersized buffer, which will be populated based on its larger actual size, leading to a buffer overflow. Fig.10 presents the BF taxonomy for this vulnerability.

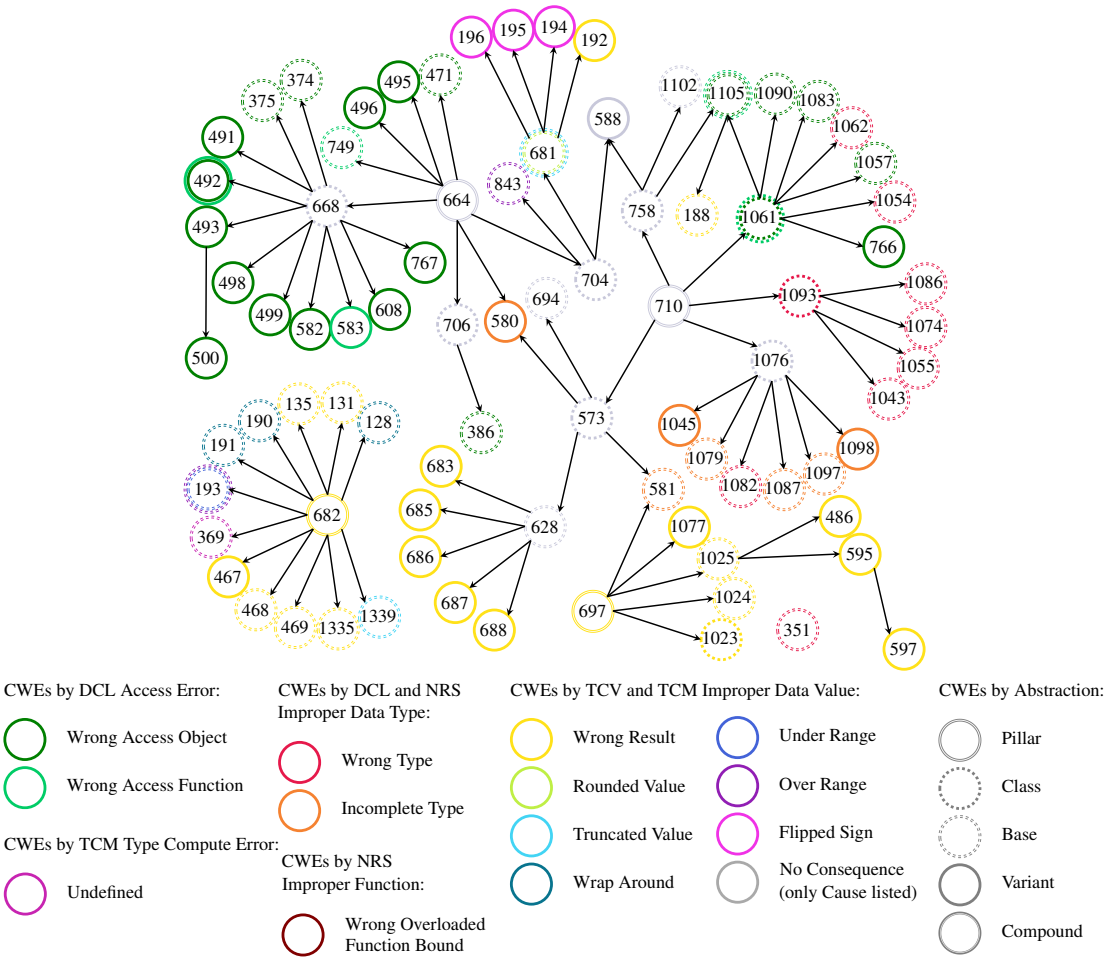


Fig. 8: A digraph of the data type related CWEs, *mapped by BF DCL, NRS, TCV, and TCM consequences.* Each node represents a CWE ID. Each arrow represents a parent-child relationship. → Click on an ID to open the CWE entry.

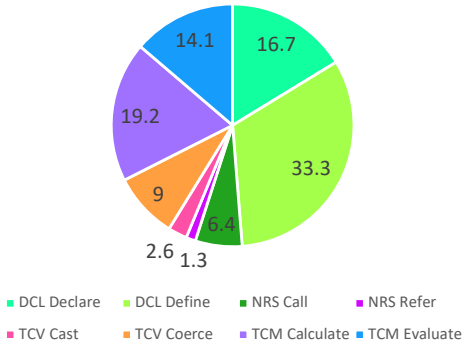


Fig. 9: A diagram of the percentages of data type related CWEs by the operations of the *BF Data Type Bugs* classes.

3) The Fix: To fix the bug, the GPAC team checked the maximum integer size (`((u64)ptr→nb_entries > (u64)SIZE_MAX/sizeof(u64)`), fixing the DVR bug and resolving the entire vulnerability [21].

B. CWE-468, Example 1 – Incorrect Pointer Scaling

Our second example is a pointer scaling bug, illustrating how a simple piece of code could have a bug and several weaknesses behind it. This is a common C/C++ pointer bug; it happens when a programmer miscalculates a pointer increment (or decrement) by a fraction of its type size (e.g., attempting to move an `int` pointer one byte to the right).

CWE-468 provides an excellent two lines of code example: `int *p = x; char * second_char = (char *)(p + 1); .`

1) Analysis: The chain starts with an improper casting of the pointer `p` to `char *` that leads to invocation of a wrong addition operator[6] `int* + 1` instead of `char* + 1`. Therefore, the pointer moves 4 bytes instead of 1 byte (3 bytes off), reading the wrong value, outside the object `x` (buffer overflow). Fig.11 presents the BF taxonomy for this weakness.

2) The Fix: To fix the bug, the programmer should first cast and then add: `int *p = x; char * second_char = (char *)p + 1; .`

[6]The C standard does not explicitly talk about "overloading", but the required properties and equalities are met for overloaded operators.

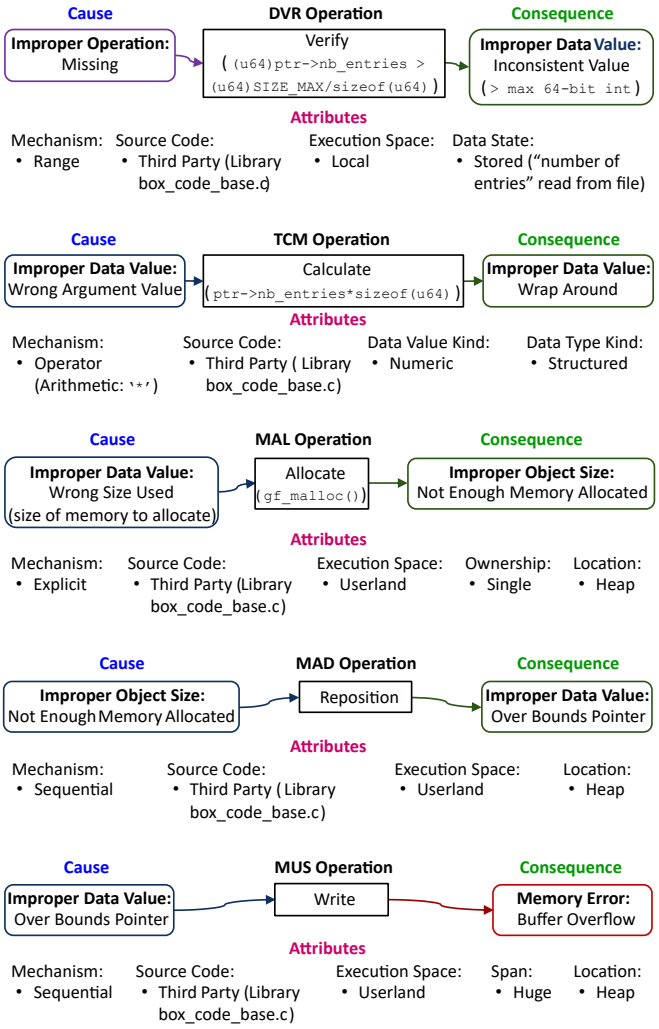


Fig. 10: BF for CVE-2021-21834.

C. CVE-2021-23440 and Type Mismatch bypassing Input Validation

Our third example illustrates again a pattern of weaknesses that appears in different vulnerabilities. This pattern is described by the same BF chain. Type mismatch bugs are common in languages that have different operators for loose comparison and strict comparison, such as JavaScript and PHP. In this bug, the programmer uses a loose comparison when it should be a strict comparison, leading to input validation bypass [22].

Using the BF classification, we describe this pattern of vulnerabilities via two BF chains. The first is an one-class chain, TCM (improper comparison: loose instead of strict). The second is a two-classes chain, TCV → DVL. That is, due to missing type conversion an improper validation (DVL) happens, allowing an attacker to create an injection error. This vulnerability could lead to a failure, such as denial of service or, even worse, (remote) code execution.

One concrete example is CVE-2021-23440. It was discovered by the Snyk Team with several other loose comparison bugs.

1) Brief Description: The package set-value for JavaScript is vulnerable to prototype pollution in versions

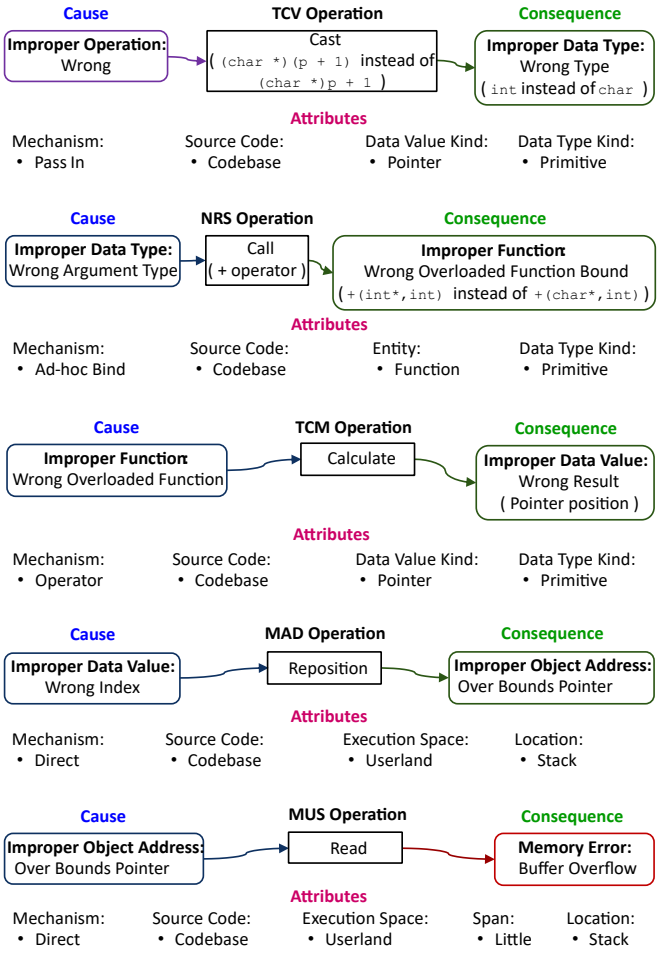


Fig. 11: BF for CWE-468, Example 1.

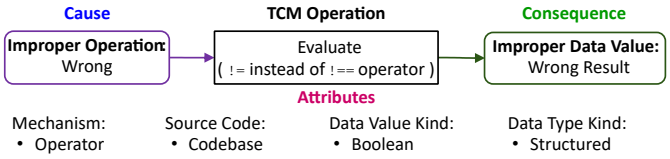
<2.0.1, >=3.0.0, <4.0.1. It happens due to a type mismatch in the prototype pollution verification.

2) Analysis: This vulnerability has two possible BF chains. The first chain is: the loose comparison operator is misused. The second chain is: a type conversion is missing before using the loose comparison, leading to improper input validation and prototype pollution, a kind of command injection in JavaScript [23]. Fig. 12 presents the BF taxonomy for this vulnerability.

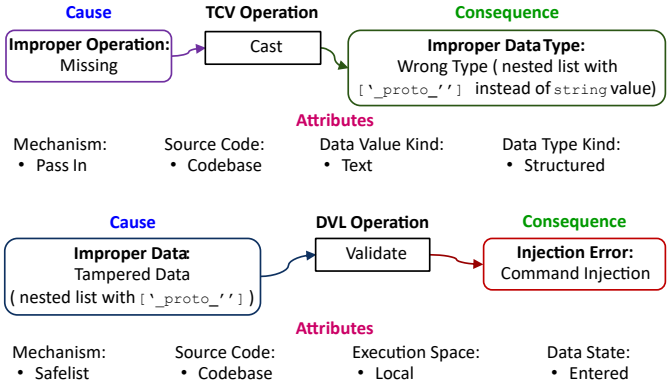
3) The Fix: There are two ways of fixing this vulnerability: the developers could use strict comparison, solving the TCM bug, or they could do a type conversion before the loose comparison. They chose the latter [24].

VIII. Related Works

The Bugs Framework (BF) is concerned with security-related bugs, i.e., bugs that open doors for exploitation. The work of Sun et al. [25] is the only one we found to establish a taxonomy classifying Data Type related security bugs. The classes in their taxonomy are called Fix Patterns. Although their paper is succinct, we understand that at least two Fix Patterns relate to the BF Data Type Bugs classes: COP (changing the type of an object to its parent class) and COS (changing the type of an object to its subclass) are related to TCV. Several other authors attempted to create successful taxonomies of bugs/weaknesses that lead to security failures.



(a) First BF chain: Loose comparison misuse.



(b) Second BF chain: Conversion missing before loose comparison.

Fig. 12: BF for CVE-2021-23440.

Hui et al. reviewed them in [26]. Unfortunately, none of those taxonomies tackle the specific problem of data type bugs. However, we identified several publications that discuss some of the underlying weaknesses of type-related vulnerabilities, such as type confusion, loose comparison, integer overflow, and type mismatch.

Type confusion vulnerabilities happen when a pointer is converted from a subclass pointer to a base class pointer. Their BF descriptions usually contain a NRS or a TCV class. Haller et al. [27] studied this kind of bugs and developed TypeSan, a practical type confusion detector for C++ language.

Loose comparison vulnerabilities happen when the developer wrongly uses the loose comparison feature of dynamically typed languages such as PHP and JavaScript. Their BF descriptions could be related to all BF Data Type Bugs classes. Li et al. [28] studied loose comparison bugs in PHP and proposed LChecker to detect this kind of bugs in PHP code.

Type mismatch vulnerabilities happen when an in argument or a return variable has an unexpected type. They also happen due to improper declarations. Their BF descriptions could be related to all BF Data Type Bugs classes. Gao et al. [29] have discussed type mismatch bugs and performed a detailed quantification of static type systems for JavaScript. Pradel et al. [30] also studied type mismatches in JavaScript and proposed a tool (TypeDevil) to detect it. Pascarella et al. [31] investigate how inconsistent documentation can lead to type mismatch bugs in Python.

Integer overflow vulnerabilities happen when a congruent arithmetic operation (modulo 2^n, for a n-bit integer) wraps around. Their BF descriptions relate to the TCM class. Dietz et al. [32] reviewed integer overflow in C/C++ language. Their work gives a good picture of the issue.

Although, the aforementioned papers have studied the bugs and weaknesses discussed in this paper, our work is considerably different. The current literature focuses on tools and procedures to detect a particular bug/weakness, while we are focusing on the theoretical side of all this. We are developing a classification system for all possible kinds of data type bugs/weaknesses and a methodology to describe the interrelationship between those kinds of weaknesses and other ones, such as memory corruption bugs and improperly checked input bugs. We believe that such a structured model for software bugs will allow the development of better automatic tools in the future.

In addition to CWE, to the best of our knowledge, there is no attempt to describe data type bugs/weaknesses. The relationship between the data type related CWEs and the BF classes are presented in section VI.

IX. Conclusion

In this paper, we introduce four new BF classes: Declaration Bugs (DCL), Name Resolution Bugs (NRS), Type Conversion Bugs (TCV), and Type Computation Bugs (TCM). We present their operations, possible causes, consequences, attributes, and sites. We show how they cover all CWEs related to conversion, calculation (incl. comparison), wrap-around (incl. integer overflow), pointer scaling, coercion (juggling), and other data type related weaknesses. We analyze particular data type vulnerabilities and provide their precise BF descriptions. The BF structured taxonomies show the initial error in code (the bug), providing a quite concise and still far more clear description than the unstructured explanations in current repositories, advisories, and publications. The BF Data Type Bugs taxonomy can be used by bug reporting tools, as it is a structured extension over the data type-related CWEs [4]. To our knowledge there is no other bug taxonomy that allows precise causal descriptions of data type related declaration, resolution, conversion, and computation bugs/weaknesses.

Future work should identify and describe more CVEs related to data type related declaration, resolution, conversion, and computation bugs/weaknesses, evaluating the BF Data Type Bugs taxonomy for usability. In such an evaluation, a machine learning algorithm or multiple analysts would classify and describe newly reported bugs [33], while helping improve BF's taxonomy by fine-tuning the classes.

BF has the expressiveness power to clearly describe any software bug or weakness, underlying any vulnerability. Precise BF descriptions of software vulnerabilities as chains of bug-weaknesses-failure will allow clear communication among software developers, testers, IT professionals, and IT managers. The vulnerability databases, such as NIST NVD and CISA KEV, will have CVE entries in machine readable formats that cybersecurity researchers can use for building code review tools and for a broad spectrum of ML and AI systems for detection of software vulnerabilities and exploring complex malicious attacks. This will aid better software development/coding practices, mitigation designs, automated cybertesting capabilities, and will greatly advance our way of securing the cyberspace and the critical infrastructure. The BF taxonomy will allow clear explanations of what happens in a vulnerability to IT professionals and non-IT executives, as well as researchers, developers, and students. It will support development of precise software testing tools with unambiguous reports.

BF aims to provide more than a classification system. It is a model of the underlying weaknesses, describing a sequence of causes and consequences, starting with the bug (the defect) to the final failure (the exploit). Although the entire framework is not ready yet, we believe that providing a model of the underlying weaknesses helps to clarify if a bug has been fixed or just mitigated, allowing correct risk assessment of vulnerabilities and minimizing incomplete fixes. A taxonomy providing cause→consequence relations should improve bug reports of security-related bugs, reducing developers' time to understand bug reports [34].

The CWE digraphs by BF class consequences should be deeply analyzed. Generation of digraphs with CWEs related to particular software errors (e.g., wrap-around), detecting corresponding clusters, and understanding their relationships would create a comprehensive view of the CWE model for researchers and practitioners. In turn, comparing and contrasting the CWE's exhaustive list of weaknesses with all the possible consequence-cause transitions to other BF classes would improve BF as a tool for describing CVEs.

We will continue developing orthogonal BF classes that cover and extend the CWE weakness types. The ultimate goal for BF is to help IT professionals understand how software vulnerabilities originate from bugs, propagate through weaknesses, and end as failures. The gain would be fewer bugs, fewer vulnerabilities, less time to fix/patch code, and better tools for bugs detection.

References

[1] *CVE-2022-1096*, Accessed: 2022-06-30, 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-1096.

[2] NVD, *National Vulnerability Database (NVD)*, Accessed: 2022-09-08, 2022. [Online]. Available: https://nvd.nist.gov.

[3] MITRE, *Common Vulnerabilities and Exposures (CVE)*, Accessed: 2022-09-08, 2022. [Online]. Available: https://cve.mitre.org/.

[4] MITRE, *Common Weakness Enumeration (CWE)*, Accessed: 2022-09-08, 2022. [Online]. Available: https://cwe.mitre.org.

[5] CISA, *Known Exploited Vulnerabilities Catalog*, Accessed: 2022-09-08, 2022. [Online]. Available: https://www.cisa.gov/known-exploited-vulnerabilities-catalog.

[6] D. Malzahn, Z. Birnbaum, and C. Wright-Hamor, "Automated vulnerability testing via executable attack graphs," in *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, IEEE, 2020, pp. 1–10. DOI: 10.1109/CyberSecurity49315.2020.9138852.

[7] NIST, *The Bugs Framework*, Accessed: 2022-09-08, 2022. [Online]. Available: https://samate.nist.gov/BF/.

[8] I. Bojanova, C. E. Galhardo, and S. Moshtari, "Input/output check bugs taxonomy: Injection errors in spotlight," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2021, pp. 111–120. DOI: 10.1109/ISSREW53611.2021.00052.

[9] I. Bojanova and C. E. Galhardo, "Classifying Memory Bugs Using Bugs Framework Approach," in *2021 IEEE 45th Annual Computer, Software, and Applications Conf. (COMPSAC)*, 2021, pp. 1157–1164. DOI: 10.1109/COMPSAC51774.2021.00159.

[10] M. Graff and K. R. Van Wyk, *Secure coding: principles and practices*. O'Reilly Media, Inc., 2003.

[11] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

[12] C. Munoz, "Type theory and its applications to computer science," *Quarterly News Letter of Institute for Computer Application in Science and Engineering (ICASE)*, vol. 8, no. 4, 2007. [Online]. Available: https://shemesh.larc.nasa.gov/fm/papers/ICASE1999-QNews.pdf.

[13] MITRE, *CWE CATEGORY: Numeric Errors*, Accessed: 2022-09-08, 2022. [Online]. Available: https://cwe.mitre.org/data/definitions/189.html.

[14] MITRE, *CWE CATEGORY: Type Errors*, Accessed: 2022-09-08, 2022. [Online]. Available: https://cwe.mitre.org/data/definitions/136.html.

[15] MITRE, *CWE CATEGORY: String Errors*, Accessed: 2022-09-08, 2022. [Online]. Available: https://cwe.mitre.org/data/definitions/133.html.

[16] S. Chaliasos, T. Sotiropoulos, G.-P. Drosos, C. Mitropoulos, D. Mitropoulos, and D. Spinellis, "Well-typed programs can go wrong: A study of typing-related bugs in jvm compilers," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021. DOI: 10.1145/3485500.

[17] *Static Analysis Tool Exposition (SATE)*, Accessed: 2022-09-08, 2021. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/samate/static-analysis-tool-exposition-sate.

[18] CISA, *ICS Advisory (ICSA-21-119-04), Multiple RTOS (Update E)*, Accessed: 2022-09-08, 2022. [Online]. Available: https://www.cisa.gov/uscert/ics/advisories/icsa-21-119-04.

[19] Cisco Talos, *Talos Vulnerability Report, TALOS-2021-1297 GPAC Project on Advanced Content library MPEG-4 Decoding multiple multiplication integer overflow vulnerabilities*, Accessed: 2022-09-08, 2021. [Online]. Available: https://talosintelligence.com/vulnerability_reports/TALOS-2021-1297.

[20] *GPAC*, Accessed: 2022-09-08, 2021. [Online]. Available: https://github.com/gpac/gpac/blob/v1.0.1/src/isomedia/box_code_base.c.

[21] A. David, *Fixes for talos report TALOS-2021-1297*, Accessed: 2022-09-08, 2021. [Online]. Available: https://github.com/gpac/gpac/commit/b515fd04f5f00f4a99df741042f1efb31ad56351.

[22] A. Della Libera, *JavaScript type confusion: Bypassed input validation (and how to remediate)*, Accessed: 2022-09-08, 2021. [Online]. Available: https://snyk.io/blog/remediate-javascript-type-confusion-bypassed-input-validation/.

[23] A. Sharma, *From Prototype Pollution to Full-on Remote Code Execution, How Can Adversaries Exploit npm Modules?* Accessed: 2022-09-08, 2020. [Online]. Available: https://blog.sonatype.com/how-can-adversaries-exploit-npm-modules.

[24] J. Schlinkert, *Security Fix for Prototype Pollution*, Accessed: 2022-09-08, 2021. [Online]. Available: https://github.com/jonschlinkert/set-value/pull/33/files#diff-e727e4bdf3657fd1d798edcd6b099d6e092f8573cba266154583a746bba0f346.

[25] X. Sun, X. Peng, K. Zhang, Y. Liu, and Y. Cai, "How security bugs are fixed and what can be improved: An empirical study with mozilla," *Science China Information Sciences*, vol. 62, no. 1, 019102:1–019102:3, 2019. DOI: 10.1007/s11432-017-9459-5.

[26] Z. Hui, S. Huang, Z. Ren, and Y. Yao, "Review of Software Security Defects Taxonomy," in *Rough Sets and Knowledge Technology*, 2010, pp. 310–321. DOI: 10.1007/978-3-642-16248-0_46.

[27] I. Haller, Y. Jeon, H. Peng, *et al.*, "Typesan: Practical type confusion detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 517–528. DOI: 10.1145/2976749.2978405.

[28] P. Li and W. Meng, "Lchecker: Detecting loose comparison bugs in php," in *Proceedings of the Web Conference 2021*, 2021, pp. 2721–2732. DOI: 10.1145/3442381.3449826.

[29] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: Quantifying detectable bugs in javascript," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 758–769. DOI: 10.1109/ICSE.2017.75.

[30] M. Pradel, P. Schuh, and K. Sen, "Typedevil: Dynamic type inconsistency analysis for javascript," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 314–324. DOI: 10.1109/ICSE.2015.51.

[31] L. Pascarella, A. Ram, A. Nadeem, D. Bisesser, N. Knyazev, and A. Bacchelli, "Investigating type declaration mismatches in python," in *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, IEEE, 2018, pp. 43–48. DOI: 10.1109/MALTESQUE.2018.8368458.

[32] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 25, no. 1, pp. 1–29, 2015. DOI: 10.1145/2743019.

[33] T. M. Adhikari and Y. Wu, "Classifying software vulnerabilities by using the bugs framework," in *8th Inter. Symp. Digital Forensics and Security (ISDFS)*, 2020, pp. 1–6. DOI: 10.1109/ISDFS49300.2020.9116209.

[34] S. Kim and E. J. Whitehead, "How long did it take to fix bugs?" In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, 2006, pp. 173–174. DOI: 10.1145/1137983.1138027.