NIST Special Publication
500-XXX

# Bugs Framework (BF)

## *Formalizing Cybersecurity Weaknesses and Vulnerabilities*

Irena Bojanova

This publication is available free of charge from:
https://doi.org/10.6028/NIST.XXX.XXXX

NIST NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

# NIST Special Publication 500-XXX

# Bugs Framework (BF)

## *Formalizing Cybersecurity Weaknesses and Vulnerabilities*

Irena Bojanova
*Software and Systems Division*
*Information Technology Laboratory*

This publication is available free of charge from:
https://doi.org/10.6028/NIST.XXX.XXXX

March 2024


U.S. Department of Commerce
*Gina M. Raimondo, Secretary*

National Institute of Standards and Technology
*Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology*

**NIST Technical Series Policies**
Copyright, Use, and Licensing Statements
NIST Technical Series Publication Identifier Syntax

**Publication History**
Approved by the NIST Editorial Review Board on YYYY-MM-DD

**How to cite this NIST Technical Series Publication:**
Irena Bojanova (2024) Bugs Framework (BF): Formalizing Cybersecurity Weaknesses and Vulnerabilities. (National Institute of Standards and Technology, Gaithersburg, MD), Publication Identifier. https://doi.org/10.6028/NIST.XXX.XXXX

**NIST Author ORCID iD**
0000-0002-3198-7026

**Contact Information**
irena.bojanova@nist.gov

**Public Comment Period**
Month Day, YYYY - Month Day, YYYY

**Submit Comments**
bf@nist.gov

## Abstract

The Bugs Framework (BF) is a classification of security bugs and related faults, featuring a formal language for unambiguous specification of security weaknesses and underlined by them vulnerabilities. It organizes bugs and faults by the operations of distinct software or hardware execution phases – as weakness causes, and the resulting errors propagating to other faults or causing failures – as weakness consequences. The phases do not overlap by operation, which guarantees complete orthogonal weakness types coverage (without gaps and overlaps) and unique precise weakness and vulnerability descriptions (with clear causality). The BF formal language is generated by the BF Left-to-right Leftmost-derivation One-symbol-lookahead (LL(1)) attribute context-free grammar (ACFG), based on the BF taxonomy, bugs models, and vulnerability models. This formalism enables a new range of research and development efforts for creation of comprehensively labeled weakness and vulnerability datasets, and diverse vulnerability classifications; as well as vulnerability specification generation, bug detection, and vulnerability analysis and remediation. The BF weakness and vulnerability specifications may serve as a formal augmentation to the Common Weakness Enumeration (CWE) and the Common Vulnerabilities and Exposures (CVE) natural language descriptions.

This Special Publication (SP) presents an overview on the Bugs Framework (BF). Further details will be available in NIST SP xxx-xxxA-I at https://csrc.nist.gov/publications/.

The expected audience is of security researchers, software and hardware developers, information technology (IT) managers, and IT executives. To our knowledge, the ideas, approach, and methodologies in which the BF formal language, models, tools, and datasets are being created and presented here are unique.

## Keywords

Bug; Bug Classification; Bug Detection; Bug Taxonomy; Bug Triaging; Code; CVE; CWE; Cybersecurity; Design; Exploitation; Exploitable Error; Failure; Fault; Firmware; Firmware Design; Firmware Specification; Formal Language; Formal Methods; Generation Tool; Hardware; Hardware Design; Hardware Specification; Labeled Dataset; LL(1) Grammar; Microcode; Microcode Design; Microcode Specification; NVD; Security; Security Bug; Error; Fault; Security Failure; Security Vulnerability; Security Weakness; Software Design; Software Specification; Specification; Vulnerability; Vulnerability Analysis; Vulnerability Dataset; Vulnerability Remediation; Vulnerability Resolution; Vulnerability Mitigation; Weakness Dataset.

# Table of Contents

# List of Figures

## 1. Introduction

Software and its underlying hardware are the foundation of every computer system in our interconnected world. They enable modern life and drive productivity, but also form the digital attack surface and the thread vectors used by malicious actors. Cybersecurity of critical infrastructure and supply chains are an increasingly pressing societal challenge, as attacks on cyberspace are not only growing, they are also more sophisticated and more dangerous – exploiting undetected software or hardware security weaknesses. Modern cybersecurity research and application must overcome them and assure security vulnerability prevention, remediation, or mitigation. However, the current state of art for specifying and labeling security weaknesses and vulnerabilities is not keeping up with their requirements.

The current state of the art in Cybersecurity involves use of weakness and vulnerability descriptions from the Common Weakness Enumeration (CWE) [1] and the Common Vulnerabilities and Exposures (CVE) [2] repositories. The National Vulnerabilities Database (NVD) [3] also labels the CVE entries (CVEs) with CWE entrees (CWEs). The Known Exploited Vulnerabilities Catalog (KEV) [4] lists publicly exploited CVE with top priority for remediation. However, the CWE and CVE descriptions are in natural language and far from formal. The CWE to CVE assignments could be challenging and ambiguous, as CWEs may be overly specific, unclear, or overlapping [5].

The CVE was initiated by MITRE in 1999 [6] to address the problem of having ”no common naming convention and no common enumeration of the vulnerabilities in disparate databases” [7]. The creation of CWE followed in 2006 to address ”the issue of categorizing software weaknesses” and establishing ”acceptable definitions and descriptions of these common weaknesses”; ”support for hardware weaknesses” was added in 2020 [8]. The CWE also adopted the list approach adding tree based abstractions. The CVE and CWE lists are regularly refined and new weakness types, publicly disclosed vulnerabilities, and related content are added [8]. Past classification efforts as ”the development of the periodic table of elements and the identification of animals” [7] may give a perspective. The CVE enumeration idea relates to what the classification of elements was before Mendeleev’s discovery of a systematic approach and methodology on how to organize the elements: ”... it is also clear that before such a table could be thought of, the elements first needed to be enumerated and named” [7]; ”The community may not agree on a classification scheme for vulnerabilities, but we probably know enough to begin to enumerate the vulnerabilities, independent of their classification. ... all that is necessary at this point is 1-dimensional representation, at least from our operational perspective. The use of more complex representations can wait ...” [7]. As of March, 2024, there are 938 weakness types enumerated in CWE and over 228 000 vulnerabilities enumerated in CVE; over 170 000 CVEs are labeled with CWEs by NVD.

While the natural language descriptions may still be useful to many, formal specifications would be valuable augmentation to the current state of the art CWE and CVE entrees. The adopted list (nomenclature) approach with descriptions in natural language, however has

proven to create challenges. Many CVE and CWE entrees have imprecise descriptions [9] with unclear causality and lack explainability [10]. Being an enumeration, the CWE also has gaps and overlaps in coverage. Many CVEs list final errors (the sinks) as root causes (, which should be the bugs). Although, CWE lists CVE examples, NVD was the first effort toward labeling CVEs, which potentially would allow their classification. However, due to the problems listed above, the CWE assignments to CVE entrees have proven to be not enough comprehensive nor clear/formal for the purposes of modern research, development, and application efforts – including using Machine Learning (ML) and Artificial Intelligence (AI). Cybersecurity is now at the stage where a new systematic approach and formal methodology for specifying weaknesses and vulnerabilities is a necessity. Clear, unambiguous formal descriptions would support efforts to understand, identify, prioritize, and resolve or mitigate security vulnerabilities. Comprehensively labeled datasets based on them would support development of new ML and AI enabled capabilities for securing the critical infrastructure and supply chains.

The Bugs Framework (BF) is being developed as a formal classification of security bugs and related faults enabling unambiguous specification of caused by them weaknesses and underlined by them vulnerabilities. Bugs and faults are organized by the operations of distinct execution phases – as weakness causes, and the operations erroneous results that become faults or result in failures – as weakness consequences. The phases do not overlap by operation, which guarantees complete orthogonal weakness types coverage (without gaps and overlaps) and unique precise weakness and vulnerability descriptions (with clear causality). Analogously to the periodic table BF allows identification/prediction of unencountered yet security weakness types. First ideas about formalizing software bugs [11] and a "Periodic Table" of bugs [12] started forming in 2014-2015. The first BF classes [13–15], however, had rather naive organization, as there was no clarity on what constitutes causes and consequences nor on weakness and vulnerability causation, propagation, chaining, and convergence (e.g., Buffer Overflow (BOF), Injection (INJ), and other classes from [13–15] had to be redeveloped); and there was no separation of failure classes (e.g., Information Exposure (IEX) [16] had to be reclassified and redeveloped). The systematic formal approach and methodology for organizing software (and from there hardware) security bugs were discovered via meticulous research in 2019 and the follow up publications [17–21] started demonstrating their application.

BF comprises:

- Strict *definitions* of security bug, exploitable error, weakness, and vulnerability; fault and error; and security failure in the context of cybersecurity, elucidating causation and propagation rules.
- Formal *bugs models* with possible flow of operations within and between related execution phases where bugs and faults could occur.
- Structured, complete, orthogonal by operation, context-free bugs and faults *taxonomies*.

- A formal *vulnerability state model* of an improper bug state propagating through fault states towards a failure(s).
- A formal *vulnerability specification model* of chained BF weakness instances leading to a failure(s).
- A *formal language* for unambiguous specification of weaknesses and vulnerabilities.
- *Databases* and *tools* facilitating generation of formal weakness and vulnerability specifications (including of CWEs and CVEs); and graphical representation.
- Comprehensively labeled *datasets* of weaknesses and vulnerabilities.
- Diverse security *vulnerability classifications*.

The BF formal language is generated by the BF Left-to-right Leftmost-derivation One-symbol-lookahead (LL(1)) attribute context-free grammar (ACFG) derived from the BF context-free grammar (CFG). Is lexis, syntax and semantics are bases on the bugs and vulnerability models, and the orthogonal (not overlapping by operation) bugs and faults taxonomies, utilizing strict BF concepts definitions of security bug, weakness, vulnerability and failure; and fault, error, and exploitable error. BF's formalism guarantees unique precise descriptions (with clear causality) of weaknesses (including CWE) and vulnerabilities (including CVE); and complete weakness types coverage (without the gaps and overlaps exhibited by CWE). It enables the creation of comprehensively labeled weakness and vulnerability datasets and formal security vulnerability classifications. Its strict methodologies for the chaining weaknesses underlying a vulnerability, their systematic labeling on any level of abstraction, and backwards bug identification from a security failure can support a new range of cybersecurity research and development efforts for vulnerability specification generation, bug detection, and vulnerability analysis and remediation.

BF can specifically benefit the creation of tools based on code analysis, ML, and AI for:

- Weakness and vulnerability specification generation
- Datasets of vulnerabilities generation
- Bugs detection (triaging)
- Vulnerability analysis and resolution

This SP presents an overview on the BF systematic approach and formal methodologies for classifying bugs and faults per orthogonal by operation execution phases, specifying weaknesses and vulnerabilities, generating comprehensively labeled weakness and vulnerability datasets and vulnerability classifications. Further details will be available in NIST SP xxx-xxxA-I at https://csrc.nist.gov/publications/:

- NIST SP xxx-xxxA BF Security Concepts
- NIST SP xxx-xxxB BF Bugs Models
- NIST SPs xxx-xxxCx BF _yyy Taxonomies, where _yyy is a BF Class Type ID
- NIST SP xxx-xxxD BF Vulnerability Models

- NIST SP xxx-xxxE BF Formal Language
- NIST SP xxx-xxxF BF Databases, Tools, and APIs
- NIST SP xxx-xxxG BF Weakness and Vulnerability Datasets
- NIST SP xxx-xxxI BF Vulnerability Classifications

The expected audience is of cybersecurity researchers, software and hardware developers, Information Technology (IT) managers, and IT executives.

## 2. Current State of the Art

The current state of the art in describing and mapping security weaknesses and vulnerabilities involves the Common Weakness Enumeration (CWE) [1], the Common Vulnerabilities and Exposures (CVE) [2], and the National Vulnerabilities Database (NVD) [3]. The Known Exploited Vulnerabilities Catalog (KEV) [4] is also tightly related to CVE.

CWE is a community-developed list of software and hardware weakness types with descriptions, examples, references, and mitigation strategies – each CWE entry is assigned a *CWE-x* ID (identifier), where *x* is of one to four digits. CVE is a catalog of publicly disclosed security vulnerabilities with descriptions and references – each CVE entry is assigned a *CVE-yyyy-x* ID, where *yyyy* of disclosure and *x* is a four or five digits unique for that year number. NVD maps CVEs to CWEs and assigns Common Vulnerability Scoring System (CVSS) [22, 23] severity scores. KEV is an organization of publicly exploited CVEs with top priority for remediation – they are not necessarily the most severe as many of those don not get exploited in the wild.

CWE and CVE are widely used by the security community at large. CWE includes useful information on modes of introduction, mitigation technique, detection modes, and demonstrative examples. However, CWE is a hierarchical structure with inter-dependent weakness types, which may be too broad (coarse-grained), not orthogonal (an exhaustive Cartesian product by operation an attributes), and ambiguous. CVE includes references to reports, proof or concept, and code. However, many CWE and CVE descriptions are not sufficient, accurate, and precise enough[9, 11, 13]; many have unclear causality [17–19]. CWE also has gaps and overlaps in coverage [17–19], some of which have been recently pointed by the Hardware CWE Special Interest Group (HW CWE SIG) [24]. Many CVEs list final errors (the sinks) as root causes (, which should be the bugs). These CWE and CVE challenges propagate also to NVD and may lead to imprecise or wrong CWE to CVE assignments by NVD or third-party analysts. Additionally, CWE and CVE do not exhibit strict methodologies for chaining the weaknesses underlying a security vulnerability, systematic comprehensive labeling, and backwards bug identification (tracking) from a security failure. There are no tools to aid users in the creation and visualization of weakness and vulnerability descriptions. These issues propagate also to NVD and KEV (see Fig. 1).

The imprecise descriptions and luck of explainability make CWE and CVE difficult to use in modern cybersecurity research [10]. Augmenting their natural language descriptions

| Repository Challenges | Imprecise Descriptions | Unclear Causality | Gaps in Coverage | Overlaps in Coverage | Wrong CVE to CWE mapping | No Tracking Methodology | No Tools |
|---|---|---|---|---|---|---|---|
| CWE | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| CVE | ✓ | ✓ | | | | ✓ | ✓ |
| NVD | ✓ | ✓ | | | ✓ | ✓ | ✓ |

**Fig. 1.** CWE and CVE Challenges, and related to them NVD Challenges.

with unambiguous formal specifications will make them more suitable to be used as comprehensively labeled datasets for training ML and AI models either [21].

An example of an imprecise CWE description is "CWE-502: *Deserialization of Untrusted Data: The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid.*" It is not clear what "*sufficiently*" means in the "*sufficiently verifying*" phrase. The "*verifying that data is valid*" phrase is also confusing. It mixes the notions of validation (syntax check) and verification (semantics check), for which BF defines two distinct bugs classes.[18]

Unclear causality in CVEs leads to wrong CWE assignments. For example, CVE-2018-5907 is described as "*Possible buffer overflow in msm_adsp_stream_callback_put due to lack of input validation of user-provided data that leads to integer overflow in all Android releases (Android for MSM, Firefox OS for MSM, QRD Android) from CAF using the Linux kernel.*" If carefully examined, this is luck of input validation leads to integer overflow and then to buffer overflow. [18] However, NVD labels it with CWE-190 – Integer Overflow or Wraparound, while the cause is CWE-20 – Improper Input Validation. The full chain is: CWE-20→CWE-190→CWE-119; the last one being – Improper Restriction of Operations within the Bounds of a Memory Buffer. In addition, with many CVEs the chains are incomplete and there is no way to go backwards from the failure and reveal the bug from a vulnerability description.

Some CWEs add information on possible causing weaknesses, which could be very useful but may mislead that these are the only possible causing weaknesses, and introduce terms that may confuse the understanding of the main weakness that CWE is meant to describe [25].

BF addresses the CWE and CVE challenges; it has the expressive power to formally describe any bug, fault or weakness underlying any vulnerability in the of context of cybersecurity. As a first step, it could be utilized to augment CWE and CVE, and from there NVD and KEV with unambiguous weakness/ vulnerability BF specifications.

## 3. BF Approach

The Bugs Framework (BF) approach is different from the exhaustive list approach exhibited by enumerations. BF is a formal classification! It organizes bugs and faults by the

operations of distinct software or hardware execution phases – as weakness causes, and the resulting errors propagating to other faults or causing failures – as weakness consequences. The phases do not overlap by operation, which guarantees complete orthogonal weakness types coverage (without gaps and overlaps) and unique precise weakness and vulnerability descriptions (with clear causality). An operation defines what the software or hardware does on an appropriate level of abstraction. A bug relates to an operation. A fault relates to an input operand(s) to an operation. An error relates to the result from a buggy operation or a faulty operand; it can become the fault for another operation. The bugs and the faults form the set of causes for a Bf class; the errors form the set its of consequences.

Examples of operations are dereference, write, or deallocate memory, but also data type related condition evaluation, or encryption of plain text. Examples of bugs are missing code (e.g., an entire operation is missing), erroneous code (e.g., a wrong operator is used), or wrong algorithm (e.g. wrong encryption algorithm). Examples of faults are dangling pointer, wrong type, or weak cryptographic key.

Each BF class is a taxonomic category of a weakness type, defined by a set of operations, a set of valid *cause→consequence* transitions, a set of attributes, and a set of sites. It is associated with a distinct phase of software or hardware execution, the operations specific for that phase and the operands required as input to those operations. Operations or operands improperness define the causes. A consequence is the result of the operation over the operands. It becomes the cause for a next weakness or is a final exploitable error, leading to a failure. The attributes describe the operations and the operands. They help us understand the severity of the bug or the weakness. The sites are places in code that should be checked for that class of bugs.

The BF specification of a particular weakness is based on one taxonomic BF class; it is an instance of that BF class with one cause, one operation, one consequence, and their attributes. The operation binds the *cause→consequence* causal relation within a weakness – e.g., deallocation via a dangling pointer leads to a failure known as double free (double deallocate). The BF specification of a vulnerability is a chain of such instances of underlying weaknesses and their *consequence→cause* transitions.

BF has the expressive power to clearly describe any security weakness as a *cause-operation-consequence* instance of a BF class and any vulnerability as a chain of underlying weaknesses leading to a failure. It addresses the CWE and CVE challenges via the BF formal language, which lexis, syntax and semantics are based on the structured, complete, orthogonal, and context-free BF class taxonomies, as well as on the bugs/faults models, and vulnerability models.

*Structured* means a weakness is described via one *cause*, one *operation*, one *consequence*, and *attributes* from the lists defining a BF class. This assures precise causal descriptions of weaknesses as BF *(cause, operation, consequence)* weakness triples, with attributes' informed severity.

*Complete* means BF has the expressive power to describe any security bug, fault, and weakness via BF taxons. This assures the BF weakness types have no gaps in coverage.

*Orthogonal* means the sets of operations of any two BF classes do not overlap; there is no use of exhaustive Cartesian product. This assures the BF weakness types do not have overlaps in coverage.

*Context-free* means an operation cannot have different meanings depending on the context. This assures BF is applicable for source code in any programming language for any platform, operating environment, or application technology.

### 3.1. BF Security Weakness

BF models a *weakness* as an improper state and its transition (see Fig. 3). The transition is to another weakness or to a failure.

An improper state is defined as an $(operation, operand_1, \cdots, operand_n)$ tuple with at least one improper element. The transition is defined by the erroneous result from the operation over the input operands – the output from the improper state.

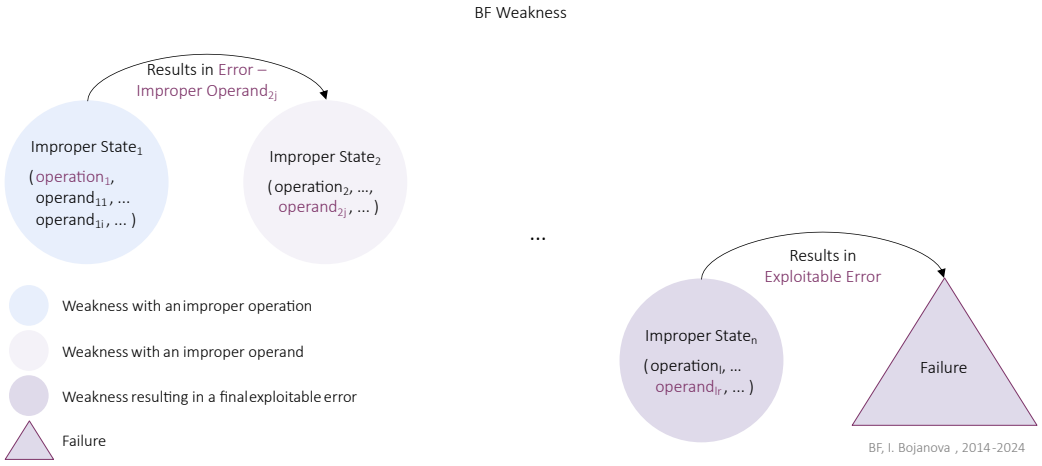



**Fig. 2.** BF Weakness. An improper state defined as an $(operation, operand_1, \cdots, operand_n)$ tuple with at least one improper element, and a transition defined by the erroneous result from the operation over the operands.

The improper operation or improper operand is the cause for the weakness. The improper result from an operation over its operands is the consequence from that weakness and it becomes a cause for a next weakness or a failure(s).

Figure 3 presents the possible BF weakness states – a bug weakness is depicted in blue, a fault weakness is depicted in purple.

A state is improper due to a security bug – the operation is improper, or due to a fault – an input operand is improper. The output from an improper state (the transition) can be

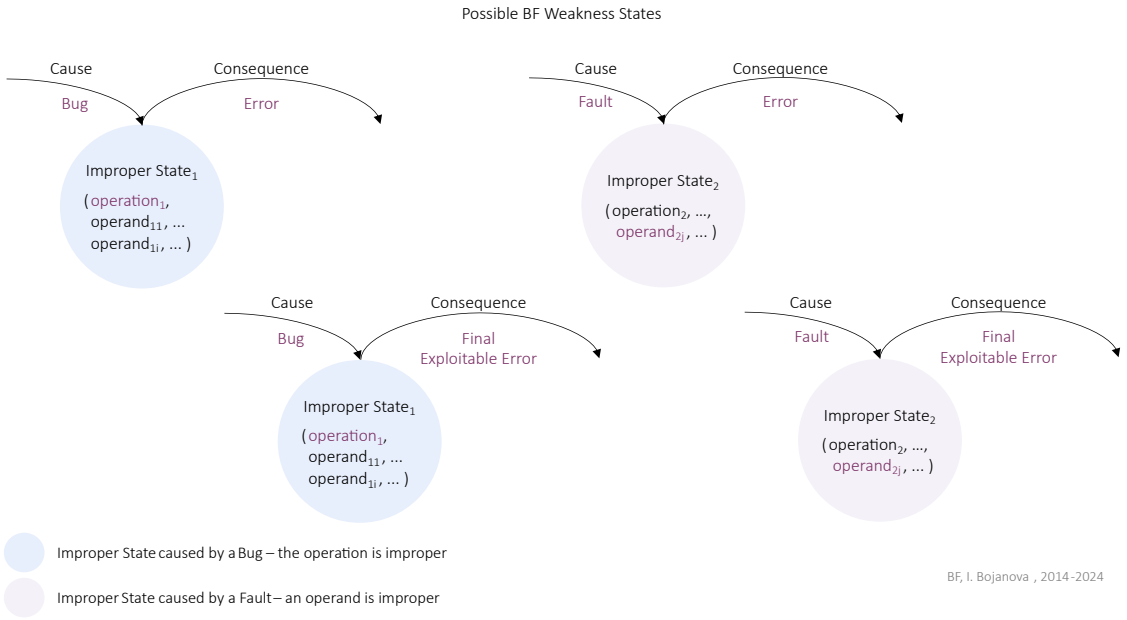



**Fig. 3.** Possible Weakness States. Formally they can be presented as *(bug, operation, error)*, *(fault, operation, error)*, *(bug, operation, exploitable error)*, and *(fault, operation, exploitable error)* weakness triples.

an error – that can propagate to a new fault, or a security exploitable error – an undefined system behavior. A security exploitable error usually directly relates to a CWE.

Formally the possible weakness states can be presented as textit(Bug, Operation, Error), *(Fault, Operation, Error)*, *(Bug, Operation, Exploitable Error)*, and *(Fault, Operation, Exploitable Error)* weakness triples.

A *security vulnerability* then is a chain of such weaknesses that starts with a bug, propagates through errors that become faults, and ends with an exploitable error. The bug must be fixed to resolve the vulnerability; fixing a fault will mitigate the vulnerability.

## 3.2. BF Security Vulnerability

BF models a *vulnerability* as a chain of improper states propagating towards a failure(s) as the erroneous result (*Error*) from one state becomes the improper operand (*Fault*) for the next state (see Fig. 4).

The initial state – depicted in blue – is always caused by a *Bug* – a code or specification defect within the operation. A propagation state – in light purple – is caused by at least one faulty (ill-formed) operand. The final state – in dark purple – results in a *Final Exploitable Error* (a undefined system behavior), which usually directly relates to a CWE, and leads to a failure.

An error is the result of an improper state from the operation over the operands. It becomes

an improper operand for a next improper state. For example, on Fig. 4, $Operation_1$ from $Improper\ State_1$ is improper, due to a Bug, and results in $Improper\ Operand_{2i}$, leading to $Improper\ State_2$. The last operation results in a Final Exploitable Error, leading to a failure.

The initial state is of an improper operation over proper operands; it is the state with the Bug in the operation, which must be fixed to resolve the vulnerability. A propagation state is of a proper operation over at least one improper operand; it is a state with a faulty operand, which if fixed would only mitigate the vulnerability.

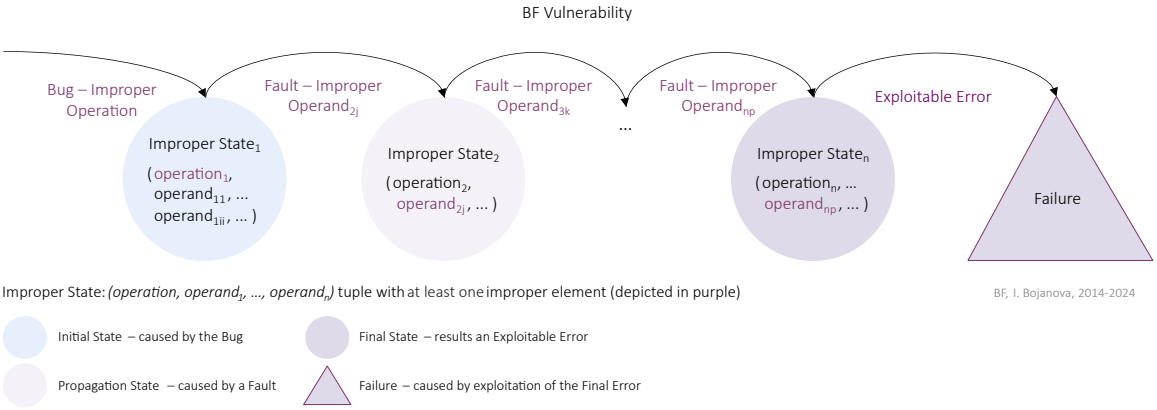



**Fig. 4.** BF Vulnerability. A chain of weaknesses as improper states propagating via *Error→Fault* transitions towards a security failure(s).

Vulnerabilities may also converge at their final states and chain via exploitation resulting faults.

### 3.3. BF Bugs Detection

Theoretically, the problem of detecting a Bug in software or hardware would be of generating the graph of all possible vulnerability chains of weaknesses, searching the graph via a brute force recursive backtracking algorithm with specific constraints to find the set of possible valid paths, and eventually via code analysis select the only proper path solutions.

However, as a formal language generated by an LL(1) CFG, BF supports the superior approach where a recursive-descent LL(1) parser does not require backtracking. Knowing the failure(s) and all the possible transitions at execution that adhere to the BF causation within a weakness, BF causation between weaknesses, and BF propagation rules (see Sec. 6.2), the Bug can be identified (see Fig. 5) by simply going backwards by operand until an operation is improper. Fixing the bug within that operation would resolve the security vulnerability.

Using the BF formal language syntax and semantics (the BF taxonomies and the BF causation and propagation rules) a state tree (a undirected graph with exactly one simple path

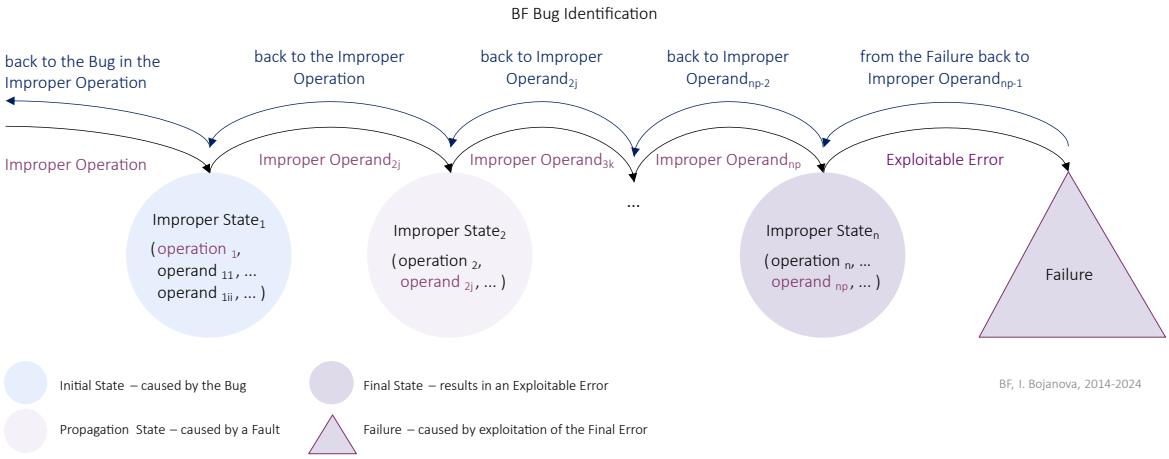



**Fig. 5.** BF Bug Identification. Going backwards from a Failure through Faults to the Bug.

between any pair of nodes) can be directly generated backwards starting from a Failure and a Final Exploitable Error. The Failure is the root of the tree, the paths are reverted possible vulnerability chains of weaknesses from the Exploitable Error through Faults to Bugs, where a weakness is a *(Bug, Operation, Error)*, *(Fault, Operation, Error)*, *(Bug, Operation, Exploitable Error)*, and *(Fault, Operation, Exploitable Error)* triple. Thus allowing generation of all possible BF specifications for a particular CVE.

## 4. BF Security Concepts

A *security bug* or *fault* type relates to a distinct phase of software, firmware (including microcode) or hardware (e.g., electronic circuits logic) execution, which is defined by a set of operations and their input operands and output results.

BF defines the concepts of bug, fault, error, exploitable error, weakness, vulnerability, and failure in the context of cybersecurity as follows.

- A *security bug* is a code or specification defect (an operation defect) – proper operands over an improper operation. It is a software, firmware, or hardware (circuit logic) implementation defect that may result also from a design flaw. System configuration or environment change may resurface new bugs.

  For example, software may run perfectly on a 64-bit operating system (OS) environment but exhibit a security bug on a 32-bit platform leading to buffer overflow.

- A *fault* is a name, data, type, address, or size error (an operand error) – an improper operand(s) over a proper operation. *Name* is about a resolved or bound object, function, data type, or namespace; *data*, *type*, *address*, and *size* are about an object. The error could result from a physical hardware defect.

For example, wrong data could result from missing validation or from an erroneous calculation, but also – from an exposed to high temperature melting chip damaging the data. As another example, speculative execution in shared environment may allow covert side channels.

- A *error* is a result from an operation with a bug or an operation with a faulty operand that can propagate to a new fault.
- A *security exploitable error* is an undefined system behavior resulting from an operation with a faulty operand.

  For example, integer overflow, query injection, buffer overflow.
- A *security weakness* is a *(bug, operation, error)*, *(fault, operation, error)*, *(bug, operation, exploitable error)*, or *(fault, operation, exploitable error)* triple; i.e., it is of a bug type – a bug causes an error, or of a fault type – a fault causes an error or an exploitable error. A bug informs the operation is improper; a fault informs about an improper operand.
- A *security vulnerability* is a chain of weaknesses that starts with a bug, propagates through errors that become faults, and ends with an exploitable error. The bug must be fixed to resolve the vulnerability; fixing a fault will mitigate the vulnerability.
- A *security failure* is a violation of a system security requirement caused by an adversary attack leveraging an exploitable error.

  For example, information exposure (*confidentiality loss*), data tempering (*integrity loss*), denial of service (*availability loss*), arbitrary code execution (*everything could be lost*).

  A failure may result in a fault, causing a new vulnerability of only fault type weaknesses. Fixing the bug in the first vulnerability will resolve the chain of vulnerabilities.

  Occasionally, for an exploit to be harmful, several vulnerabilities must converge at their exploitable errors. The bug in at least one of the chains must be fixed to avoid the failure.

For details on the BF concepts definitions refer NIST SP xxx-xxxA BF Security Concepts.

## 5. BF Taxonomy

The BF taxonomy structure is based on orthogonal by operations phases of software, firmware (including microcode), and hardware execution.

The BF bugs and faults landscape covers the operations in software, firmware, and hardware execution phases on appropriate levels of abstraction. The software and firmware

operations relate to code of applications, libraries, utilities, programming languages, services, operating systems. The hardware operations relate to electronic circuits logic, which adhere to the same input-process-output model as the software and firmware. Some phases of execution may be only on applications level (e.g., input/output check), others may cover more levels of abstraction (e.g., the programming language type system or the OS file system). In any case, if there is a failure, there must have been an operation that has a bug propagating through faulty operands of other operations until a final exploitable error (leading to the failure) is reached.

The BF Bugs Models represent related execution phases with their proper operations flow. The BF Class Taxonomies define the possible bugs and faults as causes for the operations of a specific phase to result in errors and execution errors as consequences.

### 5.1. BF Bugs Models

The BF Bugs Models present related execution phases with the operations where particular types of bugs could occur and the operations flow for faults propagation towards failures.

For example, memory bugs (see Fig. 6) could be introduced at any of the phases of an object's life-cycle: *address formation*, *allocation*, *use*, and *deallocation*. The phases determine the BF Memory Corruption/Disclosure Bugs classes: Memory Addressing Bugs (MAD), Memory Management (MMN) – combining the MAL and MDL phases, and Memory Use Bugs (MUS). Each data type memory bug or weakness involves one memory operation: *Initialize Pointer*, *Reposition*, *Reassign*, *Allocate*, *Extend*, *Reallocate–Extend*, *Initialize Object*, *Read*, *Write*, *Clear*, *Deallocate*, *Reduce*, and *Reallocate–Reduce*.

The phases define BF classes that do not overlap in operations. These BF Bugs Models form the basis for defining secure coding principles, such as data type safety, input/output safety, and memory safety.

As another example, Data Type bugs (see Fig. 7) could be introduced at any of the *declaration* (DCL), *name resolution* (NRS), *data type conversion* (TCV), or *data type related computation* (TCM) phases. The phases determine the BF Data Type Bugs classes Declaration Bugs (DCL), Name Resolution Bugs (NRS), Type Conversion Bugs (TCV), and Type Computation Bugs (TCM). Each data type related bug or weakness involves one data type operation: *Declare*, *Define*, *Refer*, *Call*, *Cast*, *Coerce*, *Calculate*, or *Evaluate*.

The Bugs Models also specify the possible flow of operations within and between closely related execution phases where related bugs or faults could occur, thus defining part of the causation between weaknesses rules that help identifying possible chains of bugs/weaknesses. For example, the possible flow between data type related operations is depicted in Fig. 7 with purple, blue, and green colored arrows. A declared and defined entity is referred in source code via its name. Names, referred to in remote scopes, get resolved via 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 con-

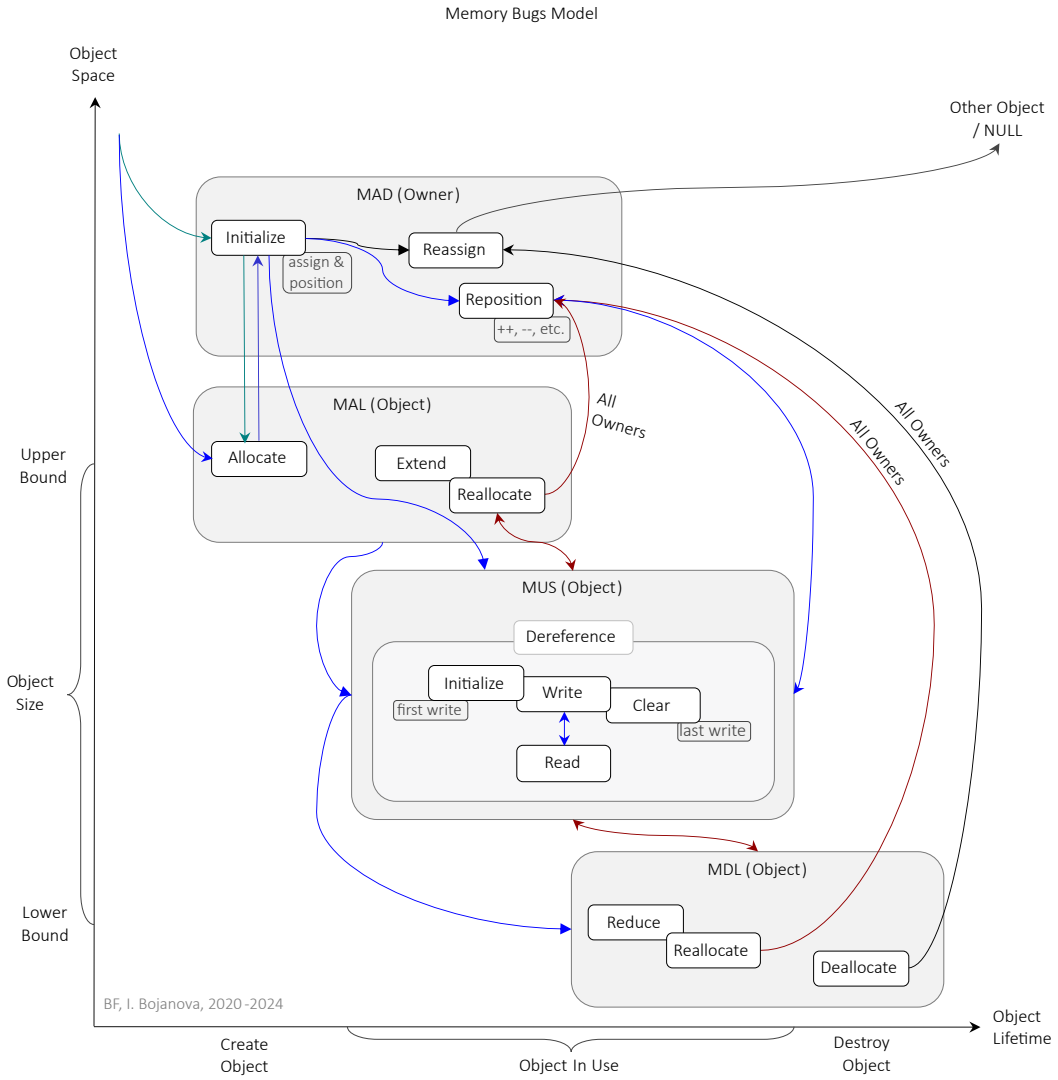



**Fig. 6.** BF Memory Bugs Model. Shows the memory related software/firmware execution phases with non-overlapping operations where bugs or faults could happen, and the operations flow.

verted 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.

As another example, the possible flow between memory related operations is depicted in

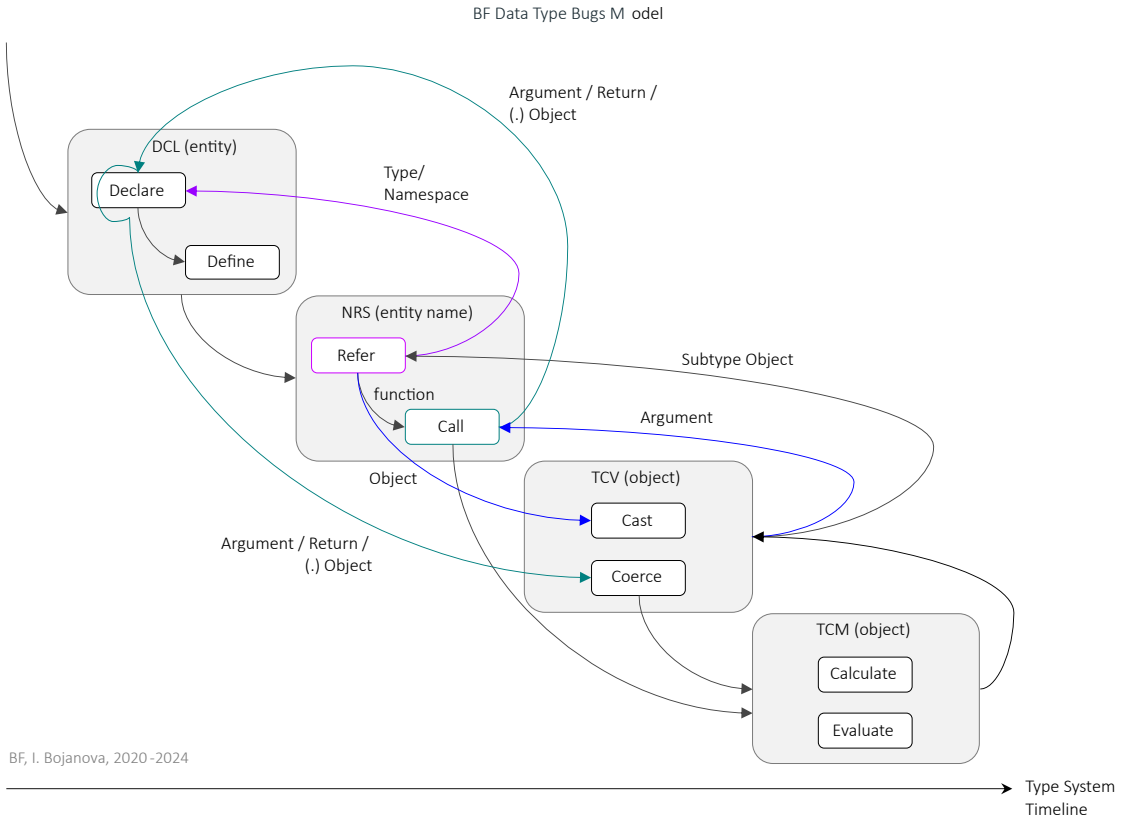



**Fig. 7.** BF Data Type Bugs Model. Shows the data type related software/firmware execution phases with non-overlapping operations where bugs or faults could happen, and the operations flow.

Fig. 6 with blue, green, and red colored arrows. Blue is for the main flow; green is for allocation requested at a specific address; red is for the extra flow in case of reallocation. Following the blue arrows, the very first operation is MAL Allocate an object. Following the green arrows, the first operation is MAD Initialize a pointer. Next operation, following the blue arrows, should be MAD Initialize the pointer to the address returned by Allocate. While, following the green arrows, next operation should be MAL Allocate an object at the address the pointer holds. After an object is allocated and its pointer is initialized, it can be used via MUS Read or MUS Write. The boundaries and the size of an object are set at allocation, then they can be changed by any MAL or MDL operation. If an object is owned by more than one pointer, Reallocate (in MAL or MDL) should be followed by Reposition over all these owners. A Deallocate an object operation should properly be followed by Reassign of all its pointers to either *NULL* or another object.

Each Bugs Model operation flow also defines semantic rules for causation between weaknesses of same BF class type, which allows identification of possible sub-chains of weaknesses.

For the complete BF Bugs Model combining all BF Bugs Models and specifying the flow between their operations refer NIST SP xxx-xxxC BF Bugs Models.

## 5.2. BF Classes

The BF taxonomy comprises weakness and failure categories (see its XML representation structure on Fig. 8, query it via the BF API). The weakness category comprises of bugs/faults class types – e.g.,:

- *_DAT* – Bugs/faults allowing a type compute exploit.
- *_INP* – Bugs/faults allowing an injection exploit.
- *_MEM* – Bugs/faults allowing a memory corruption/disclosure exploit.

The failure category comprises the Failure (_FLR) class type – loss of a security property due to the exploit of a vulnerability.

A BF class type encompasses bugs classes of closely related execution phases. For example, _DAT groups the following BF classes:

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

Then, _INP encompasses the Data Validation (DVL) and Data Verification (DVR) bugs classes; _MEM groups the Memory Addressing (MAD), Memory Management (MMN), Memory Use (MUS) classes, etc.

The _FLR class type encompasses the failure classes, e.g.,:

- *Information Exposure (IEX)* – Unauthorized disclosure of information – confidentiality loss.
- *Arbitrary Code Execution (ACE)* – Execution of unauthorized commands or code execution – everything could be lost. Remote Code Execution (RCE) is a kind of ACE, where arbitrary code is executed on a target system or device from a remote location, typically over a network.
- *Denial of Service (DOS)* – Disruption of access/use to information or information system (service) – availability loss.

```
<!--Bugs Framework (BF), I. Bojanova, 2014-2024-->
<BF Name="BF" Title="Bugs Framework">
  <Category Name="Weakness" Definition="A software security weakness is a (bug, operation, error) or
            (fault, operation, error) triple. It is an instance of a weakness type that relates to
            a distinct phase of software execution, the operations specific for that phase and
            the operands required as input to those operations.">
      <ClassType Name="_DAT" Title="Data Type" Definition="Data Type (_DAT) class type -
                 Bugs/Faults allowing a type compute exploit.">
          <Class Name="DCL" Title="Declaration" Definition="Declaration (DCL) class - An object,
                 a function, a type, or a namespace is declared or defined improperly.">
              <Operations>...</Operations>
              <Operands>...</Operands>
              <Causes>
                  <BugType Name="Code Defect" Definition="Code Bug type - Defect in the implementation
                           of the operation - proper operands over an improper operation.
                           A first cause for the chain of weaknesses underlying a software security
                           vulnerability. Must be fixed to resolve the vulnerability.">
                      <Bug Name="Missing Code" Definition="The entire oper">...</Bug>
                      <Bug Name="Wrong Code" Definition="An inappropriat">...</Bug>
                      <Bug Name="Erroneous Code" Definition="The operation i">...</Bug>
                  </BugType>
                  <BugType Name="Specification D" Definition="An error in the">...</BugType>
                  <FaultType Name="Type" Definition="The set or rang">...</FaultType>
              </Causes>
              <Consequences>...</Consequences>
              <Sites>...</Sites>
          </Class>
          <Class Name="NRS" Title="Name Resolution" Definition="The name of an ">...</Class>
          <Class Name="TCV" Title="Type Conversion" Definition="Data are conver">...</Class>
          <Class Name="TCM" Title="Type Computatio" Definition="An arithmetic e">...</Class>
      </ClassType>
      <ClassType Name="_INP" Title="Input/Output Check" Definition="Input/Output Check (_INP) class type -
                 Bugs/Faults allowing an injection exploit.">
      <Class Name="DVL" Title="Data Validation" Definition="Data are valida">...</Class>
      <Class Name="DVR" Title="Data Verificati" Definition="Data are verifi">...</Class>
    </ClassType>
    <ClassType Name="_MEM" Title="Memory Corruption/Disclosure" Definition="Memory Corruption/Disclosure
               (_MEM) class type - Bugs/Faults allowing a memory corruption/disclosure exploit.">
      <Class Name="MAD" Title="Memory Addressi" Definition="The pointer to ">...</Class>
      <Class Name="MMN" Title="Memory Manageme" Definition="An object is al">...</Class>
      <Class Name="MUS" Title="Memory Use" Definition="An object is in">...</Class>
    </ClassType>
    ...
  </Category>
  <Category Name="Failure" Definition="A security failure is a violation of a system security requirement.
            It is caused by the exploit of the final error of a vulnerability.
            In some cases, the final errors of several vulnerabilities must converge for
            the exploit to cause a failure.">
    <ClassType Name="_FLR" Title="Security Failure" Definition="Loss of a security property due
               to the exploit of a vulnerability.">
      <Class Name="IEX" Title="Information Exp" Definition="Unauthorized di">...</Class>
      <Class Name="ACE" Title="Arbitrary Code " Definition="Execution of un">...</Class>
      <Class Name="DOS" Title="Denial of Servi" Definition="Disruption of a">...</Class>
      <Class Name="TPR" Title="Data Tempering" Definition="Unauthorized mo">...</Class>
        ...
    </ClassType>
  </Category>
</BF>
```

**Fig. 8.** BF Taxonomy Structure – a weakness category with bugs/faults class types, and a failure category.

- *Data Tempering (TPR)* – Unauthorized modification or destruction of information – integrity loss.

A BF bugs class relates to a distinct phase of software, firmware, or hardware execution, the operations specific for that phase, the operands required as input to the operations, and the results as output from the operation over the operands.

Each BF class taxonomy is defined by a set of *operations*, their *consequence*→*cause* transitions, *attributes*, and *sites*. Operations or operands improperness define the causes. A consequence is the result of an operation over its operands. It becomes the cause for a next weakness or is a final exploitable error, leading to a failure(s). The attributes describe the operations and the operands, and help in identifying the severity of the bug or weakness. The sites show where in code a bug might occur.

For example, Fig. 9 depicts the BF Type Conversion (TCV) class [19], covering software security bugs/faults, causing error such as *Flipped Sign* and *Truncated Value*; Fig. 10 depicts the BF Data Validation (DVL) class [18], covering software security bugs/faults, causing exploitable errors such as *Query Injection* and *Source Code Injection*; and Fig. 11 depicts the BF Memory Use (MUS) class [17] , covering software security bugs/faults, causing exploitable errors such as *Use After Deallocate* (e.g., *Use After Free* and *Use After Return*) and *Buffer Overflow*.

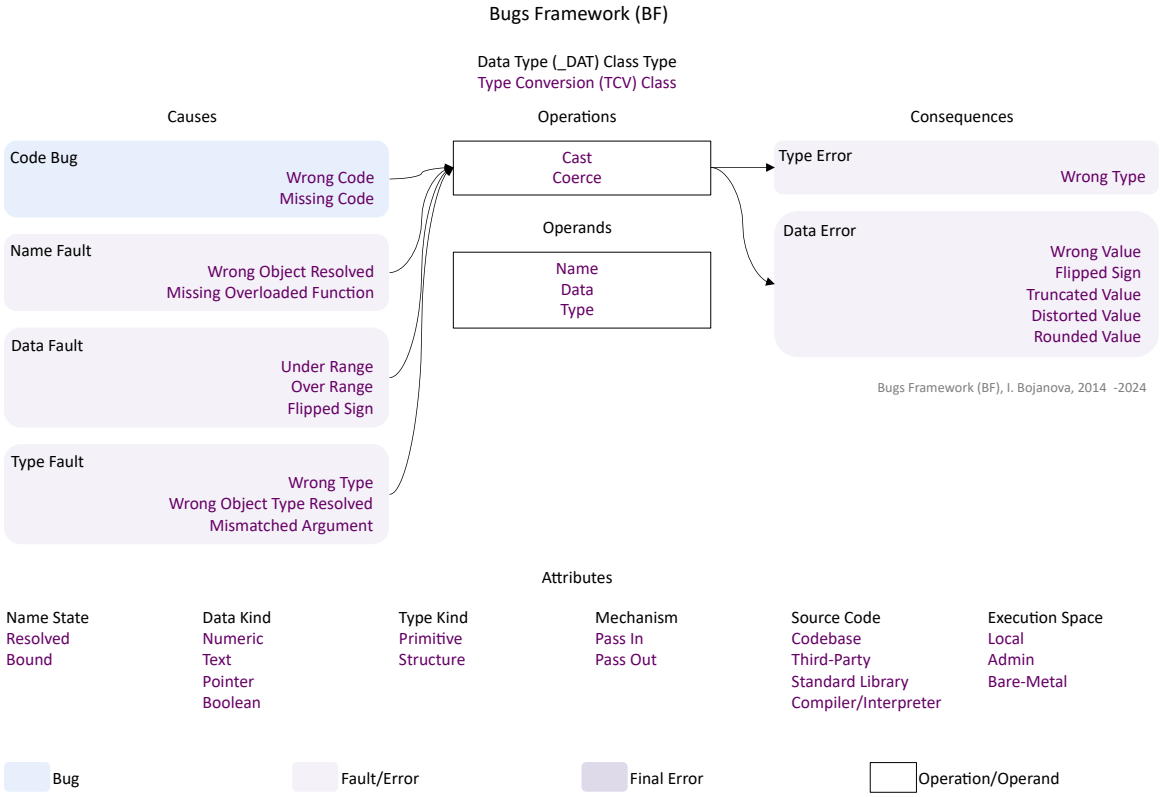



**Fig. 9.** The BF Type Conversion (TCV) class of the BF Data Type (_DAT) class type.

The specification of a security weakness is based on one taxonomic BF bugs/fault class. It is formalized as an instance of a BF class with one cause, one operation, one consequence,

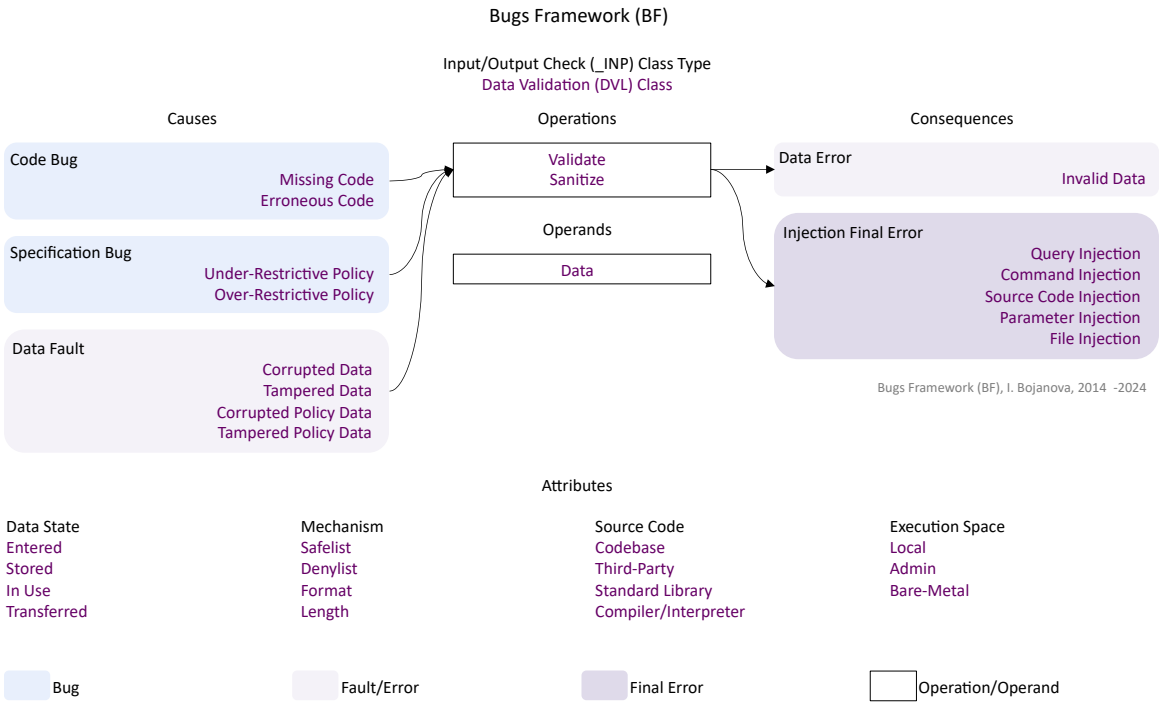



**Fig. 10.** The BF Data Validation (DVL) class of the BF Input/Output (_INP) class type.

and their attributes – selected from the values of the causes, operations, consequences, and attributes taxons of that class. The operation binds the *cause→consequence* relation – e.g., *Missing Validate* may lead to the final exploitable error known as *Query Injection*; *Write* via *Over Bounds Pointer* leads to the final exploitable error known as *Buffer Overflow*.

Each BF class strictly defines value taxons for class and operations; and type and value taxons for causes (as bugs or faults), consequences (as errors or exploitable errors), and operation and operand attributes (e.g., see the purple terms on Fig. 9, 10, and 11)

For example, the definitions of the taxon types *Code Bug* type and *Mechanism* operation attribute type are as follows.

- *Code Bug* – Defect in the implementation of the operation – proper operands over an improper operation. A first cause for the chain of weaknesses underlying a security vulnerability. Must be fixed to resolve the vulnerability.
- *Mechanism* – Shows how the buggy/faulty operation code is performed.

As further example, the definitions of the taxon values *Data Validation (DVL)* class, *Missing Code* bug, *Corrupted Data* fault, *Validate* operation, and *Query Injection* final exploitable error (see them on Fig. 11) are as follows.

- *Data Validation (DVL)* – Data are validated (syntax check) or sanitized (escape, filter, repair) improperly.

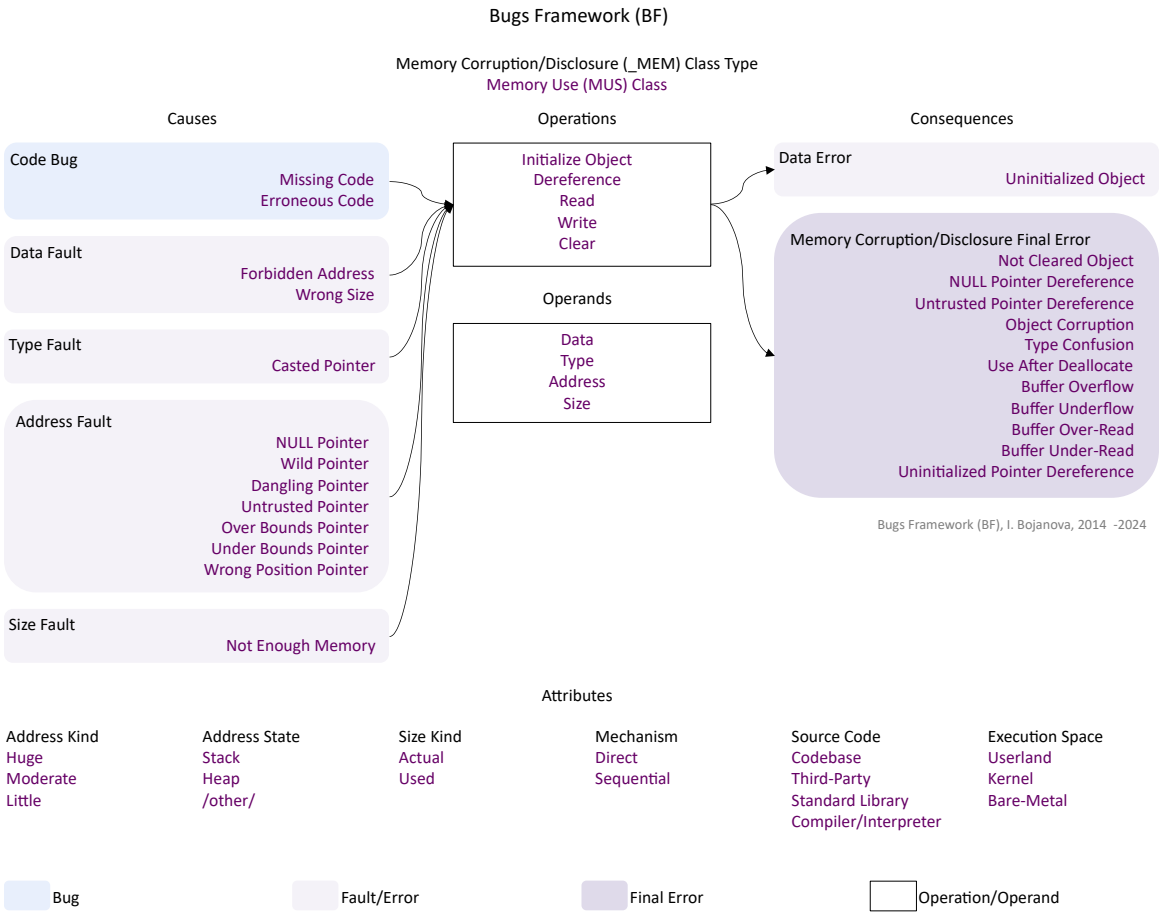



**Fig. 11.** The BF Memory Use (MUS) class of the BF Memory Corruption/Disclosure (_MEM) class type.

- *Corrupted Data* – Unintentionally modified data due to a previous weakness (e.g., with a decompress or a decrypt operation); would lead to invalid data for next weakness.
- *Missing Code* – The operation is entirely absent.
- *Validate* – Check data syntax (proper form/grammar, incl. check for missing symbols/elements) in order to accept (and possibly sanitize) or reject it.
- *Query Injection* exploitable error – Maliciously inserted condition parts (e.g., or $1 == 1$) or entire commands (e.g., *drop table*) into an input used to construct a database query. Examples: *SQL Injection*; *No SQL Injection*; *XPath Injection*; *XQuery Injection*; *LDAP Injection*.

As another example, the definitions of the taxon values *Memory Use (MUS)* class, *Wrong Size* cause, *Wrong Write* cause, and *Buffer Overflow* final exploitable error (see them on Fig. 11) are as follows.

- *Memory Use (MUS)* - An object is initialized, read, written, or cleared improperly.
- *Wrong Size* - The value used as size does not match the actual size of the object.
- *Write* – Change the data value of an object to another meaningful value.
- *Buffer Overflow* – Writing above the upper bound of an object – aka Buffer Over-Write.

The definitions are part of the machine readable representations (e.g., in XML – see Fig. 8) of the BF taxonomy. They are visualized also as tooltips of the BF Tool [26] and under the BFCVE specifications on the BF Website [27].

Each BF class taxonomy also defines semantic rules for causation within a weakness as matrices of meaningful *(bug/fault, operation, error/exploitable error)* triples. For example, *(Wrong Size, Write, Buffer Overflow)* is a valid weakness triple, while *(Wrong Size, Write, Buffer Over-Read)* is not.

For the complete set of developed BF class types, classes, taxon definitions, and semantic matrices refer NIST SPs xxx-xxxCx BF _yyy Taxonomies, where _yyy is a BF Class Type ID. For the complete BF formal language lexis refer NIST SP xxx-xxxE BF Formal Language.

### 5.3. BF Methodology

The methodology for developing BF classes is as follows (see Fig. 12).

1. Phases: Identify related software, firmware, or hardware phases in which specific kinds of bugs could be introduced. Each phase would define a new BF class. The BF classes of related phases define a new BF Class Type. For example, the Memory Addressing (MAD) and the Memory Use (MUS) BF classes (see Fig. 11) corresponds to the related memory addressing and memory use software or firmware execution phase. They are also of the Memory Corruption/Disclosure Bugs Class Type.
2. Operations: Identify the operations for that phase – these would define the possible *operation* values in the *(cause, operation, consequence)* weakness triples for this new BF class. For example, the Memory Use (MUS) BF class has five operations (see Fig. 11).
3. Bugs Model: Define the BF Bugs model of operations flow for these related phases. For example, the BF Memory Bugs Model covers the memory use phase (see the MUS block on Fig. 6). Usually, a Bugs Model covers the operation flow for two or more phases/classes of the same BF class type. For example, the BF Memory Bugs Model (see Fig. 6) covers the memory addressing (MAD), memory allocation (MAL), Memory Use (MUS), and Memory Deallocation (MDL) phases.
4. Bug Causes: Identify all code/specification defects – these would define the possible

*Bug* cause values for the weakness triples. For example, MUS has three possible values of *Code Defect Bug* type (see Fig. 11).

5. Fault Causes: Identify the operands (inputs) for the operations – these would define the cause/ consequence (*Fault*/*Error*) types. For example, see the *Data Fault*, *Type Fault*, *Address Fault*, *Size Fault*, *Data Error*, and *Memory Corruption/Disclosure Final Exploitable Error* types listed for MUS on Fig. 11. Identify all operand errors – these would define the possible *Fault* cause values for the weakness triples. For example, there are 26 *Fault*/*Error* values for MUS on Fig. 11.
6. Error Consequences: Identify all result (output) errors from the operations that propagate as causes for other weaknesses (t.e., improper input operands for other operations) – these would define the possible *Error* consequences in the weakness triples.
7. Final Exploitable Error Consequences: Identify all result (output) errors from the operations that do not propagate as causes for other weaknesses – these would define the possible *Final Exploitable Error* consequences for the weakness triples.
8. Operation Attributes: Identify specific descriptive values for the following operation attribute types.
9. Operand Attributes: Identify specific descriptive values for each operand type *Kind* and *State* attribute types.
10. Sites: Identify possible sites in code for such bugs/faults - a step applicable mainly for low level bugs.

Finally, create the BF class taxonomy in machine readable formats and generate graphical representation for enhanced understanding.

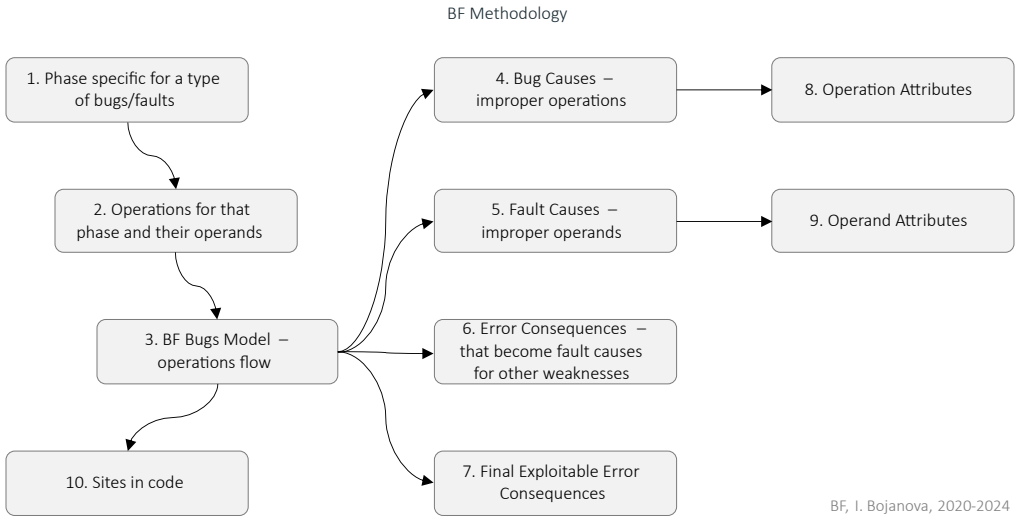



**Fig. 12.** BF Methodology for developing a BF class.

## 6. BF Vulnerability Models

The BF Vulnerability Models represent an improper states view of possibly converging and chaining vulnerabilities and a BF taxonomy based specification view.

### 6.1. BF Vulnerability State Model

BF models a *vulnerability* (see Fig. 13) as a deterministic state automata with a set of states, each defined as a $(operation, operand_1, \cdots, operand_n)$ tuple with at least one improper element (depicted in purple), and transitions for chaining weaknesses (depicted with $\curvearrowright$).

The *Initial State* corresponds to a weakness caused by a *Bug* in the operation, resulting in an Error. A *Propagation State* corresponds to a weakness caused by a *Fault* of an operand, resulting in an *Error*. The *Final State* corresponds to a weakness caused by a *Fault* of an operand, resulting in an Exploitable Error. The Failure is a violation of a security requirement caused by an exploit.

Fixing the Bug – the operation implementation or specification defect – will resolve the vulnerability; fixing a fault would mitigated it. Fixing a bug may include fixing first a design flaw, including unaccounted system configuration or environment. Fixing a fault may involve fixing first a hardware defect.

Occasionally, for an exploit to be harmful, several vulnerabilities must converge (depicted with $\oplus$) at their exploitable errors. The bug in at least one of the chains must be fixed to avoid the failure.

A failure may result in a fault, causing a new vulnerability of only fault type weaknesses. Fixing the bug in the first vulnerability will resolve the chain of vulnerabilities.

### 6.2. BF Vulnerability Specification Model

BF models a *vulnerability specification* (see Fig. 14) as a chain of *(cause, operation, consequence)* weakness triples with operation and operand attributes, and transitions adhering to the BF weakness and vulnerability causation and propagation rules. It reflects the BF Taxonomy structure (see Sec. 5.2) and the BF Vulnerability State Model chaining and convergence.

Causation within a weakness is by meaningful *(cause, operation, consequence)* weakness triples defined for each BF taxonomy – the bug or faulty input operand of an operation results in an error or a final exploitable error. More specifically, as matrices of meaningful *(bug, operation, error)*, *(fault, operation, error)*, *(bug, operation, exploitable error)*, or *(fault, operation, exploitable error)* triple weakness triple. For example, *Under-Restrictive Policy, Validate, Source Code Injection)* is a meaningful weakness triple, but *Corrupted Data, Validate, Source Code Injection)* – is not.

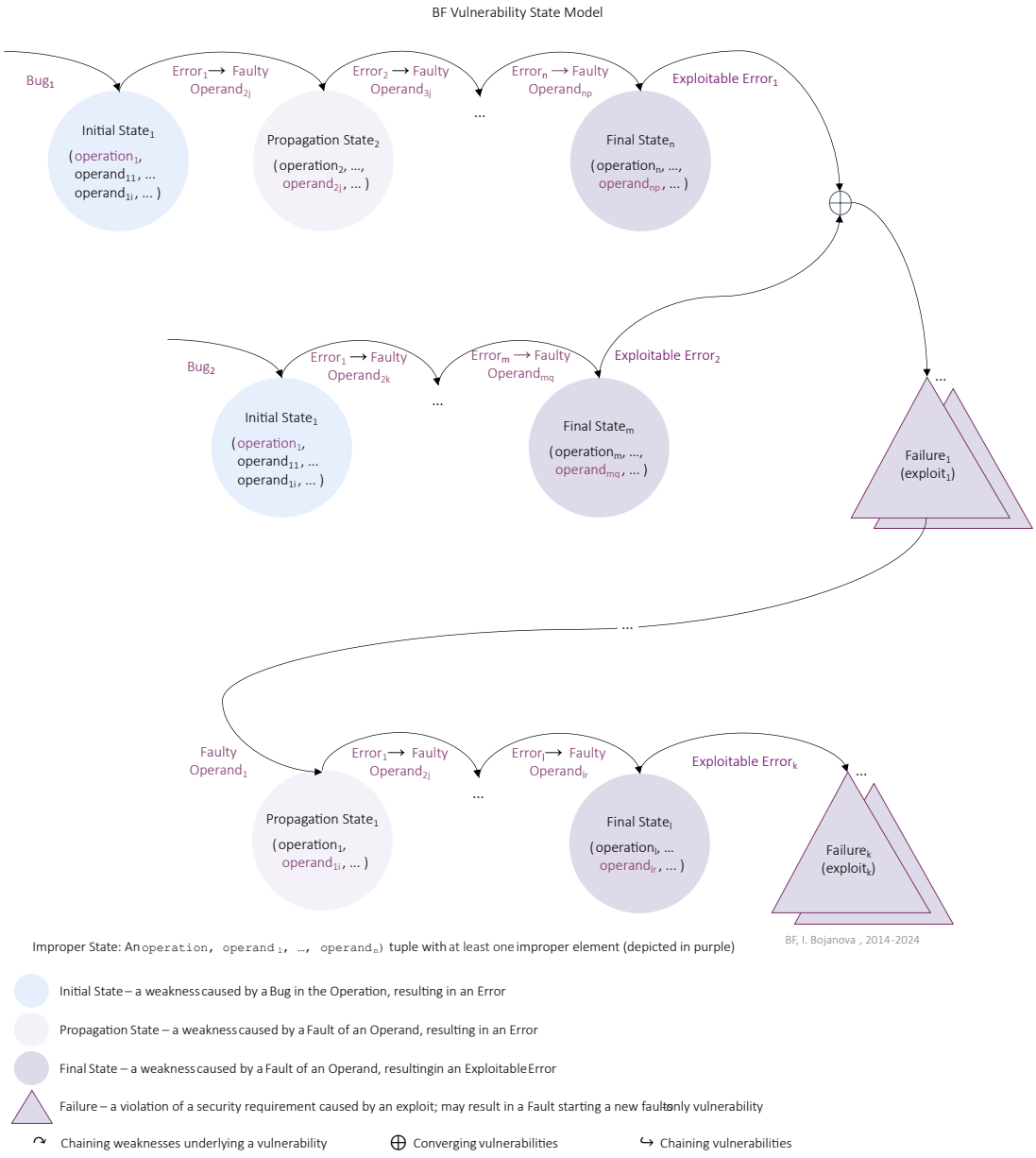



**Fig. 13.** BF Vulnerability State Model. A chain of improper states that starts with a bug, propagates via errors becoming faults, and leads to a failure. Occasionally, several vulnerabilities must converge at their final errors for an exploit to be harmful. May propagate to other faults-only-vulnerabilities via faults resulting from exploits.

Causation between weaknesses is by valid operation flow – a graphs of meaningful *(operation*$_1$, ..., *operation*$_n$*)* weakness state paths. Propagation between weaknesses is by valid *error→fault* transitions – the error resulting from the operation of a weakness becomes the fault of another weakness. The matching is by fault type and fault value for weaknesses of the same BF class type; or only by fault type and valid *consequence→cause*. For example,

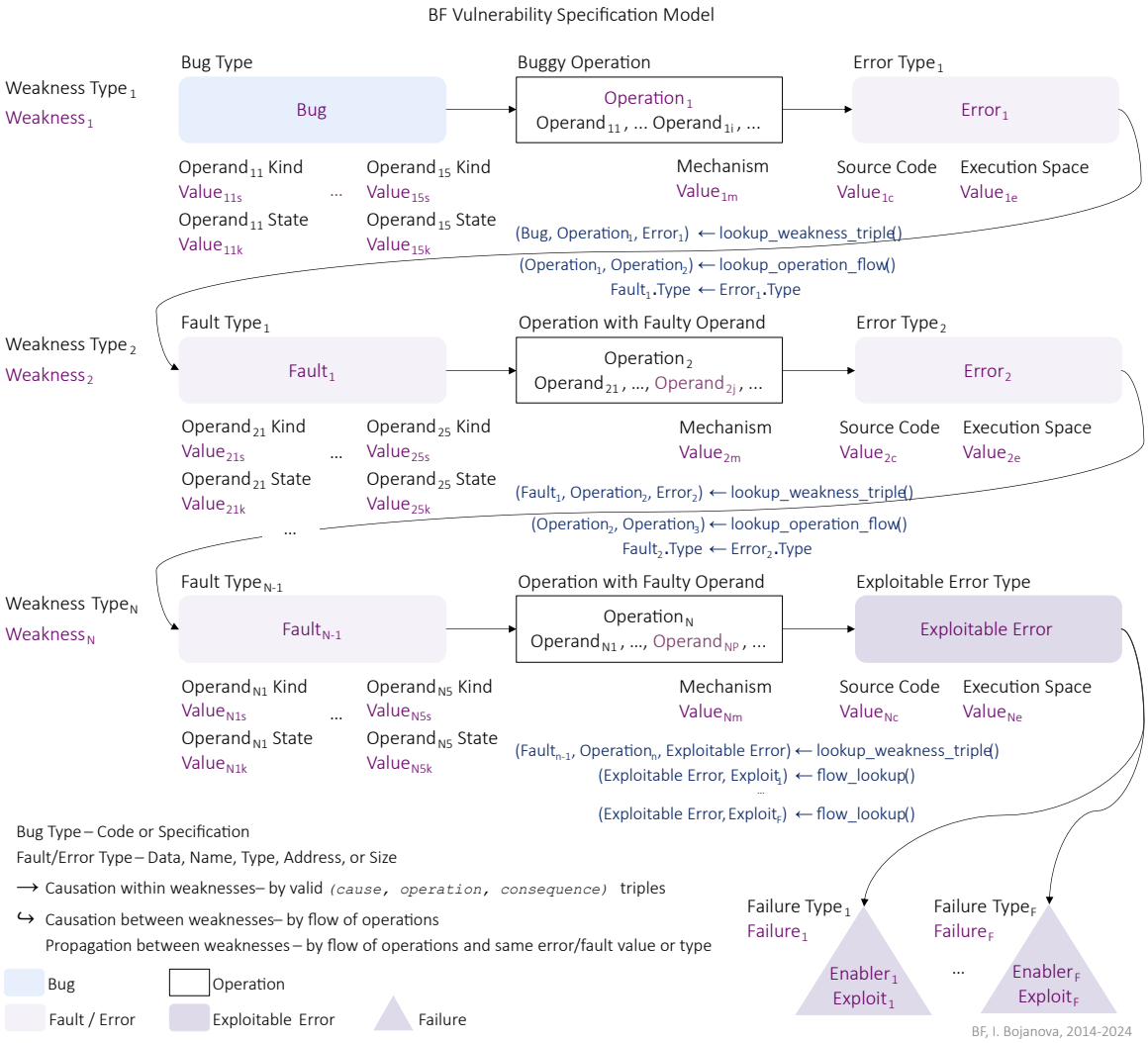



**Fig. 14.** BF Vulnerability Specification Model. Reflects the BF concepts definitions, the BF taxonomies within weakness causation rules, the BF bugs models between weaknesses causation rules, and the BF state model propagation rules.

*(Wrong Type, Coerce, Flipped Sign)* may propagate to *(Wrong Argument, Evaluate, Under Range)* as Evaluate may follow Coerce (see Fig. 7), the within weakness causations and *Flipped Sign→Wrong Argument* transition are valid.

Causation between vulnerabilities is by a valid *exploit→operation* transitions – the result from an exploit becomes the fault for a new faults-only vulnerability. Propagation between vulnerabilities is by *exploit result* matching to the start *fault* type. For example, information exposure of private keys may become the fault starting a new vulnerability.

For simplicity Fig. 14 does not show vulnerability convergence and chaining, as they correspond directly to the transitions in the Vulnerability State Model (see 13).

## 7. BF Formal Language

The BF formal language is generated by the BF Left-to-right Leftmost-derivation One-symbol-lookahead (LL(1)) attribute context-free grammar (ACFG) derived from the BF CFG. As based in an LL(1) grammar, BF is guaranteed to be unambiguous – the BF weakness and vulnerability specifications are guaranteed to be clear and precise. *Clear* means easy to understand, straightforward, and unambiguous – there is no room for confusion or misinterpretation. *Precise* means exact, accurate, and specific, which also implies unambiguous. The BF lexis, syntax, and semantics are based on the BF Taxonomies (see for example Fig. 11), bugs models (see, for example, Fig. 6), and vulnerability models (see Fig. 13 and 14). *Lexis* refers to the vocabulary (words and symbols) used by a specification language. *Syntax* is about validating the grammatical structure (the form) of a specification. *Semantics* is about verifying the logical structure (the meaning) of a specification.

The *BF context-free grammar (CFG)* is a powerful tool for specifying and analysing security weaknesses and vulnerabilities. It is defined as a four-tuple

$$G = (V, \Sigma, R, S) \tag{1}$$

, where:

- $\Sigma$ defines the BF lexis (the alphabet of the CFG) as a finite set of tokens (terminals) comprised by the sets of BF taxons and BF symbols (see Listing 3)

$$\Sigma = \{\alpha \mid \alpha \in \Sigma BFtaxons \cup \Sigma BFsymbols\}$$

- $V$ and $R$ define the BF syntax as
  - a finite set of variables (nonterminals)

$$V = \{S, V_1, \ldots, V_n\}$$

    and
  - a finite set of syntactic rules (productions) in the form

$$R = \{A \longmapsto \omega \mid A \in V \ \wedge \ \omega \in (V \cup \Sigma)^*\}$$

    , where:

    $(V \cup \Sigma)^*$ is a string of tokens and/or variables

    $A \longmapsto \omega$ means any variable $A$ occurrence may be replaced by $\omega$.
- $S \in V$ is the predefined start variable from which all BF specifications derive.

A BF *specification* starts from $S$ and ends with the empty string, denoted with the null symbol $\varepsilon$. The *derivation* is via a sequence of steps where nonterminals are replaced by the right-hand side of a production. The production rules are applied to a variable regardless of its context.

The *BF formal language* is generated by the BF LL(1) ACFG $G = (V, \Sigma, R, S)$ (see Listing 7) that augments with semantic rules the syntax defined by the BF CFG (see Listings 4, 5, and 6).

The formal language is defined as the set of all strings of tokens $\omega$ derivable from the start variable $S$.

$$L(G) = \{\omega \in \Sigma^* : S \overset{*}{\Longrightarrow} \omega\} \tag{2}$$

, where:

- $\Sigma^*$ is the set of all possible strings that can be generated from $\Sigma$ tokens
- $S$ is the start variable
- $\alpha \overset{*}{\Longrightarrow} \beta$ means string $\alpha$ derives string $\beta$

Note that $\omega$ must be in $\Sigma^*$, the set of strings made from terminals. Strings involving non-terminals are not part of the language.

### 7.1. BF Lexis

The BF lexis refers to the vocabulary (words and symbols) of the BF formal language – the set of tokens $\Sigma$. It is defined by the set of BF taxons for category, operation, and class types and classes of bug, fault, final error, operation attribute, operand attribute, and the set of BF symbols for describing causation within a weakness ($\rightarrow$), causation between weaknesses and vulnerabilities ($\hookrightarrow$), and convergence of vulnerabilities ($\oplus$). See Listing 3 – the taxons are in quotes (e.g., *'Missing Code'* or *'Query Injection'*) and considered literal word; for simplicity the symbols are not in quotes.

$$\Sigma = \left\{\Sigma BFtaxons, \Sigma BFsymbols\right\} \tag{3}$$

, where

$$\begin{aligned}\Sigma BFtaxons = \{&\Sigma Category, \Sigma ClassType, \Sigma Class, \Sigma BugType, \Sigma Bug,\\ &\Sigma Operation, \Sigma OperationAttributeType,\\ &\Sigma FaultType, \Sigma Fault, \Sigma OperandAttributeType, \Sigma OperandAttribute,\\ &\Sigma FinalErrorType, \Sigma FinalError\}\\ \Sigma BFsymbols = \{&\rightarrow, \hookrightarrow, \oplus\}\end{aligned}$$

, where BF classes are of a weakness or a failure category and within each category of a class type, bugs could be of code defect or of specification defect type, faults could be of data, (data) type, name, address, or size type, etc. For example:

$$\Sigma Category = \{'Weakness','Failure'\}$$
$$\Sigma ClassType = \{'\_INP','\_DAT','\_MEM',...\}$$
$$\Sigma Class = \{'DVL','DVR','DCL','NRS','TCV','TCM','MAD','MMN','MUS',...\}$$

$$\begin{aligned}\Sigma Operation = \{&'Validate','Sanitize','Verify','Correct','Declare','Define','Refer',\\ &'Call','Cast','Coerce','Calculate','Evaluate','InitializePointer',\\ &'Reposition','Reassign','Allocate','Extend','Reallocate-Extend',\\ &'Deallocate','Reduce','Reallocate-Reduce','InitializeObject',\\ &'Dereference','Read','Write','Clear','Generate/Select','Store',\\ &'Distribute','Use'...\}\end{aligned}$$

$$\begin{aligned}\Sigma BugType &= \{'CodeDefect','SpecificationDefect'\}\\ \Sigma Bug &= \{'MissingCode','ErroneousCode','Under-RestrictivePolicy',\\ &'Over-RestrictivePolicy','WrongCode','MissingModifier',\\ &'WrongModifier','AnonymousScope','WrongScope',\\ &'MissingQualifier','WrongQualifier','MismatchedOperation',...\}\end{aligned}$$

$$\begin{aligned}\Sigma FinalErrorType &= \{'Injection','Access','TypeCompute',\\ &'MemoryCorruption/Disclosure',...\}\\ \Sigma FinalError &= \{'QueryInjection','CommandInjection','SourceCodeInjection',\\ &'ParameterInjection','FileInjection','WrongAccessObject',\\ &'WrongAccessType','WrongAccessFunction','Undefined',\\ &'MemoryLeak','MemoryOverflow','DoubleDeallocate',\\ &'ObjectCorruption','NotClearedObject',\\ &'NULLPointerDereference','UntrustedPointerDereference',\\ &'TypeConfusion','UseAfterDeallocate','BufferOverflow',\\ &'BufferUnderflow','BufferOver-Read','BufferUnder-Read',...\}\end{aligned}$$

For complete BF formal grammar lexis and taxons definitions refer NIST SP xxx-xxxB BF Formal Language.

### 7.2. BF Syntax

The BF formal language syntax is about validating the grammatical structure (the form) of a BF specification. It is defined by BF production rules (non-terminals) for constructing/producing valid specifications of the language – adhering to the BF vulnerability specification model structure and flow (see Fig. 14) (including the BF state model converging and chaining – see Fig. 13). The CFG production rules are expressed via the Extended Backus–Naur Form (EBNF) using the meta-notations of:

- ::= for 'is defined as',
- $+$ for '1 or more occurrences'
- ? for '0 or 1 occurrences'
- () for grouping.

A simplified BF CFG EBNF (see Listing 4) defines a chain of one of more vulnerabilities, possibly converging with other vulnerabilities, each leading to one or more failures.

$$S ::= \big(Vulnerability\ (\oplus\ Vulnerability)?\ Failure+\big) + \varepsilon \tag{4}$$

$$\begin{aligned} Vulnerability &::= +Weakness \\ Weakness &::= Cause\ Operation\ Consequence \\ Cause &::= Bug \mid Fault \\ Consequence &::= Error \mid FinalError \end{aligned}$$

A vulnerability is defined as a chain of weaknesses, each defined as a *(cause, operation, consequence)* triple. A cause is defined as a bug or a faults. A consequence is defined as an error or a final error. (see Listing 4)

However, according to the BF vulnerability specification model (see Fig. 14) only the cause of the first weakness can be a bug and only the last consequence can be a final error. A vulnerability with a single weakness, is the only case when a weakness is defined with both a bug cause and a final error consequence. An intermediate weakness is caused by a fault and results in an error. Reflecting these rules into the productions of Listing 4, while also eliminating the *Cause* and *Consequence* variables, the BF CFG syntax productions are as follows.

$$S ::= \big(Vulnerability\ (\oplus Vulnerability)?\ Failure+\big) + \varepsilon \tag{5}$$

$$
\begin{aligned}
Vulnerability &::= SingleWeakness \\
&\quad \mid FirstWeakness\ \big(Weakness+\big)\ LastWeakness \\
SingleWeakness &::= Bug\ Operation\ FinalError \\
FirstWeakness &::= Bug\ Operation\ (Error \mid FinalError) \\
Weakness &::= Fault\ Operation\ Error \\
LastWeakness &::= Error\ Operation\ FinalError
\end{aligned}
$$

To assure unambiguous BF specifications, the next step is to successfully derive a BF LL(1) formal grammar from the BF CFG. A CFG is an *LL(1) grammar* if and only if only one token (terminal) or variable (non-terminal) is needed to make a parsing decision. LL(1) grammars are *not ambiguous* and *not left-recursive*.

The BF CFG four tuple $G = (V, \Sigma, R, S)$ would be an LL(1) grammar
$\iff \forall S \longmapsto A \mid B$, where $A, B \in V$ :

- $\forall A \longmapsto \alpha \mid \beta$, where $A \in V \land \alpha, \beta \in (V \cup \Sigma)^* \land \alpha \neq \beta$ :
  $First(\alpha) \neq First(\beta) \land Lookahead(A \longmapsto \alpha) \cap Lookahead(A \mapsto \beta) = \emptyset$
- if $\alpha \overset{*}{\Longrightarrow} \varepsilon$ then $First(\beta) \cap Follow(A) = \emptyset$

, where:

$\iff$ means 'if and only if'

$x \Longrightarrow y$ means $y$ can be derived from $x$ in exactly one application of some production of the grammar

$x \overset{*}{\Longrightarrow} y$ means that $y$ is derived from $x$ via zero or more (but finitely many!) applications of some sequence of productions – t.e., there is some series of applications of rules that goes from $x$ to $y$.

The *BF LL(1) formal CFG* is derived from the BF EBNF productions on Listing 5 via left-factorization and left recursion elimination. It is suitable for recursive descent parsing, as each production option start is unique and, on each step, which rule must be chosen is uniquely determined by the current variable and the next taxon (if there is one).

$$S ::= Vulnerability\ Converge\_Failure \tag{6}$$

$$
\begin{aligned}
\mathit{Vulnerability} &::= \mathit{Bug\ Operation\ OperAttrs\_Error\_FinalError} \\
\mathit{OperAttrs\_Error\_FinalError} &::= \mathit{Oper\_Attr\ OperAttrs\_Error\_FinalError} \\
&\mid \mathit{Error\ Fault\ OprndAttrs\_Operation} \\
&\mid \mathit{FinalError} \\
\mathit{OprndAttrs\_Operation} &::= \mathit{OperandAttribute\ OprndAttrs\_Operation} \\
&\mid \mathit{Operation\ OperAttrs\_Error\_FinalError} \\
\mathit{Converge\_Failure} &::= \oplus\ \mathit{Vulnerability\ Converge\_Failure} \\
&\mid \mathit{Enabler\ Exploit\ ExploitResult\ NextVulner\_Failure} \\
\mathit{NextVulner\_Failure} &::= \mathit{Fault\ Operation\ OperAttrs\_Error\_FinalError} \\
&\mid \mathit{Enabler\ Exploit\ Failure}\ \varepsilon
\end{aligned}
$$

The BF LL(1) formal grammar is a powerful tool for describing and analyzing the BF formal language. It defines the set of rules by which valid unambiguous BF specifications are constructed.

The BF specifications are derived from S by step-by-step production application, substituting for the one leftmost nonterminal at a time until the string is fully expanded, i.e., consist of only terminals. An important effect of being based on an LL(1) grammar implies BF is unambiguous!

For the complete BF LL(1) formal CFG (with all production rules defined) refer NIST SP xxx-xxxE BF Formal Language.

### 7.3. BF Semantics

The BF formal language semantics is about verifying the logical structure (the meaning) of a BF specification. It is defined by extending the BF LL(1) CFG to a BF LL(1) ACFG with static semantic rules – adhering to the BF vulnerability specification model causation and propagation rules (see Fig. 14) (including the BF state model converging and chaining – see Fig. 13).

The BF LL(1) ACFG syntax rules with subscripted nonterminals (if they appear more than once) and the semantic rules to check for valid weakness triples, valid flow of operations, error-fault by value matching for same BF class types; and the predicates for matching by *Type* are defined as follows.

(7)

$$
\begin{aligned}
&\textit{SyntaxRules}: \\
&S ::= \textit{Vulnerability}\ \textit{Converge\_Failure} \\
&\textit{Vulnerability} ::= \textit{Bug}\ \textit{Operation}_1\ \textit{OperAttrs\_Error\_FinalError} \\
&\textit{OperAttrs\_Error\_FinalError} ::= \textit{Oper\_Attr}\ \textit{OperAttrs\_Error\_FinalError} \\
&\quad \mid \textit{Error}\ \textit{Fault}_1\ \textit{OprndAttrs\_Operation} \\
&\quad \mid \textit{FinalError} \\
&\textit{OprndAttrs\_Operation} ::= \textit{OperandAttribute}\ \textit{OprndAttrs\_Operation} \\
&\quad \mid \textit{Operation}_k\ \textit{OperAttrs\_Error\_FinalError} \\
&\textit{Converge\_Failure} ::= \oplus\ \textit{Vulnerability}\ \textit{Converge\_Failure} \\
&\quad \mid \textit{Enabler}\ \textit{Exploit}\ \textit{ExploitResult}\ \textit{NextVulner\_Failure} \\
&\textit{NextVulner\_Failure} ::= \textit{Fault}_2\ \textit{Operation}\ \textit{OperAttrs\_Error\_FinalError} \\
&\quad \mid \textit{Enabler}\ \textit{Exploit}\ \ \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
&\textit{SemanticRules}: \\
&\big(\textit{Bug},\ \textit{Operation}_1,\ \textit{Error}\big) \leftarrow \textit{lookup\_weakness\_triple}() \\
&\big(\textit{Bug},\ \textit{Operation}_1,\ \textit{FinalError}\big) \leftarrow \textit{lookup\_weakness\_triple}() \\
&\big(\textit{Fault}_1,\ \textit{Operation}_k,\ \textit{Error}\big),\ k>1 \leftarrow \textit{lookup\_weakness\_triple}() \\
&\big(\textit{Fault}_1,\ \textit{Operation}_k,\ \textit{FinalError}\big),\ k>1 \leftarrow \textit{lookup\_weakness\_triple}() \\
&(\textit{Operation}_1, ...\textit{Operation}_k),\ k>1 \leftarrow \textit{lookup\_operation\_flow}() \\
&\textit{Fault}_1 \leftarrow \textit{if}\ (\textit{Fault}_1.\textit{ClassType} == \textit{Error.ClassType})\ \textit{then}\ \textit{Error} \\
&\textit{Predicates}: \\
&\textit{Fault}_1.\textit{Type} == \textit{Error.Type} \\
&\textit{Enabler.Type} == \textit{FinalError.Type} \\
&\textit{Fault}_2.\textit{Type} == \textit{ExploitResult.Type}
\end{aligned}
$$

The static semantic rules (see Listing 7) are expressed via a set of grammar attributes (properties to which values can be assigned), a set of semantic functions (for computing the attribute values), and a possibly empty set of predicate functions for each production rule (as in Donald Knuth's attribute grammars [28]).

The BF LL(1) ACFG adds the *Type* synthesized attribute for the nonterminals *Error*, *Fault*, and *ExploitResult* to store the operands types (*Name*, *Data*, *Type*, *Address*, *Size*); and *FinalError* and *Enabler* to store the final exploitable error types (e.g., *Injection*, *Access*, *Type Compute*, *Memory Corruption/Disclosure*, ...).

For the complete BF LL(1) formal CFG NIST SP xxx-xxxE BF Formal Language.

## 8. BF Application

BF futures generation tools reflecting the BF approach, taxonomy, and formal language syntax and semantics. The APIs provide BF related data retrieval and specific tool functionalities. BF is applicable for systematic comprehensive labeling of common weakness types and disclosed vulnerabilities, and based on that generation of weakness and vulnerability classifications.

### 8.1. BF Databases

The BFDB database hosts the BF data. The BF taxonomy and the BF formal language rules are defined via a relational (see Fig. 15) and graph databases, and data interchange formats such as XML and JSON (query it via the BF API).

The BFDB relational database defines and organizes the BF taxonomy structure. It contains the types, names, and definitions of the BF taxons, and their relationships within the taxonomy, as well as the BF causation and propagation rules.

The VulDB mashup database defines and organizes additional data for querying BF towards CWE, CVE, NVD, and GitHub [29], KEV [4], Software Assurance Reference Dataset (SARD) [30], and Exploit Prediction Scoring System (EPSS) [31] repositories.

For details on the BF databases refer NIST SP xxx-xxxF BF Databases, Tools, and APIs.

### 8.2. BF Tools

The BFCWE tool and the BFCVE Tool [32] tool facilitate generation of formal weakness and vulnerability specifications. The BF Tool guides the creation of complete BF vulnerability specifications.

#### 8.2.1. BFCWE Tool

The BFCWE Tool [33] facilitates the creation of CWE-to-BF (CWE2BF) mappings by weakness operation, error, and final exploitable error, and possibly by entire main *(cause, operation, consequence)* BF weakness triple and generates BFCWE formal specifications and graphical representations of the mappings and the specifications for enhanced understanding.

Basic security weaknesses research and meticulous analysis of the natural language descriptions of all data type, input/output check, memory related, etc. CWEs (as well as of relevant code examples and CVEs) is conducted to create CWE2BF mappings by weakness operation, error, final error and then by detailed BF *(bug, operation, error)*, *(fault, operation, error)*, or *(fault, operation, exploitable error)* weakness triples (see [19], [18], [17], and [25]).

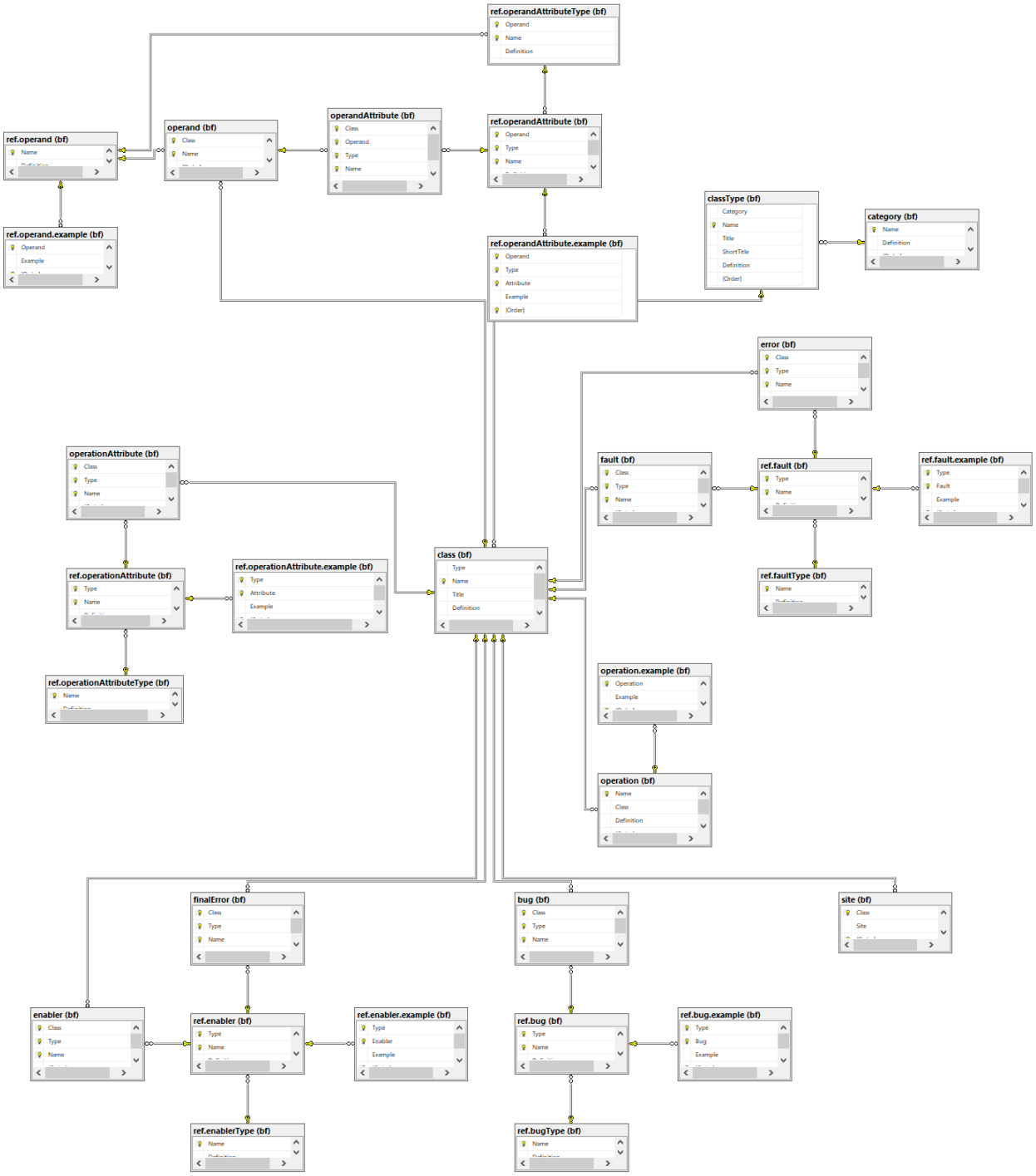


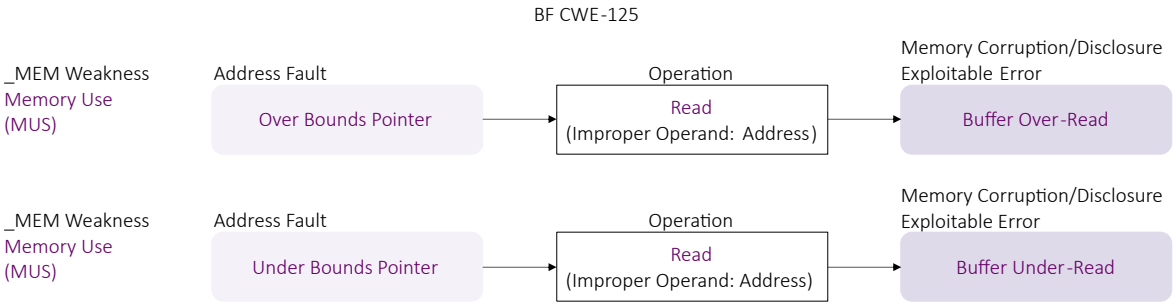

**Fig. 15.** Diagram of the BFDB relational database.

The BFCWE tool generates BFCWE formal specifications as entries of the BFCWE software security weakness types dataset. It also generates di-graphs for enhanced understand-

ing of the CWE2BF mappings (by operation, error, final error, and complete weakness triples) with parent-child CWE relations, and of the BFCWE formal specifications.

For example, analysis of the natural language descriptions, examples, and mitigation techniques for CWE-125 reveals its possible main BF weakness triples are *(Over Bounds Pointer, Read, Buffer Over-Read)* and *(Under Bounds Pointer, Read, Buffer Over-Read)*. Figure 16 illustrates the generated by the BFCWE tool graphical representations of their BF specifications, with all the possible combinations to consider as the main weakness for CVEs mapped to CWE-125.

Although, a CWE should be about a single weakness, the descriptions of some CWEs also reveal possible causing chains of weakness triples (see [25] for _MEM).




| **Class Type** | **Definition** |
|---|---|
| *Memory Corruption/Disclosure (_MEM)* | *Bugs/weaknesses allowing a memory corruption/disclosure exploit.* |
| **Class** | **Definition** |
| *Memory Use (MUS)* | *An object is initialized, read, written, or cleared improperly.* |
| **Operation** | **Definition** |
| *Read* | *Use the value of an object's data.* |
| **Cause** | **Definition** |
| *Address Fault* | *The object address in use is wrong.* |
| *Over Bounds Pointer* | *Holds an address above the upper boundary of its object.* |
| *Under Bounds Pointer* | *Holds an address above thelower boundary of its object.* |
| **Consequence** | **Definition** |
| *Memory Corruption/Disclosure Final Error* | *An exploitable or undefined system behavior caused by memory addressing, allocation, use, and deallocation bugs.* |
| *Buffer Over-Read* | *Reading above the upper bound of an object.* |
| *Buffer Under-Read* | *Reading below the lower bound of an object.* |

**Fig. 16.** Graphical representation of the BF Memory Use (MUS) specifications of CWE-125.

All identified weakness triples are checked towards the BF Causation Matrix of meaningful *(cause, operation, consequence)* triples, which defines part of the BF LL(1) Formal Language semantics.

This same methodology helps reveal CWEs overlaps, as many CWEs have the same BF specifications. Although, a CWE should be about a single weakness, the descriptions of some CWEs also reveal possible causing chains of weakness triples [25].

For details on the BFCWE tool refer NIST SP xxx-xxxF BF Databases, Tools, and APIs.

### 8.2.2. BFCVE Tool

The BFCVE Tool [32] generates possible chains of weaknesses for a vulnerability by identified failure and final exploitable error or possibly entire final exploitable weakness, generates possible BFCVE formal specifications and their graphical representations, and identifies and recommends a CWE(s) for NVD assignment. Code analysis and the BF GUI (Graphical User Interface) functionality can be used to identify and complete the unique unambiguous BF vulnerability specifications.

The BF relational database, the NVD Representational State Transfer Application Programming Interface (REST API), and the GitHub REST API are utilized to extract CVEs with assigned CWEs for which *Code with Fix* is available. For example, as of March 2024 there are more than 600 CVEs that map to BF Data Type (_DAT [19]) weakness triples by CWE, more than 4600 – to BF Input/Output Check (_INP [18]) weakness triples, and more than 3970 CVEs – to BF Memory Corruption/Disclosure (_MEM [17]) weakness triples for which GitHub diffs are available through NVD. Other repositories also may provide commits and even code of vulnerable functions – e.g., there are 269 CVEs in DiverseVul [34] for which final weaknesses map to BF Memory Corruption/Disclosure (_MEM [17]) weakness triples, and the *Code with fix* can be extracted from the fix commits via the GitHub REST API.

Next, information on the failure(s) and the final exploitable weakness is gained from the CVE report(s), the CVE description, and the CWE2BF weakness triple mappings if a CWE(s) is assigned by NVD. The BFCVE tool utilizes the BF relational database and the NVD REST API to extract the CWE2BF triples for that CVE. Then, the BFCVE tool applies the BF causation and propagation rules (i.e., the BF formal language syntax and semantics) to go backwards from the failure(s) through the final exploitable weakness to generate all possible BF chains of weaknesses for that specific CVE, independently of whether the CVE *Code with Fix* is available.

Going backwards from the failure, the BFCVE tool builds a connected acyclic undirected graph (a tree, which root is the failure) of all possible weakness chains with type-based *fault-to-error* backward propagation, plus for weaknesses of same BF class type – with name-based backward propagation. Then the chains undergo scrutiny to ensure further alignment with the BF Formal Language semantics – the Causation Matrix of all meaningful *(cause, operation, consequence)* weakness triples and the Propagation Graphs of meaningful *(operation$_1$, ..., operation$_n$)* bug or fault state paths and Matrix of all valid *consequence*→*cause* transitions between weaknesses.

Identified beforehand failure(s) and final exploitable weakness triple(s) reduce dramatically the number of generated possible paths in the acyclic graph. This is also a good starting point for specifying vulnerabilities not recorded in CVE, as far as failure(s) and final ex-

ploitable weakness information are identifiable.

The CVE *Code with Fix* can then be examined by security researchers or utilizing AI towards the generated chains of weakness triples to pinpoint the unique unambiguous BF vulnerability specification. For that both the BF tool functionality and automated code analysis and Large Language Models (LLMs) can be utilized.

For example, main vulnerability for CVE-2014-0160 Heartbleed is mapped in NVD to CWE-125 [20]. The CWE2BF mappings for CWE-125 restricts to two the final exploitable weakness options for Heartbleed: *(Over Bounds Pointer, Read, Buffer Over-Read)* or *(Under Bounds Pointer, Read, Buffer Under-Read)*. However, the CVE-2014-0160 description reveals the word *over*, which points CWE-125 is too abstract for it and eliminates the second BF final exploitable error option. In addition, as Heartbleed leads to information exposure, the last part of the BF weaknesses chain specification is: *(Over Bounds Pointer, Read, Buffer Over-Read)→Information Exposure (IEX)*. Note that the *Read* operation uniquely identifies the BF MUS class, as BF classes do not overlap by operation [17].

Going backwards from *Over Bounds Pointer* using the BF causation ad propagation rules, the BFCVE tool generates the tree of suggested weakness chains for Heartbleed. As shown on Fig. 17, the failure is the root, the final exploitable error is the first node, and a bug is the last node in each path. The only options for the weakness causing the final exploitable weakness are: *(Wrong Index, Reposition, Over Bounds Pointer)* and *(Wrong Size, Reposition, Over Bounds Pointer)*. Both of them have the same options for causing chains, only two of which do not start with a bug, but even for them the preceding weakness options start with a bug. Exhausting these few options via source code analysis or use of LLMs, should be straight forward to confirm the unique unambiguous chain for Heartbleed is: *(Missing Code, Verify, Inconsistent Value)→(Wrong Size, Reposition, Over Bounds Pointer)→(Over Bounds Pointer, Read, Buffer Over-Read)→Information Exposure (IEX)*.

```
Information Exposure (IEX)
  (Over Bounds Pointer, Read, Buffer Over-Read)
    (Wrong Index/Wrong Size, Reposition, Over Bounds Pointer)
      (Missing Code/Erroneous Code/Under-Restrictive Policy/Over-Restrictive Policy, Validate/Sanitize, Invalid Data)
      (Missing Code/Erroneous Code/Under-Restrictive Policy/Over-Restrictive Policy, Verify/Check, Wrong Value/Inconsistent Value)
      (Erroneous Code, Calculate, Wrap Around)
      (Erroneous Code, Calculate/Evaluate, Wrong Result)
      (Wrong Type, Calculate/Evaluate, Wrong Result)
        (Missing Code/Erroneous Code/Under-Restrictive Policy/  Over-Restrictive Policy, Verify, Wrong Type)
        (Erroneous Code, Define, Wrong Type)
        (Wrong Object Type Resolved, Coerce, Wrong Type)
          (Missing Qualifier/Wrong Qualifier, Refer, Wrong Object Type Resolved)
```

**Fig. 17.** BFCVE tool generated tree of possible chains for the main CVE-2014-0160 (Heartbleed) vulnerability using the BF methodology for backwards Bug identification from a Failure.

Finally, the BFCVE tool can generate graphical representation(s) of the BFCVE formal specifications for enhanced understanding. For example, Fig. 20 illustrates the generated by the BFCVE tool graphical representation of the BF Hearbleed specification and related BF taxons definitions. For simplicity, only part of the definitions table is visualized.

For details on the BFCVE tool refer NIST SP xxx-xxxF BF Databases, Tools, and APIs.

### 8.2.3. BF GUI Tool

The BF Tool [26] is a Graphical User Interface (GUI) application (see Fig. 18), which works both with the BF relational database and the BF in XML or JSON format (useful especially when connectivity to the databases is not available). It has a rich Graphical User Interface (GUI), allowing the user to create a new BF CVE specification, save it as machine-readable *.bfcve* file (see Fig. 19), and open and browse previously created *.bfcve* specifications.

The BF tool guides the specification of a software security vulnerability as a chain of underlying weaknesses. A software security bug causes the first weakness, leading to an error. This error becomes the cause (i.e., the fault) for a next weakness and propagates through subsequent weaknesses until a final exploitable error is reached, causing a security failure. The causation within a weakness is by a meaningful *(cause, operation, consequence)* triple. The causation and propagation between weaknesses are by a meaningful *consequence-¿cause* transition, and by flow of operations and by same error/fault type, correspondingly.

If an existing CVE is being specified, the user can select *CVE Year* and *CVE ID* in the *CVE Details* GroupBox to see its description, vendor, and product from the CVE repository, and its CVSS score from NVD. To create a BFCVE specification of that CVE, the user is guided to define an initial weakness, possible propagation weaknesses, and a final weakness leading to a failure. In the case of a vulnerability with only one underlying weakness, that would be both an initial and final weakness.

To start defining a weakness, the user has to select a BF Class from the *BF Class* TreeView in the *Weakness* GroupBox container, where the classes are grouped by BF class types as parent nodes. The selection of a class, populates the five TreeView controls in the *Weakness* GroupBox container: *Bug/Fault*, *Operation*, *Error/Final Exploitable Error*, *Operation Attributes*, and *Operand Attributes*. To specify the weakness the user has to select child nodes from the five TreeView controls and enter comments in the text-boxes beneath them.

The BF tool enforces the initial weakness to start with a Bug, the rest of the weaknesses to start with a Fault. The *Bug/Fault* Label changes to *Bug* when the initial weakness is viewed and to *Fault* when propagation or final weakness is viewed. In the case of a Bug, the child nodes are allowed only under the *Code* and the *Specification* nodes. In the case of a Fault, the child nodes are allowed only under the *Data*, *Type*, *Address*, and *Size* nodes. Tooltips with term definitions are displayed over all TreeView nodes. The BF tool also enforces that the weakness with the *Final Exploitable Error* consequence is the final weakness, leading to a failure.

Once a weakness is specified, the user can use the >> Button to proceed and create the next weakness from the vulnerability chain. The weakness chaining is restricted by the

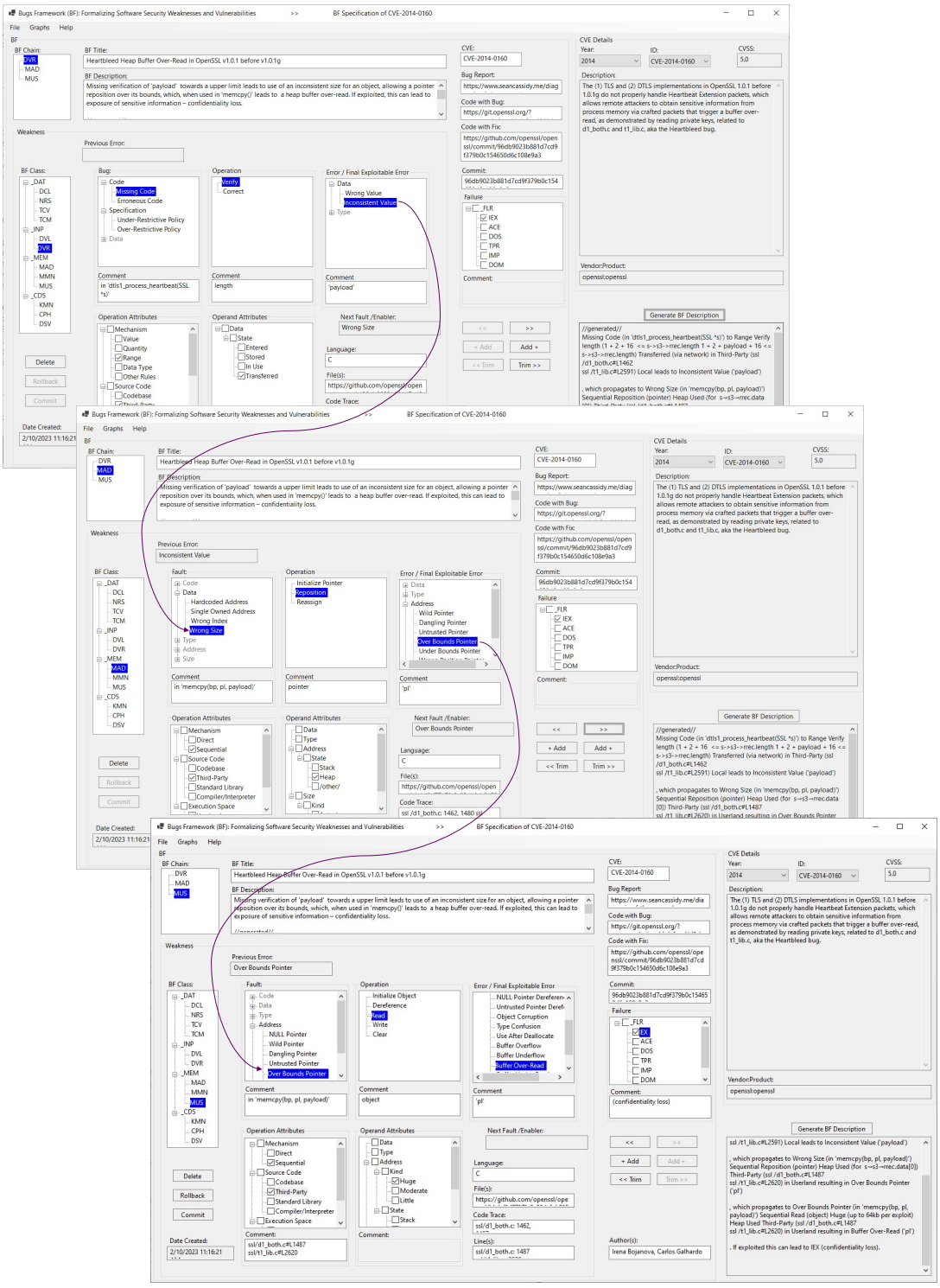


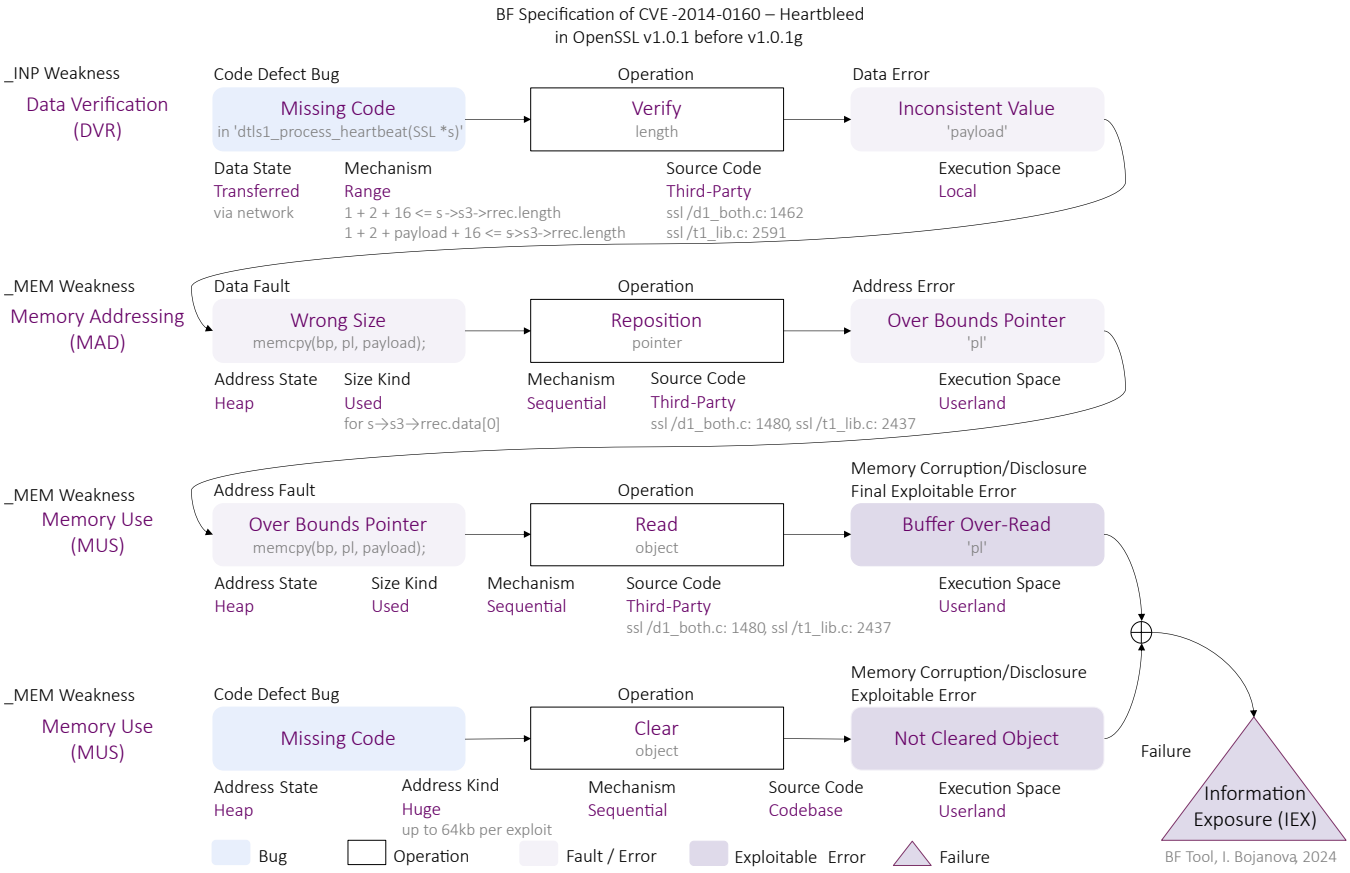

**Fig. 18.** BF GUI tool – utilizes the BF taxonomy and enforces the BF formal language syntax and semantics. Screenshots show the comprehensive BF labels for CVE-2014-0160 Heartbleed.

```
<?xml version="1.0" encoding="utf-8"?>
<!--Bugs Framework (BF), BFCVE Tool, I. Bojanova, NIST, 2020-2024-->
<BFCVE ID="CVE-2014-0160" Title="Heartbleed Heap Buffer Over-Read in OpenSSL v1.0.1 before v1.0.1g" Description="Missing
  <DefectWeakness Class="DVR" ClassType="_INP" Language="C" File="https://github.com/openssl/openssl/commit/96db9023b881
    <Cause Comment="in 'dtls1_process_heartbeat(SSL *s)'" Type="Code">Missing Code</Cause>
    <Operation Comment="length">Verify</Operation>
    <Consequence Comment="'payload'" Type="Data">Inconsistent Value</Consequence>
    <Attributes>
      <Operand Name="Data">
        <Attribute Comment="via network" Type="State">Transferred</Attribute>
      </Operand>
      <Operation>
        <Attribute Comment="1 + 2 + 16  &lt;= s->s3->rrec.length 1 + 2 + payload + 16 &lt;= s->s3->rrec.leng
        <Attribute Comment="ssl/d1_both.c#L1462&#xD;&#xA;ssl/t1_lib.c#L2591" Type="Source Code">Third-Party</Attribute>
        <Attribute Type="Execution Space">Local</Attribute>
      </Operation>
    </Attributes>
  </DefectWeakness>
  <Weakness Class="MAD" ClassType="_MEM" Language="C" File="https://github.com/openssl/openssl/blob/0d7717fc9c83dafab815
    <Cause Comment="in 'memcpy(bp, pl, payload)'" Type="Data">Wrong Size</Cause>
    <Operation Comment="pointer">Reposition</Operation>
    <Consequence Comment="'pl'" Type="Address">Over Bounds Pointer</Consequence>
    <Attributes>
      <Operand Name="Address">
        <Attribute Type="State">Heap</Attribute>
      </Operand>
      <Operand Name="Size">
        <Attribute Comment="for  s->s3->rrec.data[0]" Type="Kind">Used</Attribute>
      </Operand>
      <Operation>
        <Attribute Type="Mechanism">Sequential</Attribute>
        <Attribute Comment="ssl/d1_both.c#L1487&#xD;&#xA;ssl/t1_lib.c#L2620" Type="Source Code">Third-Party</Attribute>
        <Attribute Type="Execution Space">Userland</Attribute>
      </Operation>
    </Attributes>
  </Weakness>
  <Weakness Class="MUS" ClassType="_MEM" Language="C" File="https://github.com/openssl/openssl/blob/0d7717fc9c83dafab815
    <Cause Comment="in 'memcpy(bp, pl, payload)'" Type="Address">Over Bounds Pointer</Cause>
    <Operation Comment="object">Read</Operation>
    <Consequence Comment="'pl'" Type="Memory Corruption/Disclosure">Buffer Over-Read</Consequence>
    <Attributes>
      <Operand Name="Address">
        <Attribute Comment="up to 64kb per exploit" Type="Kind">Huge</Attribute>
        <Attribute Type="State">Heap</Attribute>
      </Operand>
      <Operand Name="Size">
        <Attribute Type="Kind">Used</Attribute>
      </Operand>
      <Operation>
        <Attribute Type="Mechanism">Sequential</Attribute>
        <Attribute Comment="ssl/d1_both.c#L1487&#xD;&#xA;ssl/t1_lib.c#L2620" Type="Source Code">Third-Party</Attribute>
        <Attribute Type="Execution Space">Userland</Attribute>
      </Operation>
    </Attributes>
  </Weakness>
  <Failures ClassType="_FLR">
    <Cause Comment="see also: https://git.openssl.org/?p=openssl.git;a=blob;f=ssl/t1_lib.c;h=c5c805cce286d12d81c5fdccfe9
    <Failure Class="IEX" Comment="(confidentiality loss)" />
  </Failures>
</BFCVE>
```

**Fig. 19.** CVE-2014-0160.bfcve BF Specification of Heartbleed in XML format.

error/fault type propagation rule, which to a large extent also restricts to meaningful operation flow, as the BF classes are developed to adhere to the BF Bugs Models specific for their BF class types.

The *Generate BF Description* button displays a draft BF description based on the selected values from the five TreeView controls and *Comment* TextBoxes.




| Class Type | Definition |
|---|---|
| Input/Output Check (_INP) | Input/Output Check (_INP) class type – Bugs/Faults allowing an injection exploit. |
| Memory Corruption/Disclosure (_MEM) | Memory Corruption/Disclosure (_MEM) class type – Bugs/Faults allowing a memory corruption/disclosure exploit. |
| **Class** | **Definition** |
| Data Verification (DVR) | Data Verification (DVR) class – Data are verified (semantics check) or corrected (assign, remove) improperly. |
| Memory Addressing (MAD) | Memory Addressing (MAD) class – The pointer to an object is initialized, repositioned, or reassigned to an improper memory address. |
| Memory Use (MUS) | Memory Use (MUS) class – An object is initialized, read, written, or cleared improperly. |
| **Operation** | **Definition** |
| Verify | Verify operation – Check data semantics (proper value/meaning) in order to accept (and possibly correct) or reject it. |
| Reposition | Reposition operation – Change the pointer to another position inside its object. |
| Read | Read operation – Use the value of an object's data. |
| Clear | Clear operation – Change the meaningful value of an object to a non-meaningful one (e.g. via zeroization) – e.g. before object deallocation. |
| **Cause** | **Definition** |
| ... | ... |

**Fig. 20.** Graphical representation of the CVE-2014-0160 (Heartbleed) BFCVE specification. For simplicity, only part of the table with related BF taxons definitions is visualized.

The BF tool demonstrates how the BF taxonomy, causation rules, and propagation rules tie together into the strict BF Formal Language.

For details on the BF tool refer NIST SP xxx-xxxF BF Databases, Tools, and APIs.

## 8.3. BF Datasets

### 8.3.1. BFCWE

There are 938 CWE weakness types as of March 2024 [1]. NVD uses the 130 "most commonly seen weaknesses" [35] for labeling CVEs, but it might also list other CWEs as assigned by third-party contributors.

Of the 938 CWEs, 72 map to BF Data Type (_DAT) [19], 157 – to BF Input/Output Check (_INP) [18], and //xxxxxxxxx check if not 61// 60 – to BF Memory Corruption/Disclosure (_MEM) [17] bugs class types. These 289 unique CWEs form 30% of the CWE repository and provide the base for systematic creation of a comprehensively labeled weaknesses BFCWE dataset.

The methodology utilizing BF is as follows:

1. CWE2BF Mappings: Basic software security weaknesses research and meticulous analysis of the natural language descriptions of all data type, input/output check, and memory related CWEs (as well as of relevant code examples and CVEs) is conducted to create CWE2BF mappings by weakness operation, error, final error and then by detailed BF *(bug, operation, error)*, *(fault, operation, error)*, or *(fault, operation, exploitable error)* weakness triples (see [17–21]).
2. BF Specifications: The BFCWE tool generates BFCWE formal specifications as entries of the BFCWE software security weakness types dataset.
3. Graphical Representations: The BFCWE tool generates graphical representations for enhanced understanding of the CWE2BF mappings (by operation, error, final error, and complete weakness triples) with parent-child CWE relations, and of the BFCWE formal specifications.

As the BFCWE specifications are in essence partial BFCVE specifications, the matrix and the dataset are also continuously enriched from newly developed BF specifications of CVEs and other reported security vulnerabilities.

All developed BFCWE specifications are added to the comprehensively labeled BFCWE dataset (query it via the BFCWE API).

The BFCWE dataset could augment the CWE repository and the NVD database by adding the formal BF specifications of possible BF weakness triples for each CWE entry. However, BF has the expressive power to describe any security weakness and not only the types listed in CWE. BF has its own database, BFWeaknessDB of formal weakness specifications.

For details on the BFCWE dataset refer NIST SP xxx-xxxG BF Weakness and Vulnerability Datasets.

### 8.3.2. BFCVE

The current state of the art in cybersecurity — labeling CVEs with CWEs — is not keeping up with the modern cybersecurity research and application requirements for comprehensively labeled datasets. As a formal classification system of software security bugs and related software faults enabling unambiguous specification of security weaknesses and vulnerabilities, BF offers a prominent new approach toward systematic creation of venerability datasets labeled with the BF taxonomy.

There are over 170 000 CVEs labeled with CWEs by NVD as of March 2024. Most of them map by final exploitable weakness to the BF Data Type (_DAT) [19], Input/Output (_INP) [18], and Memory Corruption/Disclosure (_MEM) [17] bug/weakness types: 3 329 map to _DAT, 63 172 – to _INP, and 40 454 – to _MEM. These 106 955 unique CVEs represent 63% of the CVEs labeled with CWEs by NVD, providing the base for systematic creation of a comprehensively labeled BFCVE security vulnerability dataset.

The methodology utilizing BF is as follows:

1. Code with Fix: The BFCVE tool retrieves CVEs with assigned CWEs for which *Code with Fix* is available.
2. Backwards State Tree (see also Sec. 3.3): The BFCVE tool generates possible chains of weaknesses for a vulnerability – backwards by identified failure and some or all the elements of the final weakness – *(fault, operation, final exploitable error)* or *(bug, operation, final exploitable error)* in the case of a one weakness vulnerability – utilizing the BF taxonomy, and syntax and semantic rules.
3. BF Specifications: Deep code analysis and the BF GUI tool are used to filter the generated chains and complete the unambiguous BF vulnerability specifications.
4. Graphical Representations: The BFCVE tool generates graphical representations for enhanced understanding of the BF vulnerability specifications as entries for the BFCVE software security vulnerability dataset.
5. CWE Assignments: The BFCVE tool identifies, refines, and recommends a CWE(s) for NVD assignment. Although this step may seem illogical, as a BF specification already provides comprehensive information, it may be useful when comparing CWE-based testing tool reports or if a more appropriate CWE is identified.

This approach would also guide vulnerability specifications for which code is not available — information from the existing BF vulnerability specifications would fuel their analyses. Analogously, going backwards from each one of these would reveal options for previous weaknesses until a weakness with a bug as a cause is reached. For example, going backwards from *(Wrong Size, Reposition, Over Bounds Pointer)*, reveals the previous causing weakness is a BF Data Validation (DVL) initial weakness among: *(Missing/Erroneous Code/Under-Restrictive Policy/Over-Restrictive Policy, Verify/Correct, Wrong Value/Inconsistent Value)*.

Developed BFCVE specifications are added to the comprehensively labeled BFCVE dataset (query it via the BFCVE API). The BF semantic graphs and matrices and the datasets are also continuously enriched from newly developed formal BF specifications of CVEs and other reported software security vulnerabilities.

The BFCVE dataset could augment the CVE repository and the NVD database by supplying the formal BF specifications of CVE entries. However, BF has the expressive power to describe any security vulnerability and not only the listed in CVE. BF has its own database, BFVulDB of formal vulnerability specifications.

For details on the BFCVE dataset refer NIST SP xxx-xxxG BF Weakness and Vulnerability Datasets.

### 8.4. BF Vulnerability Classifications

The BF Vulnerability Classification Model (see Fig. 21) shows how the BF Taxonomy and the BF Tools can be utilized for generation of BFCWE and BFCVE datasets (see also Sec. 8.3.1 and Sec. 8.3.2) and by themselves or by querying other vulnerability repositories for generation of diverse vulnerability classifications for deeply focused security research. Key part for the generation of the BFCVE dataset is the security experts and/or LLMs code analysis needed to generate complete BF vulnerability specifications. Research in this direction is undergoing by several security teams.

Security vulnerabilities could be classified by common bugs (root causes) – e.g., by wrong data type used in a variable declaration, by propagating faults, or by common exploitable errors – e.g., buffer overflow or injection. Detailing over the possible BF operation and operand attributes may be used to understand the severity of the weaknesses and how they related to commonly used scores as CVSS and EPSS. They could be classified also by the number of underlying weaknesses or by entirely same BF specifications. Intriguing classifications by BF classes and Common Platform Enumeration (CPE) data may reveal for example systematic data type safety, input/output safety, and memory safety coding problems related to particular vendors and products. The BF Vulnerability Classifications API will be providing access to the latest developments.

Such focused BF vulnerability classifications could contribute to the deeper analysis and refined understanding of security weaknesses and vulnerabilities and towards informed development of effective countermeasures.

### References

[1] MITRE (2006-2024) Common Weakness Enumeration (CWE). Available at https://cwe.mitre.org.
[2] MITRE (1999-2024) Common Vulnerabilities and Exposures (CVE). Available at https://cve.mitre.org.

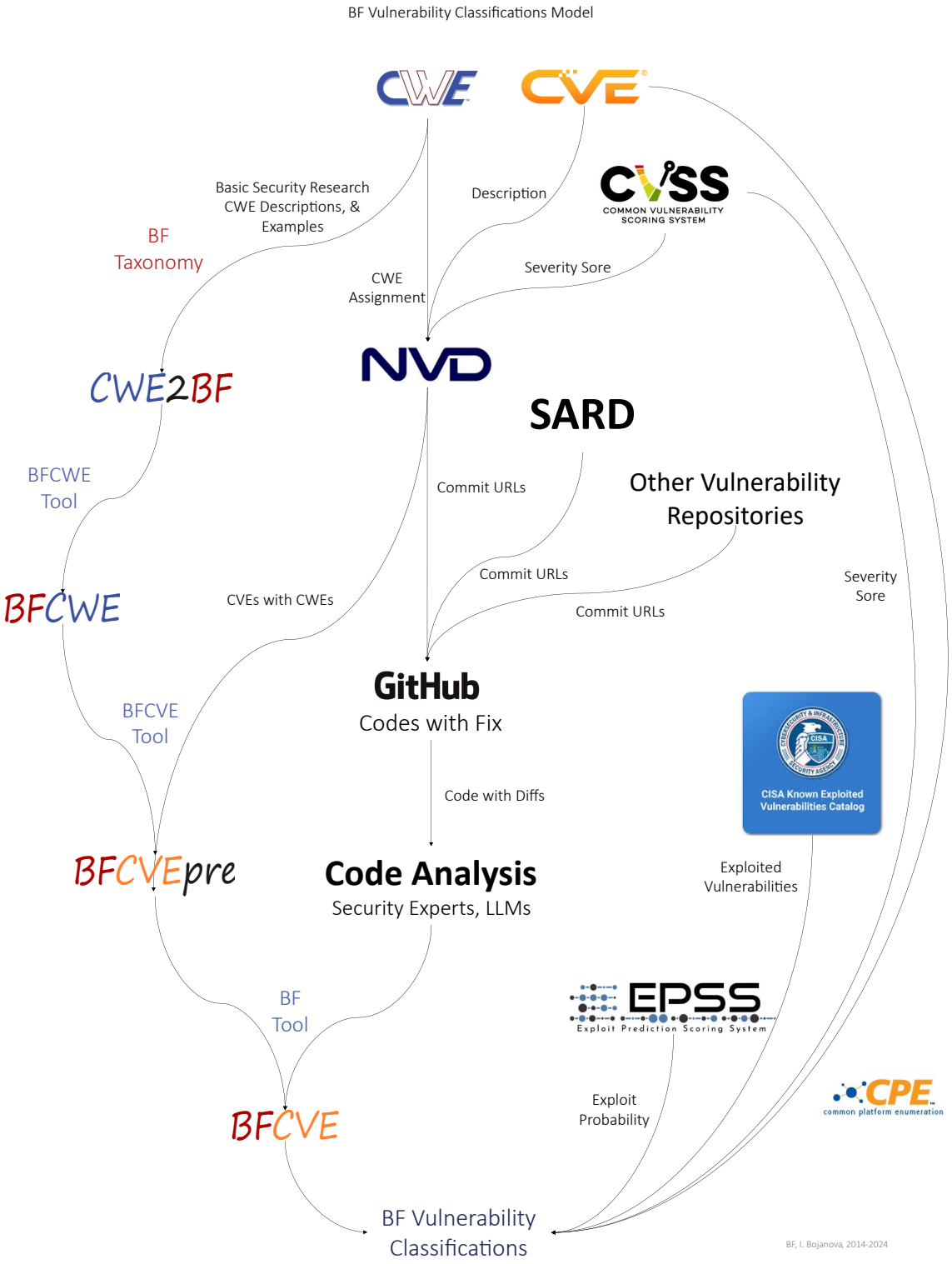



**Fig. 21.** BF Vulnerability Classifications Model.

[3] NIST (1999-2024) National Vulnerability Database (NVD). Available at https://nvd.nist.gov.

[4] CISA (2021-2024) Known Exploited Vulnerabilities Catalog (KEV). Available at https://www.cisa.gov/known-exploited-vulnerabilities-catalog.

[5] Irena Bojanova, C E Galhardo (Jan.-Feb. 2023) Bug, Fault, Error, or Weakness: Demystifying Software Security Vulnerabilities. *IEEE IT Professional*, Vol. 25, 1, p 7–12. https://doi.org/10.1109/MITP.2023.3238631

[6] MITRE CVE History. Available at https://www.cve.org/About/History.

[7] David E Mann, S M Christey (Jan. 8, 1999) Towards a Common Enumeration of Vulnerabilities. Available at https://www.cve.org/Resources/General/Towards-a-Common-Enumeration-of-Vulnerabilities.pdf.

[8] MITRE CWE History. Available at https://cwe.mitre.org/about/history.html.

[9] Yan Wu, I Bojanova, Y Yesha (2015) They Know Your Weaknesses - Do You?: Reintroducing Common Weakness Enumeration. *CrossTalk (The booktitle of Defense Software Engineering)*, pp 44–50. Available at https://web.archive.org/web/20180425211828id_/http://static1.1.sqspcdn.com/static/f/702523/26523304/1441780301827/201509-Wu.pdf?token=WJEmDLgmpr3rIZHriubA20L%2F1%2F4%3D.

[10] Drew Malzahn, Z Birnbaum, C Wright-Hamor (2020) Automated Vulnerability Testing via Executable Attack Graphs. *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)* (IEEE), p 1–10. https://doi.org/10.1109/CyberSecurity49315.2020.9138852

[11] Irena Bojanova (Dec. 9, 2014) Formalizing Software Bugs. Available at https://www.nist.gov/publications/formalizing-software-bugs.

[12] Irena Bojanova (Apr. 8, 2015) Towards a 'Periodic Table' of Bugs. Available at https://www.nist.gov/publications/towards-periodic-table-bugs.

[13] Irena Bojanova, P E Black, Y Yesha, Y Wu (2016) The NIST Bugs Framework (BF): A Structured Approach to Express Bugs. *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp 175–182. https://doi.org/10.1109/QRS.2016.29

[14] Irena Bojanova, P E Black, Y Yesha (2017) Cryptography classes in NIST Bugs Framework (BF): Encryption bugs (ENC), verification bugs (VRF), and key management bugs (KMN). *2018 IEEE 45nd Annual Computer, Software, and Applications Conference (COMPSAC)*, pp 1–8. https://doi.org/10.1109/STC.2017.8234453

[15] Irena Bojanova, Y Yesha, P E Black (2018) Randomness Classes in NIST Bugs Framework (BF): True-Random Number Bugs (TRN) and Pseudo-Random Number Bugs (PRN). *2018 IEEE 45nd Annual Computer, Software, and Applications Conference (COMPSAC)*, pp 738–745. https://doi.org/10.1109/COMPSAC.2018.00110

[16] Irena Bojanova, Y Yesha, P E Black, Y Wu (2019) Information Exposure (IEX): A New Class in the NIST Bugs Framework (BF). *2019 IEEE 45nd Annual Computer, Software, and Applications Conference (COMPSAC)*, pp 559–564. https://doi.org/10.1109/COMPSAC.2019.00086

[17] Irena Bojanova, C E Galhardo (2021) Classifying Memory Bugs Using Bugs Frame-

work Approach. *2021 IEEE 45nd Annual Computer, Software, and Applications Conference (COMPSAC)*, pp 1157–1164. https://doi.org/10.1109/COMPSAC51774.2021.00159

[18] Irena Bojanova, C E Galhardo (2021) Input/Output Check Bugs Taxonomy: Injection Errors in Spotlight. *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp 111–120. https://doi.org/10.1109/ISSREW53611.2021.00052

[19] Irena Bojanova, C E Galhardo, S Moshtari (2022) Data Type Bugs Taxonomy: Integer Overflow, Juggling, and Pointer Arithmetics in Spotlight. *2022 IEEE 29th Annual Software Technology Conference (STC))*, pp 192–205. https://doi.org/10.1109/STC55697.2022.00035

[20] Irena Bojanova, C E Galhardo (Mar.-Apr. 2023) Heartbleed Revisited: Is it just a Buffer Over-Read? *IEEE IT Professional*, Vol. 25, 2, pp 83–89. https://doi.org/10.1109/MITP.2023.3259119

[21] Irena Bojanova (Jan.-Feb. 2024) Comprehensively Labeled Weakness and Vulnerability Datasets via Unambiguous Formal NIST Bugs Framework (BF) Specifications. *IEEE IT Professional*, Vol. 26, 1, pp 60–68. https://doi.org/10.1109/MITP.2024.3358970

[22] Peter Mell, K Kent, S Romanosky (2006) Common vulnerability scoring system. Available at https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=50899.

[23] FIRST (2015-2024) Common vulnerability scoring system special interest group. Available at https://www.first.org/cvss.

[24] Constable S (2024) Chips Salsa: Industry Collaboration for new Hardware CWEs. Available at https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Chips-Salsa-Industry-Collaboration-for-new-Hardware-CWEs/post/1575521.

[25] Irena Bojanova, J J Guerrerio (Sept.-Oct. 2023) Labeling Software Security Vulnerabilities. *IEEE IT Professional*, Vol. 25, 5, pp 64–70. https://doi.org/10.1109/MITP.2023.3314368

[26] Irena Bojanova (2020-2024) NIST Bugs Framework (BF), BF Tool. Available at https://usnistgov.github.io/BF/info/tools/bf-tool.

[27] Irena Bojanova, NIST (2014-2024) Bugs Framework (BF) Website. Available at https://usnistgov.github.io/BF;https://samate.nist.gov/BF.

[28] Donald Knuth (1968) Semantics of context-free languages. *Math. Systems Theory 2*, p 127–145. https://doi.org/10.1007/BF01692511

[29] GitHub (2008) GitHub. Available at https://github.com.

[30] NIST (2005-2024) Software Assurance Reference Dataset (SARD). Available at https://samate.nist.gov/SARD.

[31] FIRST (2021-2024) Exploit Prediction Scoring System (EPSS). Available at https://www.first.org/epss.

[32] Irena Bojanova (2020-2024) NIST Bugs Framework (BF), BFCVE Tool. Available at https://usnistgov.github.io/BF/info/tools/bfcve-tool.

[33] Irena Bojanova (2020-2024) NIST Bugs Framework (BF), BFCWE Tool. Available

at https://usnistgov.github.io/BF/info/tools/bfcwe-tool.
[34] Chen Y, at all (2023) DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. Available at https://github.com/wagner-group/diversevul.
[35] MITRE (2023) Weaknesses for Simplified Mapping of Published Vulnerabilities. Available at https://cwe.mitre.org/data/definitions/1003.html.